Лабораторная работа №10.

Поиск по дереву Монте-Карло (MCTS)

Поиск по дереву Монте-Карло (MCTS) — это эвристический алгоритм поиска, который показывает отличные результаты в таких сложных областях, как го и шахматы. Алгоритм строит дерево поиска, итеративно обходит его и оценивает его узлы с помощью моделирования методом Монте-Карло.

1. Предварительные действия

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


# Этот код создает виртуальный дисплей для рисования игровых изображений.
# Это не будет иметь никакого эффекта, если на вашей машине есть монитор.
import os
if type(os.environ.get("DISPLAY")) is not str or
len(os.environ.get("DISPLAY")) == 0:
    os.environ['DISPLAY'] = ':1'
```

Но прежде чем мы это сделаем, нам сначала нужно создать оболочку для сред Gym, позволяющую сохранять и загружать игровые состояния для облегчения возврата. Требуется дополнить код. Что должен выполнять код в каждой позиции <ВАШ КОД> указано в примечаниях к коду.

```
import gym
from gym.core import Wrapper
from pickle import dumps, loads
from collections import namedtuple

# контейнер для функции get_result ниже. Работает так же, как кортеж, но
красивее
ActionResult = namedtuple(
    "action_result", ("snapshot", "observation", "reward", "is_done",
"info"))


class WithSnapshots(Wrapper):
    """
    Создает оболочку, поддерживающую сохранение и загрузку состояний среды.
     Требуется для алгоритмов планирования.

     Этот класс будет иметь доступ к основной среде как self.env, например:
     - self.env.reset() # сбросить исходную среду
     - self.env.ale.cloneState() # сделать снимок для atari. загрузить с
помощью .restoreState()
     - ...

     Вы также можете использовать reset() и step() напрямую для удобства.
     - s = self.reset() # то же, что и self.env.reset()
     - s, r, done, _ = self.step(action) # то же, что и self.env.step(action)

     Обратите внимание, что, хотя вы можете использовать self.render(), это
создаст окно, которое нельзя замариновать.
     Таким образом, вам нужно будет вызвать self.close(), прежде чем
травление снова заработает.
    """

    def get_snapshot(self, render=False):
        """
```

```
        :returns: состояние среды, которое можно загрузить с помощью
load_snapshot
         Снимки гарантируют одинаковое поведение env при каждой загрузке.

         Предупреждение! Снимки могут быть произвольными вещами (строки,
целые числа, json, кортежи)
         Не рассчитывайте на то, что они будут строками рассола при
реализации MCTS.

         Примечание разработчика: убедитесь, что возвращаемый вами объект не
будет затронут
         все, что происходит с окружающей средой после ее сохранения.
         Вы не должны, например, возвращать self.env.
         В случае сомнений используйте pickle.dumps или deepcopy.

        """
        if render:
            self.render()  # close popup windows since we can't pickle them
            self.close()

        if self.unwrapped.viewer is not None:
            self.unwrapped.viewer.close()
            self.unwrapped.viewer = None
        return dumps(self.env)

    def load_snapshot(self, snapshot, render=False):
        """
        Загружает снимок как текущее состояние env.
         Не следует менять снапшот на месте (в случае сомнений — глубокая
копия).
        """

        assert not hasattr(self, "_monitor") or hasattr(
            self.env, "_monitor"), "не могу вернуться во время записи"

        if render:
            self.render()  # закрыть всплывающие окна, так как мы не можем
загрузиться в них
            self.close()
        self.env = loads(snapshot)

    def get_result(self, snapshot, action):
        """
        Удобная функция, которая
         - загружает снимок,
         - совершает действие через self.step,
         - и снова делает снимок :)

         :returns: следующий снимок, next_observation, награда, is_done,
информация

         По сути, он возвращает следующий снимок и все, что вернул бы
env.step.
        """

        < ВАШ КОД: зашрузка, коммит, взять снимок >

        return ActionResult(
               < ВАШ КОД: next_snapshot >,  # заполните переменные
               < ВАШ КОД: next_observation >,
               < ВАШ КОД: reward >,
               < ВАШ КОД: is_done >,
               < ВАШ КОД: info >,
        )
```

2. Пробы со снимками

Давайте проверим нашу обертку. Сначала сбрасываем окружение и сохраняем его, далее случайным образом играем какие-то действия и восстанавливаем наше окружение из снимка. Оно должно быть таким же, как наше предыдущее начальное состояние.

