Below is a **high-level summary** of the multi-agent solution for **decentralized drone coverage** with **truly local rewards**. We break down **what** each file does, **how** it does it, and provide some background on the **theory** underpinning this approach.

Solution Overview

We are tackling a **multi-drone coverage** task on an N×NN \times NN×N grid. Each drone has a coverage radius, and we measure how many cells of the grid each drone covers. The **key** aspects here are:

1. **Decentralized Setting:** Each drone is controlled by **its own** DQN agent.
2. **Local Observations:** Each drone observes **only its own** position (row, col) plus a global coverage fraction (optional).
3. **Local Rewards:** Each drone's reward is how many *new* cells it covers (i.e., cells that weren't already covered by other drones).
4. **Learning Algorithm:** Each drone trains a **DQN** policy using a **neural network** for function approximation, storing transitions in a **replay buffer** and performing **Q-learning** style updates.

This setup encourages drones to spread out so that each can earn a local reward from covering unique areas of the grid.

File Summaries

1) multi_drone_env.py

- **Purpose:** Defines the multi-drone coverage environment. In a standard single-agent Gym environment, you have one observation, one action, and one reward. Here, we have a **multi-agent environment** returning:
  - A **dict** of observations: obs[i] for drone iii.
  - A **dict** of rewards: reward[i] for drone iii.
  - A **dict** of actions is expected: action[i] for drone iii.
- **Key Components:**

1. **Initialization** (__init__):
   - Sets up a N×NN \times NN×N grid (grid_size).
   - Creates num_drones drones, each with a coverage radius (coverage_radii).
   - Sets a maximum step limit (max_steps).
   - Defines the **action space** as a dict of size num_drones, each drone having 5 discrete moves.

- Defines the **observation space** as a dict of dimension (row, col, coverage_frac) for each drone.

2. **Reset** (reset()):
   - Randomly positions each drone in the grid.
   - Returns the initial **dictionary** of observations, keyed by drone ID.
3. **Step** (step(action_dict)):
   - Each drone's action is interpreted and executed (moving the drone).
   - Coverage sets are computed for each drone.
   - The environment calculates **local rewards**: each drone's reward is how many new cells it uniquely covers.
   - Checks if the episode is done (full coverage or step limit).
   - Returns {drone_id: observation}, {drone_id: local_reward}, done, and info.
4. **Render** (render()):
   - Prints a simple textual representation of the grid.
   - Marks coverage with "*" and drones with labels like "D0", "D1", etc.

- **Local Reward Computation Theory:**
  - For each drone iii, define:

coverage_i={(r,c)∣dist((r,c),dronei)≤radiusi}. \text{coverage_i} = \{ (r,c) \mid \mathrm{dist}((r,c), \text{drone}_i) \le \mathrm{radius}_i \}.coverage_i={(r,c)∣dist((r,c),dronei)≤radiusi}.

  - Let coverage_minus_i=⋃j≠icoverage_j\text{coverage_minus_i} = \bigcup_{j \neq i} \text{coverage_j}coverage_minus_i=⋃j=icoverage_j.
  - **Newly contributed cells** by drone iii is

new_i=coverage_i ∖ coverage_minus_i. \text{new_i} = \text{coverage_i} \; \setminus \; \text{coverage_minus_i}.new_i=coverage_i∖coverage_minus_i.

  - The reward for drone iii is ∣new_i∣\lvert \text{new_i} \rvert∣new_i∣ (possibly normalized by the total grid size). This ensures each drone only gets credit for coverage that *wouldn't* exist without it.

---

2) agent.py

- **Purpose:** Implements a **DQN agent** that each drone uses to decide actions and learn via replay-buffer Q-learning.
- **Key Components:**
  1. **Initialization** (__init__):

        - Builds a neural network (policy_net) mapping **observations** (3(3(3-dim)→) \to)→ **Q-values** (5(5(5-dim()).
        - Creates an **Adam optimizer**, a **mean squared error** loss, and sets up a **replay buffer**.
    2. **Action Selection** (act(obs)):
        - ϵ\epsilonϵ-greedy: pick a random action with probability ϵ\epsilonϵ, else pick the action that maximizes the Q-network.
    3. **Memory Storage** (step(obs, action, reward, next_obs, done)):
        - Stores transitions in a **replay buffer** for offline training.
    4. **Learning** (learn()):
        - Samples **minibatches** from the replay buffer.
        - **Current Q**: Qθ(obs,action)Q_\theta(\text{obs}, \text{action})Qθ(obs,action).
        - **Target Q**: reward+γmax⁡a'Qθ(next_obs,a')\text{reward} + \gamma \max_{a'} Q_\theta(\text{next_obs}, a')reward+γmaxa'Qθ(next_obs,a') if not done.
        - Minimizes MSE loss between current and target Q via **backprop**.
        - Decays ϵ\epsilonϵ after each training step.

