Задача 2. Реализация Deep Q-Learning для классической игры Atari Breakout (1976)

1. Условие задачи

1. **Цель**: Реализовать алгоритм **Deep Q-Learning** (DQN) для игры Atari Breakout (1976). Научить агента играть в игру.
2. **Что нужно сделать**:
    - **Выбрать способ эмуляции Breakout**:
        1. Использовать **Arcade Learning Environment** напрямую, либо
        2. Использовать интеграцию с **OpenAI Gym** (например, среду `BreakoutNoFrameskip-v4`).
    - **Обрабатывать входные данные (кадры)**: перевести их в формат, удобный для нейронной сети (например, обрезать, уменьшить до 84×84, перевести в градиент серого, стековать несколько кадров).
    - **Построить сверточную нейронную сеть (CNN)**, которая будет получать кадры (или стек кадров) на вход и выдавать Q-значения для всех допустимых действий (4 actionns).
    - **Реализовать DQN**: ϵ\epsilonϵ-greedy, experience replay, target network (или базовую версию без target network — по усмотрению).
    - **Запустить процесс обучения** на достаточном количестве epoch(5–10 тысяч эпизодов — в зависимости от ресурса, можно меньше если нет ресурсов).
    - **Проанализировать результаты** (средний счёт, количество выбитых «кирпичей», визуализировать поведение агента).

.

Подсказки к реализации:

1. **Состояние (state)**
    - Оригинальный кадр Breakout: 210×160210 \times 160210×160 RGB-пикселей (или чуть отличающееся разрешение в эмуляторе).
    - Для DQN рекомендуют:
        - Перевести картинку в **градиент серого** (1 канал).
        - Уменьшить размер до **84×84**.
        - Объединять **4 последних кадра** (channel stacking), чтобы агент «видел» динамику (скорость и направление шарика).
    - Итоговое состояние может иметь форму (4,84,84) (4 канала, 84×84).
2. **Действия (action)**
    - Breakout (Atari 2600) обычно имеет **4 ключевых действия**:
        - **NOOP** (ничего не делать),
        - **FIRE** (запустить шар, иногда используется как «стрельба»),
        - **LEFT** (движение платформы влево),
        - **RIGHT** (движение платформы вправо).
    - (В некоторых версиях могут быть дополнительные действия, но чаще оставляют 4.)
3. **Награда (reward)**

- Балл (score) в Breakout даётся за уничтожение «кирпичиков». Каждый сбитый кирпич даёт +1 очко.
- В большинстве эмуляторов этот reward автоматически возвращается средой на каждом шаге.
- Если шарик пропущен, эпизод завершается.

4. **Алгоритм DQN**
   - Обязательно используйте **Experience Replay**.
   - Для стабильности рекомендуют **target network**. Можно ограничиться базовым (классический DQN 2013–2015 гг.).
   - ϵ\epsilonϵ-greedy**: ϵ\epsilonϵ может начинаться с 1.0 и постепенно опускаться до 0.1 или 0.01 (в течение миллионов шагов).
   - **Оптимизационные приёмы**: применение RMSProp или Adam, уменьшение скорости обучения (learning rate decay).
   - **Сверточная архитектура**:
     - Несколько свёрточных слоёв (Conv2d), далее полносвязные (Linear).
     - На выходе — Q-значения размерности [batch_size, num_actions].
5. **Гиперпараметры** :
   - **batch_size** = 32 или 64.
   - **replay buffe**r на 100k–1M переходов.
   - γ≈0.99.
   - LR ≈ 1e-4, 2.5e-4

Адаптивное управление светофором на одном перекрёстке (базовый DQN)

1. Общее описание задачи

Вам нужно разработать **агента**, который управляет светофором на одном перекрёстке так, чтобы **минимизировать** образование пробок и время ожидания автомобилей. Для упрощения предполагается:

1. У нас **один** перекрёсток с **двумя** направлениями движения:
   - Север–Юг (NS)
   - Запад–Восток (WE)
2. Есть **две фазы** светофора:
   - **Фаза 0**: зелёный свет для NS, красный для WE
   - **Фаза 1**: зелёный свет для WE, красный для NS
3. На каждом «шаге времени» (дискретный такт) мы можем выбрать, **какую фазу** включить (0 или 1). При смене фаз можно учитывать штраф (некоторое время «жёлтый» и переход).

