Разработка алгоритмов обучения с подкреплением для задачи манипуляции шарнирной моделью руки

Курсовая работа

Содержание

Введение

Обучение с подкреплением – разновидность машинного обчуения, отличающаяся отсутствием заготовленного набора данных. Таким образом, данные создаются в результате взаимодействия алгоритма с некоторой средой, реальной или виртуальной.

Существует множество задач, в которых подготовить данные для алгоритма заранее крайне проблематично. Например, многие задачи, связанные с обратной связью пользователей: рекомендательные системы, контекстная реклама. Кроме того, можно сочетать обучение с учителем и обучение с подкреплением, что открывает новые возможности во всех сферах применения машинного обучения.

Особенно много применения рассматриваемому подходу находится в областях искусственного интеллекта и робототехники. Различные боты учатся разговаривать с людьми, улучшая свою речь в процессе общения; искуственный интеллект побеждает чемпионов мира по шахматам и го; беспилотные автомобили учатся распознавать окружающие объекты; искусственный интеллект учится играть в прятки [1] – это примеры задач, в которых собрать полный набор данных крайне проблематично, зато обучение с подкреплением в них широко используется.

Данная работа рассматривает задачу разработки манипулятора, который мог бы выполнять функции человеческой руки, и обучение его простейшим действиям, таким как перемещение предметов. Подобные наработки можно использовать во многих областях, более того, роботизированные руки уже широко применяются, например, на сборочных линиях заводов; для дистанционных хирургических операций [2]; для выполнения различных работ в опасных и труднодоступных для человека местах, например, в космосе; для протезирования [3]. Стоит отметить, что во многих из этих сфер исследования идут довольно активно, и существует простор для улучшения сущетвующих разработок.

1. Основы обучения с подкреплением

1.1 Основные понятия

Под **агентом** понимается алгоритм, решающий задачу. Под **средой** понимается окружающий мир либо симуляция, которая реагирует на **действия** агента. Таким образом, агент предпринимает определенные действия в зависимости от состояния среды.

Цель агента – найти такую **стратегию**, т. е. правило применения действий, которая будет максимизировать **награду** – один из параметров среды, который обозначает, насколько желательны результаты, к которым привели действия агента.

Задача обучения с подкреплением, как правило, формулируется как марковский процесс принятия решений, состоящий из:

- множества состояний среды S;
- множества действий A;
- множества наград, являющегося подмножеством $\mathbb{R}$;

В котором выполняется предположение Маркова:

$$P(s_{t+1}|s_t, a_t, \ldots, s_{t-k}, a_{t-k}) = P(s_{t+1}|s_t, a_t)$$

$$\forall t, k \in \mathbb{R}, t > k, a_l \in A, s_l \in S, l = \overline{t-k, t}$$

Приведем формальное описание взаимодействия агента и среды. В произвольный момент времени t агент характеризуется состоянием $s_t \in S$ и множеством возможных действий $A(s_t)$. Выбирая действие $a \in A(s_t)$, он переходит в состояние s_{t+1} и получает награду r_t. Основываясь на таком взаимодействии с окружающей средой, агент должен выработать стратегию $\pi : S \to A$, которая максимизирует величину $R = \sum_t r_t$ в случае процесса, имеющего терминальное состояние, или величину: $R = \sum_t \varepsilon^t r_t$ $(0 < \varepsilon < 1)$ для процесса без терминальных состояний.

Важные проблемы обучения с подкреплением:

- Баланс между использованием уже полученных знаний и исследованием среды. Если использовать полученные знания слишком интенсивно, велик шанс сойтись к стратегии, далекой от оптимальной.
- Баланс между краткосрочной и долгосрочной выгодой. Важно выбрать такую награду, чтобы алгоритм стремился прийти к большей, пусть и долгосрочной, выгоде.

- Определение действий, бывших причинами награды. Бывает, что агент совершает правильное действие в момент времени x, но получает награду за него только в момент $x + 10$.
- Определение множества состояний. Наблюдения могут быть неполными, так что у агента не будет всей нужной информации для принятия решений.
- Переобучение.
- . . . И многие другие.

1.2 Постановка задачи

Разработка симулятора сведется к следующим шагам:

1. Разработать физически корректную симуляцию окружающей среды;
2. Разработать дизайн манипулятора, которым можно управлять при помощи сигналов;
3. Обучить алгоритм посылать необходимые сигналы в зависимости от состояния среды.

Для начала мы поставим перед алгоритмом простую задачу. В начальном состоянии манипулятор будет иметь коробку в своей руке. Мы хотим, чтобы манипулятор поставил коробку на стол; существует возможность уронить коробку мимо стола, что является нежелательным исходом.