```
# создание окружения
env = WithSnapshots(gym.make("CartPole-v0"))
env.reset()

n_actions = env.action_space.n
```

```
print("initial_state:")
plt.imshow(env.render('rgb_array'))
env.close()

# создание первого снимка
snap0 = env.get_snapshot()
```

```
# проигрывание без снимков (быстрее)
while True:
    is_done = env.step(env.action_space.sample())[2]
    if is_done:
        print("Whoops! We died!")
        break

print("final state:")
plt.imshow(env.render('rgb_array'))
env.close()

# перезагрузка первичного состояния
env.load_snapshot(snap0)

print("\n\nAfter loading snapshot")
plt.imshow(env.render('rgb_array'))
env.close()

# получить результат (snapshot, observation, reward, is_done, info)
res = env.get_result(snap0, env.action_space.sample())

snap1, observation, reward = res[:3]

# второй шаг
res2 = env.get_result(snap1, env.action_space.sample())
```

3. MCTS: поиск по дереву Монте-Карло

Мы начнем с реализации класса Node — простого класса, который действует как узел MCTS и поддерживает некоторые шаги алгоритма MCTS.

```
assert isinstance(env, WithSnapshots)
```

Требуется дополнить код. Что должен выполнять код в каждой позиции <ВАШ КОД> указано в примечаниях к коду.

```
class Node:
    """Узел дерева для MCTS.

    Каждый узел соответствует результату выполнения определенного действия
```

```
(self.action)
    в определенном состоянии (самородитель) и, по сути, является одной рукой
в многоруком бандите, который
    мы моделируем в этом состоянии."""

    # метаданные:
    parent = None  # родительский узел
    qvalue_sum = 0.  # сумма Q-значений из всех визитов (numerator)
    times_visited = 0  # счетчик визитов (denominator)

    def __init__(self, parent, action):
        """
        Создает пустой узел без дочерних элементов.
         Делает это, совершая действие и записывая результат.

         :param parent: родительский узел
         :param action: действие для фиксации с родительского узла
        """

        self.parent = parent
        self.action = action
        self.children = set()  # набор дочерних узлов

        # получение результатов действий и их сохранение
        res = env.get_result(parent.snapshot, action)
        self.snapshot, self.observation, self.immediate_reward, self.is_done,
_ = res

    def is_leaf(self):
        return len(self.children) == 0

    def is_root(self):
        return self.parent is None

    def get_qvalue_estimate(self):
        return self.qvalue_sum / self.times_visited if self.times_visited !=
0 else 0

    def ucb_score(self, scale=10, max_value=1e100):
        """
        Вычисляет верхнюю границу ucb1, используя текущее значение и
количество посещений для узла и его родителя.

         :param scale: Умножает на это верхнюю границу. Из неравенства
Хёффдинга
                       предполагает, что диапазон вознаграждения равен [0,
шкала].
         :param max_value: значение, представляющее бесконечность (для
непосещенных узлов).

        """

        if self.times_visited == 0:
            return max_value

        # вычислить аддитивный компонент ucb-1 (добавлять к среднему
значению)
        # подсказка: вы можете использовать self.parent.times_visited для N
раз, когда узел рассматривался,
        # и self.times_visited для n посещений

        U = < ВАШ КОД: >

        return self.get_qvalue_estimate() + scale * U
```

```
    # MCTS шаги

    def select_best_leaf(self):
        """
        Выбирает лист с наивысшим приоритетом для расширения.
         Делает это, рекурсивно выбирая узлы с лучшим результатом UCB-1, пока
не достигнет листа.
        """
        if self.is_leaf():
            return self

        children = self.children

        # Выберите дочерний узел с наивысшей оценкой UCB. Возможно, вы
захотите реализовать некоторые эвристики
        # чтобы по-умному разорвать связи, хотя CartPole должен прекрасно
работать и без них.

        best_child = < ВАШ КОД: >

        return best_child.select_best_leaf()

    def expand(self):
        """
        Расширяет текущий узел, создавая все возможные дочерние узлы.
         Затем возвращается один из этих детей.
        """

        assert not self.is_done, "не может расширяться из терминального
состояния"

        for action in range(n_actions):
            self.children.add(Node(self, action))

        # Если вы реализовали какие-либо эвристики в select_best_leaf(), они
будут использоваться здесь.
        # В противном случае это эквивалентно выбору некоторого
неопределенного только что созданного дочернего узла.
        return self.select_best_leaf()

    def rollout(self, t_max=10 ** 4):
        """
        Играйте в игру от этого состояния до конца (готово) или за t_max
шагов.

