# Detailed Explanation of `drone_env_limited.py` and `train_agent.py`

(Provided by ChatGPT)

April 7, 2025

## 1 Introduction

In this document, we provide an in-depth explanation of a two-file system that uses Q-learning to cover a 2D grid with multiple drones while avoiding/handling obstacles. The two files are:

1. `drone_env_limited.py`: Defines an environment class `DroneCoverageEnvAdaptiveLimited` that manages drone spawning, movement, toggling on/off, obstacle generation, and a multi-component reward system.

2. `train_agent.py`: Implements tabular Q-learning on the environment, logs progress, and tests a learned policy.

We will walk through how each function works and the rationale behind them.

## 2 `drone_env_limited.py`

Listing 1: drone_env_limited.py

```python
import numpy as np
import random

class DroneCoverageEnvAdaptiveLimited:
    """
    An "adaptive" environment with up to max_drones total.
    - You can spawn new drones (choose radius from a user-defined list) until the limit.
    - You can toggle drones on/off, move them, or remove them.
    - Same reward structure as the original "adaptive" approach:
       reward = coverage_fraction
               - alpha * overlap_fraction
               - beta * uncovered_fraction
               - gamma_penalty * (number_of_active_drones)

    Terminal conditions:
       1) All free cells covered,
       2) Coverage doesn't improve for stall_threshold steps,
       3) max_steps reached.
    """

    def __init__(self, config):
```

```
    """
    We take all environment parameters from the 'config' dictionary:

    For example:
      config = {
        "N": 20,
        "M": 20,
        "available_sizes": [1,2,3,4,5,6,7,8,9,10],
        "max_drones": 10,
        "obstacle_percent": 0.1,
        "alpha_env": 0.5,
        "beta_env": 1.0,
        "gamma_penalty_env": 0.01,
        "stall_threshold_env": 5,
        "max_steps_env": 200
      }

    Then in train_agent.py, we do:
        env = DroneCoverageEnvAdaptiveLimited(config)
    """
    # Read environment parameters from the config dict
    self.N = config["N"]
    self.M = config["M"]
    self.available_sizes = config["available_sizes"]
    self.max_drones = config["max_drones"]

    self.obstacle_percent = config["obstacle_percent"]
    self.alpha = config["alpha_env"]      # overlap penalty weight
    self.beta = config["beta_env"]        # uncovered area penalty
    self.gamma_penalty = config["gamma_penalty_env"]
    self.stall_threshold = config["stall_threshold_env"]
    self.max_steps = config["max_steps_env"]

    # Internal state
    self.done = False
    self.obstacles = set()
    self.num_free_cells = self.N * self.M
    self.drones = [] # list of dicts: {"x": int, "y": int, "radius": int, "active":
        bool}

    self.previous_coverage = 0
    self.stall_counter = 0
    self.steps_taken = 0

def reset(self):
    """
    Resets environment:
      - new obstacles
      - clear drones
      - reset coverage tracking
    Returns an observation describing all drones (positions, radius, active) +
        obstacles.
    """
    self.done = False
```

```python
        self._generate_obstacles()
        self.drones = []
        self.previous_coverage = 0
        self.stall_counter = 0
        self.steps_taken = 0

        return self._get_observation()

    def _generate_obstacles(self):
        total_cells = self.N * self.M
        num_obstacles = int(self.obstacle_percent * total_cells)
        all_cells = [(x, y) for x in range(self.N) for y in range(self.M)]
        obstacle_cells = random.sample(all_cells, num_obstacles)
        self.obstacles = set(obstacle_cells)
        self.num_free_cells = total_cells - num_obstacles

    def _get_observation(self):
        """
        Return a Python object summarizing drones + obstacles.
        In tabular Q-learning, you'd eventually hash or convert this to a string.
        """
        drones_repr = []
        for d in self.drones:
            drones_repr.append((d["x"], d["y"], d["radius"], d["active"]))
        obs = {
            "drones": drones_repr,
            "obstacles": list(self.obstacles),
        }
        return obs

    def step(self, action):
        """
        action: dict with keys
          {
            "type": <"SPAWN", "ACT", "NOOP">,
            "radius": <int if "SPAWN">,
            "drone_index": <int if "ACT">,
            "move": <0..4 or "TOGGLE" or "REMOVE">
          }

        We apply the action (spawn / act / noop), then compute coverage & overlap
        to generate the reward, check terminal conditions, and return:
           next_obs, reward, done, info
        """
        # 1) Process the action
        if action["type"] == "SPAWN":
            self._spawn_drone(action.get("radius", 1))
        elif action["type"] == "ACT":
            idx = action.get("drone_index", -1)
            move = action.get("move", None)
            self._act_on_drone(idx, move)
        elif action["type"] == "NOOP":
            pass
        else:
```

```
            pass # ignore invalid

        # 2) Compute coverage & overlap
        coverage_count, overlap_count = self._compute_coverage_and_overlap()
        coverage_fraction = coverage_count / float(self.num_free_cells) if self.
            num_free_cells > 0 else 1.0
        overlap_fraction = overlap_count / float(self.num_free_cells) if self.
            num_free_cells > 0 else 0.0
        uncovered_fraction = 1.0 - coverage_fraction
        num_active = sum(d["active"] for d in self.drones)

        # 3) Compute reward
        reward = (coverage_fraction
                  - self.alpha*overlap_fraction
                  - self.beta*uncovered_fraction
                  - self.gamma_penalty*num_active)

        # 4) Check terminal conditions
        if coverage_count == self.num_free_cells:
            self.done = True

        if coverage_count > self.previous_coverage:
            self.previous_coverage = coverage_count
            self.stall_counter = 0
        else:
            self.stall_counter += 1
            if self.stall_counter >= self.stall_threshold:
                self.done = True

        self.steps_taken += 1
        if self.steps_taken >= self.max_steps:
            self.done = True

        # 5) Return next_obs, reward, done, {}
        return self._get_observation(), reward, self.done, {}

    def _spawn_drone(self, radius):
        """
        Spawn a new drone, if we haven't reached self.max_drones.
        If 'radius' is not in self.available_sizes, pick a random one.
        """
        if len(self.drones) >= self.max_drones:
            return # can't spawn more

        if radius not in self.available_sizes:
            radius = random.choice(self.available_sizes)

        x = random.randint(0, self.N-1)
        y = random.randint(0, self.M-1)
        d = {"x": x, "y": y, "radius": radius, "active": True}
        self.drones.append(d)

    def _act_on_drone(self, idx, move):
        """
```

```
        If valid idx, apply the specified move:
          - "REMOVE", "TOGGLE", or one of the movement codes (0..4)
        """
        if idx < 0 or idx >= len(self.drones):
            return
        drone = self.drones[idx]

        if move == "REMOVE":
            self.drones.pop(idx)
            return
        if move == "TOGGLE":
            drone["active"] = not drone["active"]
            return

        if not drone["active"]:
            # no movement if inactive
            return

        x, y = drone["x"], drone["y"]
        if move == 0:  # UP
            x = max(0, x-1)
        elif move == 1: # DOWN
            x = min(self.N-1, x+1)
        elif move == 2: # LEFT
            y = max(0, y-1)
        elif move == 3: # RIGHT
            y = min(self.M-1, y+1)
        elif move == 4: # STAY
            pass
        drone["x"], drone["y"] = x, y

    def _compute_coverage_and_overlap(self):
        cover_counter = {}
        for d in self.drones:
            if not d["active"]:
                continue
            x_d, y_d, r = d["x"], d["y"], d["radius"]
            for x_cell in range(self.N):
                for y_cell in range(self.M):
                    if (x_cell, y_cell) in self.obstacles:
                        continue
                    dist = abs(x_cell - x_d) + abs(y_cell - y_d)
                    if dist <= r:
                        cover_counter[(x_cell, y_cell)] = cover_counter.get((x_cell, y_cell
                            ), 0) + 1

        coverage_count = sum(1 for cval in cover_counter.values() if cval >= 1)
        overlap_count = sum(1 for cval in cover_counter.values() if cval >= 2)
        return coverage_count, overlap_count
```

