Московский государственный технический университет
им. Н.Э. Баумана
Факультет «Информатика и системы управления»
Кафедра «Системы обработки информации и управления»

Лабораторная работа № 6

По курсу «методы машинного обучения в АСОИУ»

«Обучение на основе глубоких Q-сетей»

Выполнил:

студент ИУ5-23М
Семенов И.А.

Проверил:

Гапанюк Ю.Е.

Подпись:

29.04.2024

Москва, 2024

Описание задания

- На основе рассмотренных на лекции примеров реализуйте алгоритм DQN.
- В качестве среды можно использовать классические среды (в этом случае используется полносвязная архитектура нейронной сети).
- В качестве среды можно использовать игры Atari (в этом случае используется сверточная архитектура нейронной сети).
- В случае реализации среды на основе сверточной архитектуры нейронной сети +1 балл за экзамен.

Описание алгоритма

DQN (Deep Q-Network) — это метод обучения с подкреплением, который объединяет идеи Q-обучения и нейронных сетей для улучшения производительности агента в сложных средах с большим количеством состояний. DQN заменяет стандартную Q-матрицу на нейронную сеть, которая приближает Q-функцию и позволяет агенту справляться с более сложными средами.

Основные компоненты DQN:

Нейронная сеть: Нейронная сеть используется для приближения Q-функции, где входом являются состояния среды, а выходом — оценка Q-значений для всех возможных действий.

Целевая сеть: Для повышения стабильности обучения используется целевая нейронная сеть, которая обновляется периодически из основной сети.

Опытный буфер (Replay Buffer): Буфер, в который агент записывает свой опыт (состояние, действие, награду, новое состояние) и из которого случайно извлекает данные для обучения.

Мини-партии: Обучение нейронной сети происходит на случайных мини-партиях из опыта буфера, что помогает уменьшить корреляцию между последовательными данными и повышает стабильность обучения.

Лосс-функция: DQN использует функцию потерь Huber для более стабильного обучения. Она сравнивает предсказанные Q-значения с целевыми значениями, вычисляемыми на основе награды и оценок целевой сети.

Описание алгоритма:

Алгоритм DQN (Deep Q-Network) представляет собой метод обучения с подкреплением, в котором используется глубокая нейронная сеть для обучения агента взаимодействовать с окружающей средой и максимизировать ожидаемую награду. Алгоритм основан на методе Q-обучения, который является одним из популярных методов обучения с

подкреплением. В DQN используется нейронная сеть для аппроксимации функции Q-значений, что позволяет агенту принимать оптимальные решения в сложных средах.

Вот подробное описание алгоритма DQN:

1. Инициализация среды и параметров:
   - Агент инициализирует среду (в данном случае, `CartPole-v1` из библиотеки OpenAI Gym).
   - Устанавливаются параметры алгоритма, такие как `gamma` (коэффициент дисконтирования), `epsilon` (вероятность случайного выбора действия), `epsilon_min` (минимальное значение `epsilon`), `epsilon_decay` (скорость снижения `epsilon`), `learning_rate` (скорость обучения модели), `batch_size` (размер батча для обучения), `memory_size` (размер буфера опыта), `train_start` (минимальное количество опыта для начала обучения) и `update_target_steps` (частота обновления целевой сети).
2. Создание моделей:
   - Создаются две нейронные сети: основная (`main_model`) и целевая (`target_model`).
   - Эти сети используются для предсказания Q-значений действий в различных состояниях.
   - Целевая сеть используется для стабилизации процесса обучения, она периодически обновляется, копируя веса из основной сети.
3. Буфер опыта:
   - Буфер опыта (`ReplayMemory`) используется для хранения опыта агента в виде кортежей (`state`, `action`, `reward`, `next_state`, `done`).
   - Буфер ограничен по размеру (`memory_size`), и старые данные заменяются новыми, если достигается его предельный размер.
4. Главный цикл обучения:
   - Проходит через заданное количество эпизодов.
   - В каждом эпизоде агент начинает в начальном состоянии и взаимодействует со средой, выбирая действия на основе `epsilon-greedy` политики.
      - Если случайное значение (`np.random.rand()`) меньше `epsilon`, агент выбирает случайное действие из доступных в среде.
      - В противном случае, агент выбирает действие с наибольшим предсказанным Q-значением из основной модели (`main_model`).
   - Агент выполняет выбранное действие, получает награду и наблюдает новое состояние.

- Опыт (`state`, `action`, `reward`, `next_state`, `done`) сохраняется в буфере опыта.
- Если количество опыта в буфере больше или равно `train_start`, начинается обучение:
    - Случайно выбирается батч из опыта в буфере.
    - Основная сеть предсказывает Q-значения для текущих состояний и целевых состояний.
    - Обновляются Q-значения, учитывая полученные награды и ожидаемые Q-значения целевой сети.
    - Основная сеть обучается на основе скорректированных Q-значений.
    - Если `step_count` кратен `update_target_steps`, целевая сеть обновляется копированием весов из основной сети.
- `epsilon` уменьшается по мере прогресса обучения до минимального значения (`epsilon_min`).

5. Вывод результатов:
- В конце каждого эпизода выводится его номер и общая награда, полученная агентом за эпизод.
- Если эпизод заканчивается (достигается условие завершения эпизода), обучение продолжается со следующего эпизода.

6. Завершение:
- После завершения всех эпизодов (достижения `max_steps`) обучение завершается, и среда (`env`) закрывается.