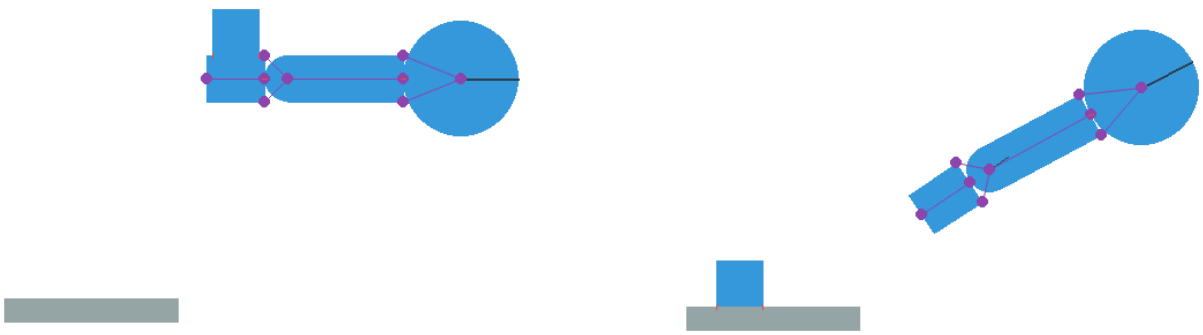

Рисунок 1.1 — Иллюстрация начального и желаемого конечного положения

Определим параметры среды.

Множество состояний бесконечно; одно состояние описывает изменяемые параметры (такие как координаты и вектор скорости) всех объектов в симуляции (таких как составные части манипулятора и коробка).

Множество действий дискретно; одно действие представляет собой сигнал, который можно послать моторам манипулятора.

Награду будем давать за каждую секунду, которую коробка касается стола. Таким образом, чем дольше коробка удерживается на столе, тем лучше. Мы сможем ранжировать по предпочтительности следующие ситуации:

1. коробка упала сразу;
2. коробка коснулась стола, а затем упала;
3. коробка устойчиво лежит на столе.

Стоит отметить, что алгоритмы, которые мы разработаем, можно обобщить и для более сложных задач. Например, следующим шагом можно поставить такую задачу: в начальном положении коробка стоит на одном столе, ожидается, что манипулятор поднимет ее со стола и переставит на другую полку. Выбранная на данном этапе простая задача позволяет увеличить скорость разработки симулятора и обучения алгоритма.

2. Разработка симулятора

Важной и интересной с точки зрения проектирования задачей является разработка физической симуляции задачи. Здесь и далее используется язык программирования Python; полный исходный код находится в Github-репозитории: *github.com/elizabethfeden/mechanic-arm*.

2.1 Программирование физики симуляции

Для программирования физики воспользуемся библиотекой Pymunk [4]. Она позволяет создавать физические объекты разных форм (например, круг и прямоугольник), задавать им физические параметры (масса, эластичность, плотность) и вычислять скорость их перемещения, силу столкновения и прочее.

```
def _create_shape(
    position: Tuple[float, float],
    shape_constructor: Callable[[pymunk.Body], pymunk.Shape],
    mass: Optional[float]
) -> pymunk.Shape:
  body.position = position
  shape = shape_constructor(body)
  shape.mass = mass
  shape.elasticity = 0.999
  shape.friction = 1
  return shape

def create_rect(
    position: Tuple[float, float], size: Tuple[float, float],
    mass: float = 10
) -> pymunk.Poly:
  return _create_shape(position,
                       lambda body: pymunk.Poly.create_box(body, size),
                       mass)

box = create_rect(box_position, BOX_SIZE, BOX_WEIGHT)
```

Листинг 2.1: Создание коробки в pymunk

Некоторые особенности, связанные с этим этапом:

Визуализация. Пускай для обучения агенту не требуется видеть, как выглядит симуляция – достаточно числовых параметров объектов, – нам будет интересно пронаблюдать действия агента. Для этого воспользуемся библиотекой для создания оконных приложений Pygame [5]. Заметим, что Pymunk и Pygame отлично интегрированы, поэтому реализация визуализации симулятора не является трудной задачей.

Частота обновления. После каждого действия агента мы будем пересчитывать параметры физических объектов. Существует, однако, интересный вопрос: как часто стоит давать агенту совершать действия – и, соответственно, на сколько миллисекунд вперед просчитывать параметры после каждого действия? С одной стороны, мы, очевидно, хотим давать агенту действовать часто, чтобы сымитировать непрерывный поток времени. С другой, маленькие шаги во времени могут увеличить сложность и длительность обучения агента.