         На каждом этапе выбирайте действие случайным образом (подсказка:
env.action_space.sample()).

         Вычислите сумму наград от текущего состояния до конца эпизода.
         Примечание 1: используйте env.action_space.sample() для выбора
случайного действия.
         Примечание 2: если узел является терминальным (self.is_done имеет
значение True), просто верните self.immediate_reward.

        """

        # установка среды в требуемое состояние
        env.load_snapshot(self.snapshot)
        obs = self.observation
        is_done = self.is_done

        < ВАШ КОД: обеспечить развертывание и рассчет вознаграждения >
```

```
        return rollout_reward

    def propagate(self, child_qvalue):
        """
        Использует дочернее Q-значение (сумму вознаграждений) для
рекурсивного обновления родителей.
        """
        # вычисленные узла Q-значения
        my_qvalue = self.immediate_reward + child_qvalue

        # обновление qvalue_sum и times_visited
        self.qvalue_sum += my_qvalue
        self.times_visited += 1

        # распространяться вверх
        if not self.is_root():
            self.parent.propagate(my_qvalue)

    def safe_delete(self):
        """безопасное удаление для предотвращения утечки памяти в некоторых
версиях Python"""
        del self.parent
        for child in self.children:
            child.safe_delete()
            del child
```

```
class Root(Node):
    def __init__(self, snapshot, observation):
        """
        создает специальный узел, который действует как корень дерева
         :snapshot: снимок (из env.get_snapshot), с которого можно начать
планирование
         :observation: последнее наблюдение за окружающей средой
        """

        self.parent = self.action = None
        self.children = set()  # выбор дочернего узла

        # root: загрузка снимка и наблюдение
        self.snapshot = snapshot
        self.observation = observation
        self.immediate_reward = 0
        self.is_done = False

    @staticmethod
    def from_node(node):
        """инициализирует узел как root"""
        root = Root(node.snapshot, node.observation)
        # copy data
        copied_fields = ["qvalue_sum", "times_visited", "children",
"is_done"]
        for field in copied_fields:
            setattr(root, field, getattr(node, field))
        return root
```

4. Основной цикл MCTS

Со всем, что мы реализовали, MCTS сводится к простому фрагменту кода. Требуется дополнить код. Что должен выполнять код в каждой позиции <ВАШ КОД> указано в примечаниях к коду.

```
def plan_mcts(root, n_iters=10):
    """
```

```
    строит дерево с поиском по дереву Монте-Карло для n_iters итераций
     :param root: узел дерева для планирования
     :param n_iters: сколько циклов select-expand-simulate-propagete сделать
    """
    for _ in range(n_iters):
        node = < ВАШ КОД: выберите лучший лист >

        if node.is_done:
            # Все развертывания с терминального узла пусты и, следовательно,
имеют 0 вознаграждений.
            node.propagate(0)
        else:
            # Разверните лучший лист. Выполните выкат из него. Распространите
результаты вверх.
            # Обратите внимание, что здесь у вас есть некоторая свобода
действий при выборе источника распространения.
            # Любой разумный выбор должен работать.