Код демонстрирует типичный процесс обучения DQN в простой среде `CartPole-v1`. По мере обучения агент становится лучше в принятии решений, направленных на максимизацию наград в среде.

Текст программы и экранные формы с примерами выполнения программы

✓ 38 сек.

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import random
from collections import deque
import gym

# Параметры
env_name = "CartPole-v1"  # Имя среды OpenAI Gym
gamma = 0.99  # коэффициент дисконтирования
epsilon = 1.0  # начальное значение epsilon для epsilon-greedy политики
epsilon_min = 0.01  # минимальное значение epsilon
epsilon_decay = 0.995  # скорость снижения epsilon
learning_rate = 0.001  # скорость обучения модели
batch_size = 32  # размер батча для обучения
max_steps = 50  # максимальное количество шагов в эпизоде
memory_size = 2000  # размер памяти для буфера опыта
train_start = 1000  # минимальное количество опыта для начала обучения
update_target_steps = 10  # количество шагов между обновлениями целевой сети

# Создаем среду
env = gym.make(env_name)

# Буфер опыта
class ReplayMemory:
    def __init__(self, capacity):
        self.memory = deque(maxlen=capacity)

    def store(self, experience):
        self.memory.append(experience)

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

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

# Создаем нейронную сеть
def create_model(input_shape, output_shape):
    model = keras.Sequential([
        layers.Dense(24, activation='relu', input_shape=input_shape),
        layers.Dense(24, activation='relu'),
        layers.Dense(output_shape)
    ])
```

```
# Инициализируем модели
input_shape = (env.observation_space.shape[0],)
output_shape = env.action_space.n
main_model = create_model(input_shape, output_shape)
target_model = create_model(input_shape, output_shape)
target_model.set_weights(main_model.get_weights())

# Инициализируем память и другие параметры
memory = ReplayMemory(memory_size)
epsilon = epsilon  # Начальное значение epsilon
step_count = 0  # Счетчик шагов

# Главный цикл обучения
for episode in range(1, max_steps + 1):
    state = env.reset()
    state = np.reshape(state, [1, input_shape[0]])
    total_reward = 0

    for step in range(max_steps):
        # Выбор действия с использованием epsilon-greedy политики
        if np.random.rand() <= epsilon:
            action = env.action_space.sample()
        else:
            q_values = main_model.predict(state)
            action = np.argmax(q_values[0])

        # Выполняем действие и получаем награду и новое состояние
        next_state, reward, done, _ = env.step(action)
        next_state = np.reshape(next_state, [1, input_shape[0]])
        total_reward += reward

        # Сохраняем опыт в буфере
        memory.store((state, action, reward, next_state, done))

        # Переходим к следующему состоянию
        state = next_state

        # Обучаем сеть, когда накоплено достаточно опыта
        if len(memory) >= train_start:
            # Выбираем случайный батч из памяти
            batch = memory.sample(batch_size)

            # Подготовка данных для обучения
            states = np.array([experience[0][0] for experience in batch])
            actions = np.array([experience[1] for experience in batch])
            rewards = np.array([experience[2] for experience in batch])
            next_states = np.array([experience[3][0] for experience in batch])
```

✓ 38 сек.

```
        # Обучаем сеть, когда накоплено достаточно опыта
        if len(memory) >= train_start:
            # Выбираем случайный батч из памяти
            batch = memory.sample(batch_size)

            # Подготовка данных для обучения
            states = np.array([experience[0][0] for experience in batch])
            actions = np.array([experience[1] for experience in batch])
            rewards = np.array([experience[2] for experience in batch])
            next_states = np.array([experience[3][0] for experience in batch])
            dones = np.array([experience[4] for experience in batch])

            # Прогнозируем q-значения для текущих состояний и целевых состояний
            q_values = main_model.predict(states)
            q_values_next = target_model.predict(next_states)

            # Обновляем q-значения для текущего батча
            for i in range(batch_size):
                if dones[i]:
                    q_values[i, actions[i]] = rewards[i]
                else:
                    q_values[i, actions[i]] = rewards[i] + gamma * np.max(q_values_next[i])

            # Обучаем сеть
            main_model.train_on_batch(states, q_values)

            # Обновляем целевую сеть периодически
            if step_count % update_target_steps == 0:
                target_model.set_weights(main_model.get_weights())

        # Уменьшаем epsilon
        if epsilon > epsilon_min:
            epsilon *= epsilon_decay
            epsilon = max(epsilon, epsilon_min)

        # Проверяем условие завершения эпизода
        if done:
            break

        # Увеличиваем счетчик шагов
        step_count += 1

    print(f"Episode {episode}: Total reward = {total_reward}")

# Завершаем среду
env.close()
```

```
1/1 [==============================] - 0s 22ms/step
Episode 46: Total reward = 9.0
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 23ms/step
Episode 47: Total reward = 10.0
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
Episode 48: Total reward = 10.0
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 23ms/step
Episode 49: Total reward = 10.0
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
Episode 50: Total reward = 9.0
```

Вывод

На основе полученных результатов можем сделать вывод о том, что алгоритм итерации политики в данном коде реализует метод для обучения агента в среде с подкреплением. Это алгоритм, который использует сочетание оценок текущей политики и ее улучшений для нахождения оптимальной политики, которая максимизирует суммарную награду за время работы агента в среде.