## 2.1 Function-by-Function Explanation

### 2.1.1 `__init__`(config)

- Reads environment parameters from the `config` dictionary (grid size, obstacle fraction, reward coefficients, etc.).
- Initializes internal state, including:
  - `self.drones`: an empty list that will hold drone data as dictionaries.
  - `self.obstacles`: an empty set, later populated in `_generate_obstacles`.
  - `stall_counter`: tracks how many consecutive steps we've gone without improving coverage.
  - `previous_coverage`: stores coverage of the prior step to detect stalls.

### 2.1.2 `reset()`

- Called at the start of each new episode.
- Regenerates obstacles randomly by calling `_generate_obstacles()`, clears any existing drones, and resets counters (stall, coverage, steps).
- Returns an initial `observation` via `_get_observation`.

### 2.1.3 `_generate_obstacles()`

- Randomly selects a certain percentage (given by `obstacle_percent`) of the total grid cells to become obstacles.
- Stores them in `self.obstacles`.
- Adjusts `self.num_free_cells` to reflect non-obstacle cells.

### 2.1.4 `_get_observation()`

- Collects a list of drone data: $(x, y, radius, active)$ for each drone in `self.drones`.
- Returns a dictionary:

```
{
  "drones": [ (x, y, radius, active), ... ],
  "obstacles": [ (ox, oy), ... ]
}
```

- Used by the Q-learning agent for hashing or direct consumption.