Для примера сравним результаты обучения методом кросс-энтропии с разным шагом времени.

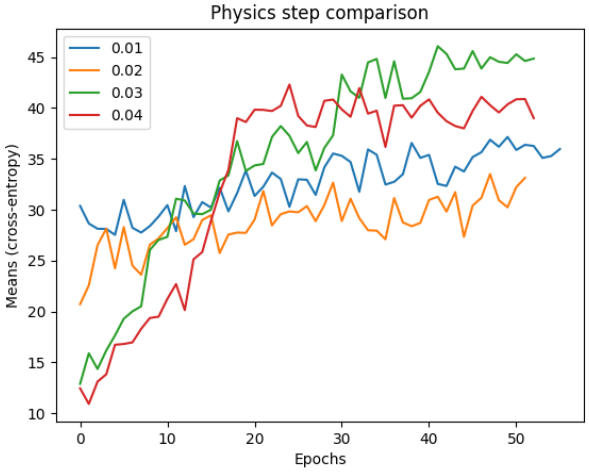

Рисунок 2.1 — Сравнение средней награды для разных размеров шага физических вычислений при обучении методом кросс-этнропии

Видим, что с большим шагом агент обучается за меньшее число эпох за счет большей простоты симуляции; с другой стороны, что интересно, агент с самым маленьким шагом также обучается достаточно хорошо: вероятно, потому, что в этом случае симуляция генерирует больше данных. Отметим также, что обучение одной эпохи модели для шага в 0.01 длится примерно в два раза больше, чем для шага в 0.02. Далее будем использовать шаг 0.03.

Подбор параметров. Важно подобрать физические параметры так, чтобы поведение симуляции было похоже на реальный мир. К сожалению, в данном случае приходится руководствоваться здравым смыслом и методом подбора. В процессе подбора можно наткнуться на случаи интересного поведения симуляции.

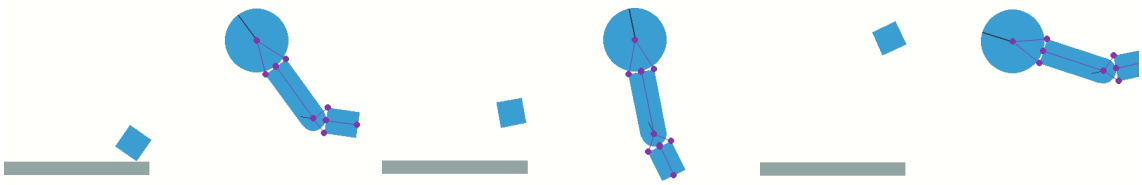

Рисунок 2.2 — Пример неправильного подбора параметров: у стола слишком высокая эластичность[1]

2.2 Дизайн манипулятора

Разработаем достаточно функциональную и удобно управляемую механическую руку. Основываться будем на двух вариантах составляющих частей: круглые элементы – моторы, которые могут вращаться по часовой стрелке и против, а также фиксировать свое положение; прямоугольные элементы – неуправляемые части механизма, которые несут роль соединений между моторами и рычагов для взаимодействия с посторонними объектами.

Подбор пропорций и физических параметров. Важно подобрать соответствующие размеры и массы составляющих частей, а также мощность моторов.

Рисунок 2.3 — Пример неправильного подбора параметров: основной мотор слишком маленький, из-за чего невозможно равномерно поворачивать механизм[2]

[1] Данное изображение можно просмотреть в gif-формате по ссылке: *github.com/elizabethfeden/mechanic-arm/blob/main/gifs/bad-parameters.gif*

[2] Ссылка на gif-формат: *github.com/elizabethfeden/mechanic-arm/blob/main/gifs/bad-proportions.gif*

Архитектура креплений. Различные элементы требуется соединять между собой при помощи соединений так, чтобы они были достаточно прочны и устойчивы, но при этом не препятствовали движению других элементов. Разработка подходящей архитектуры оказалась нетривиальной задачей.

Мы будем пользоваться двумя типами соединений: шарнирные (Pivot Joint) и штифтовые (Pin Joint). Реализацию физической логики их действия предлагает Pymunk.

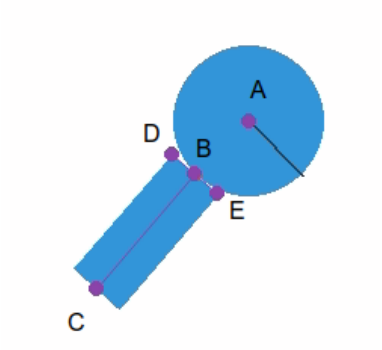

Рисунок 2.4 — Примеры креплений: A – шарнир, BC – штифт[1]

Далее будем рассматривать разные варианты крепления элементов. Будем пользоваться следующими обозначениями: A_c – для точки A означает, что она принадлежит кругу; B_r – точка B принаджежит прямоугольнику; $[A_cB_r]$ – точки A и B соединены шарниром; (A_cB_r) – точки A и B соединены штифтом.