Цель — **автоматически** управлять светофором, чтобы **очереди** автомобилей на подъездах (NS, WE) оставались как можно короче.

---

2. Формат среды и награды

1. **Состояние (state)**
   - Длина очереди на направлении NS (целое число).
   - Длина очереди на направлении WE.
   - Текущая фаза светофора (0 или 1).
2. Итого, состояние можно описывать как вектор/массив [QNS,QWE,phase][Q_{\text{NS}}, Q_{\text{WE}}, \text{phase}][QNS,QWE,phase].
3. **Действия (action)**
   - Выбор фазы светофора на следующий шаг: a∈{0,1}
   - При необходимости можно считать, что **«остаться в той же фазе»** и **«переключиться»** — это разные действия, но в данной упрощённой задаче их всего 2.
4. **Награда (reward)**
   - Штраф за длину очередей, например: r=−(QNS+QWE) r = - (Q_{\text{NS}} + Q_{\text{WE}})r=−(QNS+QWE)
   - Чем больше суммарная очередь, тем более **отрицательную** награду получаем (то есть хотим её минимизировать).
   - (Опционально) Можно добавить штраф за переключение фазы, чтобы агент не «дёргал» светофор слишком часто: r−=α×I(переход_фазы) r \mathrel{-}= \alpha \times \mathbb{I}(\text{переход_фазы})r−=α×I(переход_фазы) где α>0\alpha > 0α>0 — штраф.
5. **Динамика**
   - На каждом шаге:
     1. Обновляем фазу светофора, если действие агента это подразумевает.
     2. **Пропускаем** часть машин (например, до CCC автомобилей) в направлении, у которого зелёный. Остальные стоят.

3. **Приходят** новые машины с некоторой вероятностью (по выбранной модели потока).
4. Рассчитываем награду (противоположную сумме очередей).

- Эпизод может длиться до заданного лимита шагов (например, 200), после чего «перезапускаем» систему.

---

3. Задание

1. **Реализуйте свою «среду»** (`TrafficEnv`), которая моделирует один перекрёсток:
   - В методе `reset()` обнуляйте очереди, фазу и счётчик шагов.
   - В методе `step(action)` обновляйте состояние перекрёстка в соответствии с выбранным действием, вычисляйте награду, возвращайте `(next_state, reward, done, info)`.
2. **Создайте агента DQN**:
   - Нейронная сеть (например, 2–3 полносвязных слоя), вход — состояние [QNS,QWE,phase][Q_{\text{NS}}, Q_{\text{WE}}, \text{phase}][QNS,QWE ,phase], выход — Q-значения для двух действий (фаза 0 и фаза 1).
   - ϵ\epsilonϵ-greedy стратегия выбора действий.
   - Experience replay (буфер повторов) для обучения на мини-батчах.
   - (Опционально) Online DQN без replay: если студенты хотят сначала попробовать самый базовый вариант.
3. **Обучите агента**:
   - Запускайте среду на нескольких сотнях эпизодов (например, 200–500 эпизодов, каждый по 200 шагов).
   - Накапливайте статистику: средняя награда, средняя длина очереди, ϵ\epsilonϵ и т.д.
4. **Сравните результат**:
   - С тем, как работает «фиксированный» светофор, переключающийся через равные промежутки времени (например, каждые 10 тактов).
   - Убедитесь, что DQN-управление даёт более высокие награды (меньшие очереди).

Implementation of Deep Q-Learning for the Classic Atari Breakout (1976)

1. Task Description

Goal
Implement a Deep Q-Learning (DQN) algorithm for the Atari Breakout (1976) game and train an agent to play it.