            < ВАШ КОД >
```

5. Планирование и запуск

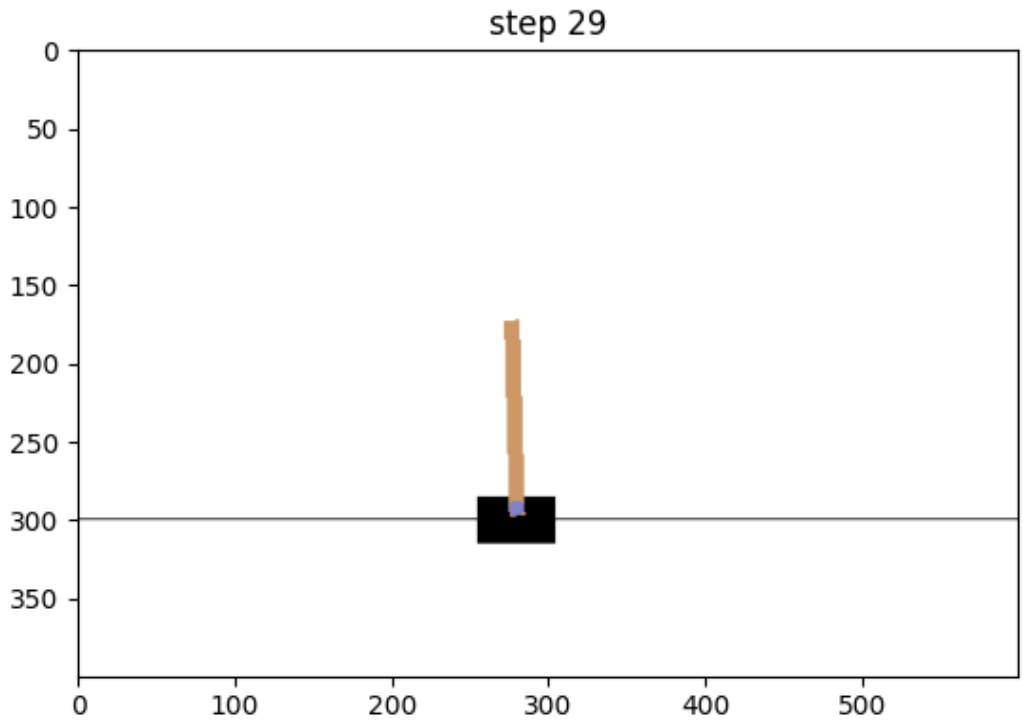
Давайте используем нашу реализацию MCTS, чтобы найти оптимальную политику. Требуется дополнить код. Что должен выполнять код в каждой позиции <ВАШ КОД> указано в примечаниях к коду.

```
env = WithSnapshots(gym.make("CartPole-v0"))
root_observation = env.reset()
root_snapshot = env.get_snapshot()
root = Root(root_snapshot, root_observation)
```

```
env = WithSnapshots(gym.make("CartPole-v0"))
root_observation = env.reset()
root_snapshot = env.get_snapshot()
root = Root(root_snapshot, root_observation)

# планируем от корня:
plan_mcts(root, n_iters=1000)


from IPython.display import clear_output
from itertools import count
from gym.wrappers import Monitor

total_reward = 0  # sum of rewards
test_env = loads(root_snapshot)  # env used to show progress

for i in count():

    # получение лучшего дочернего узла
    best_child = <ВАШ КОД: выбор дочернего узла с максимальным средним
вознаграждением>

    # выбор действия
    s, r, done, _ = test_env.step(best_child.action)

    # показ снимка
    clear_output(True)
    plt.title("step %i" % i)
    plt.imshow(test_env.render('rgb_array'))
    plt.show()
```

```
    total_reward += r
    if done:
        print("Finished with reward = ", total_reward)
        break

    # отбросить нереализованную часть дерева
    for child in root.children:
        if child != best_child:
            child.safe_delete()

    # объявить лучшего потомка новым корнем
    root = Root.from_node(best_child)

    assert not root.is_leaf(), \
        "У нас закончилось дерево! Нужно больше планирования! Попробуйте
вырастить дерево прямо внутри цикла"

    # Возможно, вы захотите запустить больше планов здесь
    # <ВАШ КОД>
```

Вывод:



Задания к лабораторной работе

1. Написать свой код к разделам 1 - 8 согласно заданиям для всех фрагментов, помеченных как: <ВАШ КОД:>

```
<ВАШ КОД: >
```

2. Запустить полученную программу, получить ожидаемые выходные данные.

Требование к отчету

1. В качестве отчета принимаются два Python файла с кодом (для Tensorflow и Pytorch).
2. Код должен быть в достаточной мере прокомментирован.
3. Отчет предоставляется в виде архива zip, формат имени архива: <номер группы>_<Фамилия_Имя>_LR<номер работы>.zip