Рассмотрим рисунок 2.4: $[A_c]$, (B_cC_r), $[B_cB_r]$, (B_cD_r), (B_cE_r). Данный дизайн оказался крайне неустойчивым: прямоугольник имеет недостаточно точек фиксации и из-за этого качается при передвижении.

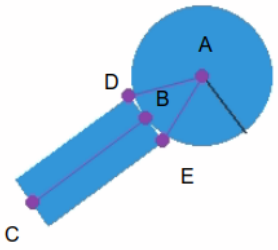

Рисунок 2.5 — Итоговый дизайн крепления к основному мотору[2]

Итоговый дизайн: $[A_c]$, (B_cC_r), (B_cB_r), (A_cD_r), (A_cE_r). Таким образом положение прямоугольника надежно зафиксировано относительно мотора, что позволяет эффективно им управлять.

[1] Ссылка на gif-формат: *github.com/elizabethfeden/mechanic-arm/blob/main/gifs/joint-design-2.gif*

[2] Ссылка на gif-формат: *github.com/elizabethfeden/mechanic-arm/blob/main/gifs/joint-design-3.gif*

Следующим этапом является крепление маленького мотора и управляемого им маленького прямоугольника («кисти»).

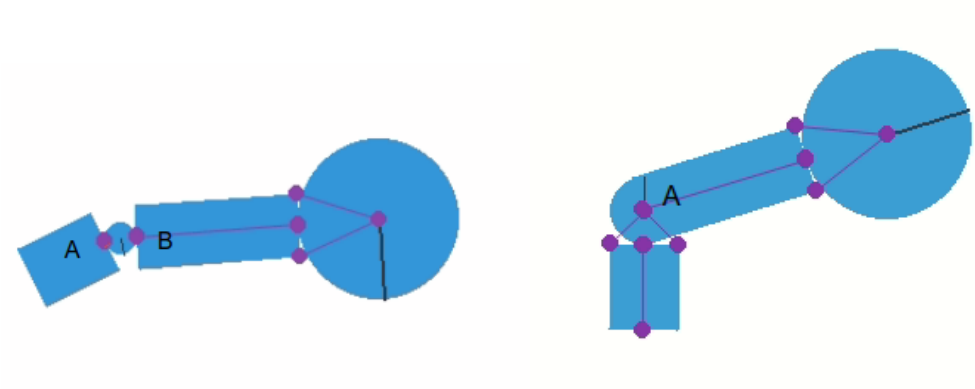

Рисунок 2.6 — Варианты крепления маленького мотора: (а) – неудачный[1], (б) – итоговый

Рисунок 2.6(а): $[A_cA_r]$, $[B_cB_r]$ – крайне неуправляемая и нестабильная конструкция. маленький мотор невозможно поворачивать, кисть плохо зафиксирована и при передвижении качается с большой инерцией.

Рисунок 2.6(б): $[A_cA_r]$ (коллизии между основным прямоугольником и маленьким мотором при вычислении физики игнорируются), крепление кисти к маленькому мотору аналогично креплению основного прямоугольника к основному мотору (рисунок 2.5).

Управление. Для каждого мотора предусмотрены четыре действия: вращать по часовой стрелке, против часовой, зафиксировать, ничего не делать. Итого в каждый момент времени агенту будут доступны шестнадцать вариантов действий.

```
# class Arm
def apply_force_to_circle(self, index: int, multiplier: float = 1):
  circle = self.dynamic_shapes[index]  # содержит и моторы, и прямоугольники
  # circle.body имеет тип Pymunk.Body
  circle.body.apply_force_at_world_point(
      (self.BASE_POWERS[index // 2] * multiplier, 0),
       circle.body.position + (0, circle.radius)
  )

# processing some actions
arm.apply_force_to_circle(arm.CIRCLE1_INDEX)  # вращать основной мотор проти
   в часовой стрелки
arm.apply_force_to_circle(arm.CIRCLE2_INDEX, -1)  # вращать маленький мотор
   по часовой стрелке
```

Листинг 2.2: Вращение мотора

[1]Ссылка на gif-формат: *github.com/elizabethfeden/mechanic-arm/blob/main/gifs/joint-design-4.gif*

В симуляции вращение мотора реализовано как применение силы к самой верхней его точке. При фиксации мотора его скорость и скорость соединенного с ним прямоугольника меняется на нулевую; это не совсем корректный с точки зрения физики переход, поэтому на больших скоростях возможны ошибки в симуляции.