What to Do

1. **Choose how to emulate Breakout**
    - Use the **Arcade Learning Environment (ALE)** directly, or
    - Use **OpenAI Gym** integration (e.g., the environment `BreakoutNoFrameskip-v4`).
2. **Process the input data (frames)**
    - Convert, crop, and downscale frames to 84×84,
    - Convert them to grayscale,
    - Stack several frames (e.g., 4) to capture the ball's motion.
3. **Build a convolutional neural network (CNN)**
    - Input: the (stacked) game frames,
    - Output: Q-values for all valid actions (4 actions).
4. **Implement DQN**
    - ϵ\epsilonϵ-greedy exploration,
    - Experience replay,
    - Target network (or a basic version without it, if desired).
5. **Train the agent** on a sufficient number of episodes
    - For example, 5–10 thousand episodes (depending on resources),
    - Fewer if you have limited computational power.
6. **Analyze the results**
    - Average score,
    - Number of bricks destroyed,
    - Visualize the agent's behavior.

2. Implementation Tips

2.1 State

- The original Breakout frame size is around 210×160 RGB pixels (may vary slightly in the emulator).
- For DQN, it is recommended to:
    1. Convert to **grayscale** (1 channel),
    2. Downscale to **84×84**,
    3. **Stack 4** recent frames so the agent "sees" the ball's speed and direction.
- The final state might have the shape **(4, 84, 84)** (4 channels, 84×84 pixels).

2.2 Actions

- **Breakout (Atari 2600)** usually has 4 key actions:
    1. **NOOP** (do nothing),
    2. **FIRE** (launch the ball),
    3. **LEFT** (move paddle left),
    4. **RIGHT** (move paddle right).
- Some versions have additional actions, but 4 is most common.

2.3 Reward

- You get +1 point for every "brick" destroyed.
- In most emulators, this reward is provided automatically each step.
- Missing the ball ends the episode (agent loses a life).

2.4 DQN Algorithm

- **Must use experience replay.**
- For stability, use a **target network** (though a basic DQN from 2013–2015 can work as a start).
- **ϵ\epsilonϵ-greedy**: typically start ϵ\epsilonϵ at 1.0 and gradually decrease to 0.1 or 0.01 over millions of steps.
- **Optimization**: RMSProp or Adam optimizers, possible learning rate decay.
- **Convolutional architecture**:
    - Multiple Conv2D layers, followed by fully connected layers,
    - Output dimension: [batch_size,num_actions]

2.5 Hyperparameters

- **Batch Size**: 32 or 64.
- **Replay Buffer**: 100k to 1M transitions.
- **γ≈0.99**
- **LR≈10^4 or 2.5×10^4**

Adaptive Traffic Light Control at a Single Intersection (Basic DQN)

1. General Task Description

You need to develop an agent that controls a traffic light at a single intersection to **minimize congestion and vehicle waiting times**. For simplicity, assume:

- There is **one intersection** with **two** directions of traffic:
    - **North–South (NS)**
    - **West–East (WE)**
- There are **two traffic light phases**:
    - **Phase 0**: Green for NS, Red for WE
    - **Phase 1**: Green for WE, Red for NS
- At each discrete time step, you can pick the phase to activate (0 or 1). Optionally, account for a penalty during phase switching (e.g., “yellow” transition).

Goal: Automatically control the light so that the vehicle queues (NS, WE) stay as short as possible.

2. Implementation Tips

2.1 State

- **Queue length** in the NS direction (integer),
- **Queue length** in the WE direction,
- **Current phase** of the traffic light (0 or 1).

Hence, the state can be represented as [QNS,QWE,phase][Q_{\text{NS}}, Q_{\text{WE}}, \text{phase}][QNS,QWE,phase].

2.2 Action

- Select the phase for the next step: $a \in \{0,1\}$

2.3 Reward

- A penalty based on queue lengths, for instance:
  $r = -(QNS + QWE)$
- The larger the total queue, the more negative the reward, so we want to minimize it.
- **(Optional)** Add a penalty for switching phases so the agent does not toggle too often:
  $r -= \alpha \times I(\text{phase\_switch}), \alpha > 0$

2.4 Dynamics

At each time step:

1. Update the traffic light phase if the agent's action requires switching,
2. Let up to CCC cars pass on the green approach,
3. New cars arrive randomly (based on your chosen traffic model),
4. Calculate the reward (negative sum of queue lengths).

An episode continues until a set limit (e.g., 200 steps), then the system resets.