- **DQN Theory:**
    - **Q-Learning** is a temporal-difference method:

Q(s,a)←Q(s,a) + α[r+γmax⁡a'Q(s',a')−Q(s,a)]. Q(s,a) \leftarrow Q(s,a) \;+\; \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right].Q(s,a)←Q(s,a)+α[r+γa'maxQ(s',a')−Q(s,a)].

    - **DQN** (Deep Q-Network) parameterizes QθQ_\thetaQθ with a neural net.
    - **Replay Buffer** ensures i.i.d. training data and stabilizes learning.
    - This agent code is replicated for **each** drone, so each drone has **its own** DQN.

---

3) training.py

- **Purpose:** The **coordinator** script that instantiates the environment and the multiple DQN agents (one per drone). It then runs a training loop over multiple episodes.
- **Key Components:**
    1. **train_multi_drone(...)**
        - Creates the environment (MultiDroneCoverageEnv).
        - Creates one DroneDQNAgent per drone.
        - For each **episode**:

        - **Reset** the env, get initial obs_dict.
        - For each time-step until done:
            1. Ask each agent for an action based on its local observation.
            2. Call env.step(action_dict), receiving (next_obs_dict, reward_dict, done, info).
            3. Each agent stores (obs, action, reward, next_obs, done) in its own replay buffer.
            4. Each agent calls learn() once, sampling from its buffer.
            5. Update obs_dict = next_obs_dict.
        - Track average local reward (or some metric) and print progress.
    - Returns the trained agents and the reward history.

2. **demo_run(...)**
    - Runs a single test episode with the learned policies (still ϵ\epsilonϵ-greedy, but you could set ϵ=0\epsilon=0ϵ=0 for pure exploitation).
    - Prints out the environment's textual rendering each step.

- **Multi-Agent Reinforcement Learning Theory:**
    - We're using a ***decentralized*** approach: each agent has its **own** policy and sees **its own** observation.
    - Agents have separate replay buffers and do not share parameters or experiences. They learn in **parallel** within the same environment.
    - The environment dispatches local rewards, so each agent's objective is to maximize its own discounted return. In this scenario, local reward is newly contributed coverage.
    - In principle, the combination of local rewards can also lead to a good global coverage if the drones learn to coordinate by not overlapping too much (since overlapping coverage yields zero additional local reward).

Technical/ Theoretical Details

1. **State Space & Observations**
    - Each drone's observation is a 3D vector: (row,col,coverage_frac)(row, col, coverage_frac)(row,col,coverage_frac).
    - If we wanted a more **realistic** partial-map input, we could expand the observation with local coverage data or sensor readings.
2. **Action Space**

- Each drone chooses one of 5 moves: UP, DOWN, LEFT, RIGHT, or STAY.
- The environment combines these into a **dict** of actions to step all drones simultaneously.

3. **Local Rewards for Coverage**
   - The environment determines for each drone iii the set of cells it covers (coverage_i), then subtracts the union of coverage from all other drones to find *newly contributed coverage*.
   - This local reward structure is *shaping* each drone to **maximize** its unique coverage area.
4. **DQN / Q-Learning**
   - Each drone's policy πi\pi_iπi is derived from a neural net Qθi(s,a)Q_{\theta_i}(s,a)Qθi(s,a).
   - The agent picks a=arg⁡max⁡a'Qθi(s,a')a = \arg\max_{a'} Q_{\theta_i}(s,a')a=argmaxa'Qθi(s,a') with probability (1−ϵ)(1 - \epsilon)(1−ϵ), or random otherwise.
   - The **TD target** for each transition is ri+γmax⁡a'Qθi(s',a')r_i + \gamma \max_{a'} Q_{\theta_i}(s', a')ri+γmaxa'Qθi(s',a').
   - Over time, ϵ\epsilonϵ decays, moving from exploration to exploitation.
5. **Decentralized vs. Centralized**
   - Here, we call it "decentralized" because each drone sees only its own local state and gets its own local reward.
   - The environment is *technically centralized* in the sense that it calculates coverage sets for all drones. But from the agent's perspective, no drone sees other drones' states or shares parameters.
6. **Scalability**
   - Because we use **function approximation** (PyTorch networks), we can, in principle, handle bigger grids than a tabular Q-table.
   - However, if the grid becomes very large (e.g., 50×50, 100×100, etc.), we may need more advanced neural architectures (e.g., CNNs) or more sophisticated multi-agent RL methods (QMIX, MADDPG, etc.).

In Summary

- **multi_drone_env.py**: Multi-agent environment logic, local coverage sets, local reward calculation.
- **agent.py**: Single-agent DQN class, including neural net, replay buffer, ϵ\epsilonϵ-greedy strategy, and Q-learning updates.

- **training.py**: A coordinator script that (1) instantiates multiple DQN agents (one per drone), (2) runs training episodes (collecting transitions, doing learning steps), and (3) demonstrates the learned policies.

The overall approach is a **decentralized multi-agent DQN** solution that incentivizes drones to discover **non-overlapping coverage** because **each** drone only gains reward from the coverage it uniquely provides.