Таким образом, была разработана достаточно стабильная и управляемая конструкция. Программно реализована возможность управлять ей в интерактивном режиме. Стоит отметить, что имеются некоторые программные ошибки; многие из них связаны со сложностями реализации физических вычислений.

3. Обучение методом кросс-энтропии

3.1 Описание метода

Пусть существует некоторая стратегия $\pi(s) = \{P(a|s) \mid a \in A\}$, причем $\sum_a P(a|s) = 1$. Для состояния s агент выбирает действие a_0 с вероятностью $P(a_0|s)$. Таким образом, в процессе обучения нашей задачей будет найти векторнозначную функцию $\pi(s)$. Найти ее значения для всех s невозможно, так как S в нащем случае непрерывно. Однако можно определить ее характер и аппроксимировать ее.

Идея метода кросс-энтропии [6] заключается в симуляции N сессий с фиксированной стратегией π; для каждой сессии хранится история действий, состояний и наград. Далее выбирается $M < N$ *элитных* сессий – сессий с наилучшими наградами. π обновляется в соответствии с историей действий и состояний элитных сессий. Таким образом предполагается, что через несколько эпох стратегия сойдется к оптимальной. Данный метод называется *эволюционным*, так как его идея основана на реальной теории эволюции и естественного отбора.

Недостатки метода:

- Требуется достаточно много удачных сессий, чтобы найти оптимальную стратегию. Соответственно, велик шанс и совершить много неудачных сессий. Это особенно проблематично для онлайн-обучения, т. е. для разновидности обучения с подкреплением, когда агент учится на своих действиях в реальном мире, а не в симуляции; в таком случае неудачи могут стоить реальных ресурсов.

- Агент с большой вероятностью не найдет стратегию для редких состояний, особенно для тех, что с большой вероятностью приводят к неудаче.

- На более поздних этапах обучения исследование крайне ограничено.

Достоинтва метода:

- Простота идеи и достаточно быстрая скорость схождения к достаточно хорошей стратегии. На поздних этапах обучения поиск оптимальной стратегии может занять много времени, но уже на средних этапах метод, как правило, показывает достаточно хорошие результаты.

- При оффлайн-обучении (т. е. обучении при помощи симуляции) метод отлично подходит для реализации параллельных вычислений и, соответственно, ускорения процесса обучения.

3.2 Реализация

3.2.1 Основная идея

Два главных аспекта метода, которые требуется реализовать, это симуляция одной сессии и анализ всех сессий для одной эпохи обучения.

```
# class CrossEntropyAgent
def action(self) -> int:
  probabilies = self.policy.predict_proba(self.state)
  action = np.random.choice(self.n_actions, p=probabilities)
  self.history_actions += [action]
  self.history_states += [self.state]
  return action
```

Листинг 3.1: Выбор действия

```
# class CrossEntropyFitter
def fit_epoch(self):
  agents = []
  rewards = []
  for _ in range(self.n_sessions):
    # симуляция одной сессии с текущей стратегией
    new_agent, reward = self._simulate_session()
    agents += [new_agent]
    rewards += [reward]

  rewards = np.array(rewards)
  indices = np.argsort(rewards)
  elites = [agents[i] for i in indices][-self.n_elites:]

  self._refit(elites)
```

Листинг 3.2: Одна эпоха обучения

3.2.2 Особенности работы с непрерывным пространством состояний

Если множество состояний дискретно, для обновления стратегии можно вручную пересчитывать вероятности:

$$P(a|s) = \frac{\sum_{s_i,a_i \in E} \mathbb{I}_{s_i=s} \cdot \mathbb{I}_{a_i=a}}{\sum_{s_i,a_i \in E} \mathbb{I}_{s_i=s}}$$

Где E – история элит, т. е. набор пар $\{s, a\}$, соответствующих действиям агента в элитных сессиях, $\mathbb{I}$ – индикатор.

Однако в нашем случае S непрерывно. Одним из возможных решений проблемы является обучение с учителем, а если быть точнее, подмножество *инкрементных* алгоритмов – тех, что могут обновлять параметры по мере поступления новых данных. Не все алгоритмы машинного обучения

инкрементны; решающее дерево требуется перестраивать каждый раз на всех данных, что может отрицательно сказаться на скорости обучения.

Мы будем использовать многослойный персептрон-классификатор (MLP classifier), реализованный в библотеке Sklearn [7]. Он представляет собой простейшую многослойную нейронную сеть, где функцией активации нейронов служит спрямленная линейная:

$$f(x) = \begin{cases} x, x > 0 \\ 0, x \leq 0 \end{cases}$$

Используем простейшую архитектуру с одним спрятанным слоем размером в 25 нейронов. Входные данные – состояние среды, выходные данные – предлагаемое действие. Каждую сессию будем дообучать классификатор на данных, полученных элитами.

Интересным моментом является инициализация параметров сети. В Sklearn по умолчанию веса нейронов инициализируются произвольным образом, и не существует прямой возможности задать их вручную. Для обучения с учителем это некритично, так как большое количество данных позволяет быстро настроить правильные параметры. Однако в нашем случае необходимо, чтобы вероятность выбора всех действий была одинаковой (или хотя бы близкой к одинаковой), т. е. $P(a|s) \approx \frac{1}{|A|}$. Если распределение неравномерно, то агент будет изначально предрасположен к одной стратегии; между действиями в разных сессиях не будет разнообразия, из-за чего результат каждый раз будет одинаковым.

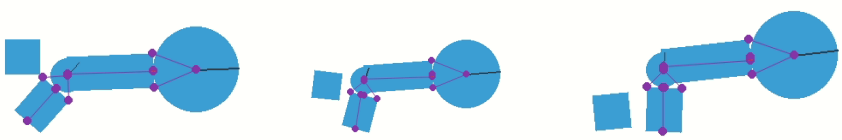

Рисунок 3.1 — Иллюстрация статегии, повторяющейся на протяжении 150 эпох из-за произвольной инициализации: агент мгновенно роняет коробку[1]

Возможным решением данной проблемы может быть использование фиксированной равновероятной стратегии на протяжении первых нескольких эпох. Однако каждая эпоха генерирует недостаточно много данных, чтобы можно было достаточно быстро переобучить параметры нейронной сети на требуемое распределение. К тому же, есть вероятность таким образом упустить некоторые множества состояний, веса для которых будут

[1] Ссылка на gif-формат: *github.com/elizabethfeden/mechanic-arm/blob/main/gifs/bad-init.gif*

неравномерны из-за того, что произвольно действующему агенту они ни разу не встречались.

Итоговым решением проблемы стало создание собственного класса, унаследованного от классификатора из Sklearn. В этом классе переопределен конструктор так, чтобы задавать нейронам веса, отражающие равномерное распределение. При этом важно оставить некоторую долю произвольной инициализации, иначе при обратном распространении градиента веса нейноронов будут меняться одинаково.

```
class MLPClassifier(sklearn.neural_network.MLPClassifier):
  # ...
  def _initialize(self, y, layer_units, dtype):
    super()._initialize(y, layer_units, dtype)
    self.coefs_[-1] = np.ones(
        (layer_units[-2], layer_units[-1])) / layer_units[-2]
    self.intercepts_[-1] = np.zeros((layer_units[-1],))
```

Листинг 3.3: Измененный конструктор класса MLPClassifier

Чтобы избежать описанной выше проблемы одинакового распространения градиента, мы задаем вручную только веса переходов из последнего скрытого слоя к выходному, но оставляем переходы от входного слоя к скрытому произвольными.

3.2.3 Параллельные вычисления

Для повышения скорости обучения потребовалось реализовать параллельное выполнение двух частей программы. Во-первых, симулировать каждую сессию можно абсолютно независимо от другой, что создает пространство для параллелизации вычислений. Во-вторых, была программно реализована возможность визуализировать симуляцию сессии агента со стратегией текущей эпохи; если запускать симуляцию в однопоточном режиме, придется просматривать визуализацию, а затем ждать, пока следующая эпоха обучится. Очевидно, это можно делать в разных процессах.

Отметим, что Python не поддерживает многопоточность, поэтому мы будем реализовавывать мультипроцессинг, когда вычисления выполняются в отдельных процессах операционной системы.

Параллельная визуализация и обучение. Здесь требуется создать два процесса, и передавать между ними информацию по каналам (pipes). Процессу, связанному с визуализацией, потребуется информация о текущей лучшей стратегии; в ответ он будет возвращать одну из метрик результата (скользящее среднее награды при повторной симуляции). Кроме непосредственно передачи информации, каналы позволяют синхронизировать процессы, чтобы для визуализации всегда существовала

стратегия, которую можно показывать, и чтобы процессы можно было корректно завершить и сохранить результаты выполнения.

Параллельная симуляция сессий. Для симуляции сессий достаточно создать `multiprocessing.Pool` – объект стандартной библиотеки Python, поддерживающий несколько процессов и раздающий им похожие задачи. Несмотря на прямолинейность данного подхода, стоит отметить неочевидную особенность. При создании нового процесса он получает доступ к памяти родительского процесса, в том числе и к текущему внутреннему значению генератора псевдослучайных чисел. Так как далее дочерние процессы работают независимо друг от друга, а изменение значений генератора фиксировано, они будут генерировать идентичные сессии (напомним, что для кросс-энтропии они различаются благодаря существованию вероятности выбрать различные действия для одного состояния). Решение данной проблемы простое (после создания задавать процессу новое состояние генаратора случайных чисел), однако обнаружить ее непросто.

В результате параллелизации получилось ускорить обучение более чем в два раза (при этом существует возможность ускорить сильнее, имея более мощный компьютер с большим количеством ядер):

Эпох	10	20
До параллелизации	2 мин 55 с	5 мин 47 с
После параллелизации	1 мин 34 с	2 мин 46 с

Таблица 3.1 — Сравнение времени обучения

3.3 Результаты

Во время обучения на каждой эпохе мы отслеживаем следующие метрики:

1. Средняя награда среди всех сессий. Позволяет оценить, насколько близко стратегия сошлась к оптимальной.

2. Медианная награда среди всех сессий.

3. Скользящее среднее награды при повторной симуляции. В то время как первые две метрики позволяют отобразить темпы и качество обучения, данная метрика больше приближена к реальной жизни и реальным испытаниям роботов. Для каждой эпохи мы повторно воспроизводим симуляцию (одновременно визуализируя ее) и запоминаем полученную награду. Скользящее среднее наград последних нескольких эпох позволяет оценить, насколько часто бы робот ошибался в реальности,

если бы мы воссоздали его с текущей стратегией и провели серию испытаний.

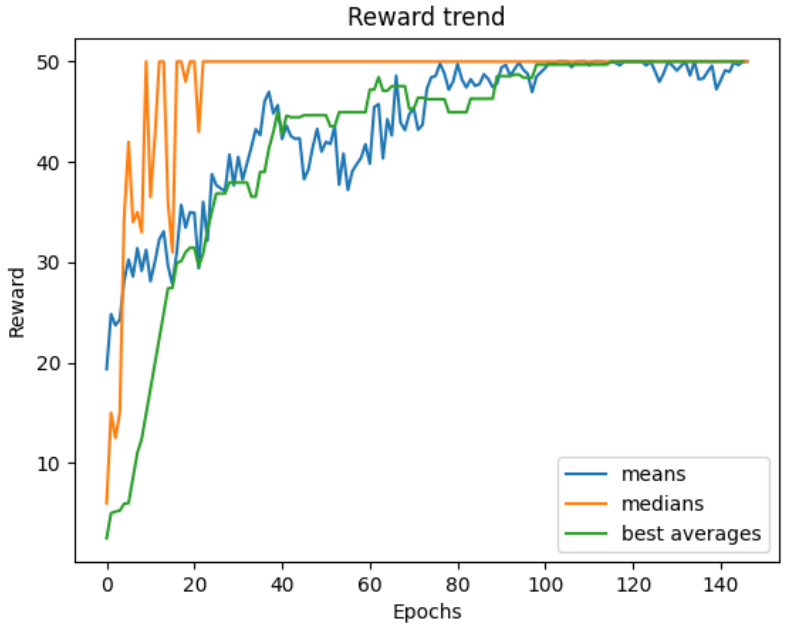

Рисунок 3.2 — График изменения метрик в процессе обучения

Видим, что в процессе обучения получилось сойтись к получению максимальной награды большинством сессий. Это означает, что мы получили стратегию, крайне близкую к оптимальной. Интересно отметить, что медиана сходится к максимальной награде особенно быстро; по этому можно судить, что мы можем при минимальных затратах получить приемлемую стратегию, для которой больше половины из симулированных на каждой эпохе сессий заканчиваются успешно.

Интересно также сравнить метрику скользящего среднего с агентом, в каждом состоянии выбирающим действие с одинаковой вероятностью (будем называть его случайным агентом)[1].

[1]Запись действий случайного агента можно просмотреть в gif-формате по ссылке: *github.com/elizabethfeden/mechanic-arm/blob/main/gifs/random-2.gif*

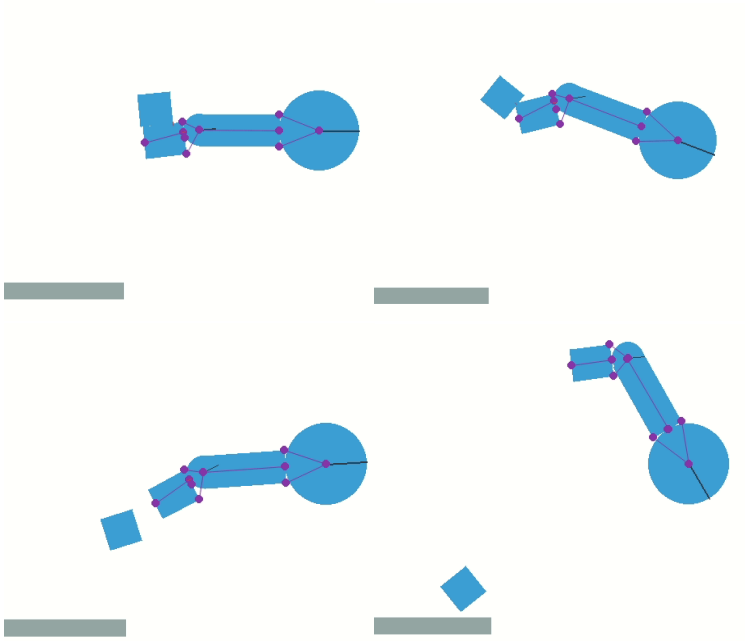

Рисунок 3.3 — Визуализация успешной стратегии[1]

Рисунок 3.4 — Сравнение скользящего среднего для случайного и обученного агента

Видим, что динамика скользящего среднего для случайного агента ожидаемо похожа на констнатную. Нельзя сказать, что случайный агент вообще неуспешен, но, тем не менее, легко видеть, что обучение дает лучшие и более стабильные результаты.

[1] Ссылка на gif-формат: *github.com/elizabethfeden/mechanic-arm/blob/main/gifs/optimal-2.gif*

Заключение

Обучение с подкреплением – интересная и актуальная разновидность машинного обучения, применяемая в маркетинге, финансах, медицине, робототехнике. В реальном мире нередки задачи, где собрать набор данных, требуемый для классичесиких методов машинного обучения, крайне проблематично; в то же время, обучение с подкреплением позволяет в том числе симулировать естественный процесс обучения новым навыкам и эволюции.

В данной работе был исследован процесс разработки симулятора для алгоритмов обучения с подкреплением и обучение методом кросс-энтропии для задачи манипуляции роботизированной рукой.

В ходе работы были успешно выполнены следующие задачи:

1. Реализация физического симулятора среды;
2. Разработка управляемого и стабильного роботизированного манипулятора;
3. Разработка интерактивного режима симуляции, где манипулятором можно управлять вручную;
4. Разработка случайного агента;
5. Разработка агента, обучающегося методом кросс-энтропии;
6. Параллелизация визуализации и обучения, а также симуляции сессий кросс-энтропии;
7. Обучение агента до оптимальной стратегии.

В итоге видим, что в нашей задаче кросс-энтропия позволяет сойтись к оптимальной стратегии; успешность обучения измерялась такими метриками, как средняя награда среди всех сессий, медианная награда среди всех сессий, скользящее среднее награды при повторной симуляции. По всем метрикам в процессе обучения результаты улучшаются, пока не достигают максимально возможных. Кроме того, обученный агент показывает хороший результат в сравнении со случайным.

В дальнейшем планируется усовершенствовать симуляцию (например, добавить симуляцию жидкостей и поставить задачу переливать из одной емкости в другую) и изучить другие методы машинного обучения, такие как Q-learning.

Список использованных источников

1. Emergent Tool Use From Multi-Agent Autocurricula / Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, Igor Mordatch // arXiv – 2020. №1909.07528 – 28 с.

2. Telerobotic neurovascular interventions with magnetic manipulation / Yoonho Kim, Emily Genevriere, Pablo Harker, Jaehun Choe, Marcin Balicki, Robert W. Regenhardt, Justin E. Vranic, Adam A. Dmytriw, Aman B. Patel, Xuanhe Zhao // Science Robotics. – 2022. Т. 7, № 65 – С. eabg9907.

3. Case-study of a user-driven prosthetic arm design: bionic hand versus customized body-powered technology in a highly demanding work environment / Wolf Schweitzer, Michael J. Thali, and David Egger // Journal of neuroengineering and rehabilitation – 2018. Т. 15, №1 – С. 1 – 27.

4. Pymunk [Электрон. ресурс]. – http://www.pymunk.org/en/latest/index.html

5. Pygame [Электрон. ресурс]. – https://www.pygame.org/news

6. Evolutionary Algorithms for Reinforcement Learning / John J. Grefenstette, David E. Moriarty, Alan C. Schultz // arXiv – 1999. – 36 с.

7. Sklearn [Электрон. ресурс]. – https://scikit-learn.org/stable/index.html