### 2.1.5 `step(action)`

- Core environment update method. Expects an `action` dictionary with:

  `"type"`: One of {`"SPAWN"`, `"ACT"`, `"NOOP"`}.

  `"radius"`: An integer radius, if `type` is `"SPAWN"`.

  `"drone_index"`: If `type` = `"ACT"`, which drone we are acting on.

"move": The operation for that drone: "REMOVE", "TOGGLE", or an integer (0..4).

- After processing `action`, calls `_compute_coverage_and_overlap()` to gather coverage stats.
- Computes the step reward:

  $$\text{reward} = \text{coverage\_fraction} - \alpha \times \text{overlap\_fraction} - \beta \times \text{uncovered\_fraction} - \gamma\_\text{penalty} \times \text{num\_active\_drones}.$$

- Checks if we reached terminal conditions (full coverage, stall threshold, max steps).
- Returns $(\text{next\_obs}, \text{reward}, \text{done}, \{\})$.

#### 2.1.6 `_spawn_drone(radius)`

- Creates a new drone with the requested `radius`, if we are below `self.max_drones`.
- Places it randomly in the grid, sets `active = True`.

#### 2.1.7 `_act_on_drone(idx, move)`

- For an existing drone `idx`, apply:
  - `REMOVE`: remove from `self.drones`.
  - `TOGGLE`: flip `drone["active"]`.
  - `0..4`: move drone up, down, left, right, or stay (assuming it is `active`).

#### 2.1.8 `_compute_coverage_and_overlap()`

- Iterates over all `active` drones. For each drone's coverage radius, mark any grid cell within that radius (Manhattan distance) as covered.
- `cover_counter[(x_cell, y_cell)]` increments if multiple drones cover that same cell.
- `coverage_count` = number of cells covered by at least 1 drone.
- `overlap_count` = number of cells covered by 2 or more drones (which is penalized in the reward).

## 3 `train_agent.py`

Listing 2: train_agent.py

```
import pickle
import random
from drone_env_limited import DroneCoverageEnvAdaptiveLimited

################################################################################
# ALL CONFIGURATION IN ONE PLACE
################################################################################
CONFIG = {
    # -----------------------------
    # ENVIRONMENT parameters
```

```
    # ----------------------------
    "N": 10,
    "M": 10,
    "available_sizes": [1,2,3,4,5,6,7,8,9,10],
    "max_drones": 10,
    "obstacle_percent": 0.1,
    "alpha_env": 0.3,
    "beta_env": 0.8,
    "gamma_penalty_env": 0.005,
    "stall_threshold_env": 5,
    "max_steps_env": 200,

    # ----------------------------
    # Q-LEARNING hyperparameters
    # ----------------------------
    "num_episodes": 2000,
    "gamma_rl": 0.9,
    "alpha_rl": 0.05,
    "epsilon_rl": 1.0,
    "epsilon_decay": 0.995,
    "epsilon_min": 0.05,
}

################################################################################
# HELPER FUNCTIONS
################################################################################

def state_to_str(obs):
    """
    Convert the environment's observation (list of drones) into a canonical string.
    We'll sort drones by (radius, x, y, active) so permutations map to the same state.
    """
    drones = obs["drones"]
    canonical = []
    for (x,y,r,a) in drones:
        a_bit = 1 if a else 0
        canonical.append((r,x,y,a_bit))
    canonical.sort()
    return str(canonical)

def possible_actions(env):
    """
    Enumerate all top-level actions:
      1) NOOP
      2) SPAWN for each radius (if below max_drones)
      3) ACT for each drone: 'REMOVE', 'TOGGLE', or move 0..4
    """
    actions = [{"type":"NOOP"}]

    # If we can spawn more drones, add spawn actions
    if len(env.drones) < env.max_drones:
        for r in env.available_sizes:
            actions.append({"type": "SPAWN", "radius": r})
```

```
    # For each existing drone, define possible "ACT" actions
    for i in range(len(env.drones)):
        actions.append({"type":"ACT", "drone_index": i, "move":"REMOVE"})
        actions.append({"type":"ACT", "drone_index": i, "move":"TOGGLE"})
        for move in range(5):
            actions.append({"type":"ACT", "drone_index": i, "move":move})

    return actions

def safe_get(Q_table, s, a):
    """
    Q_table: dict { state_str: { action_str: q_value } }
    Returns Q_table[s][a], creating if needed.
    """
    if s not in Q_table:
        Q_table[s] = {}
    if a not in Q_table[s]:
        Q_table[s][a] = 0.0
    return Q_table[s][a]

###############################################################################
# Q-LEARNING FUNCTION
###############################################################################

def Q_learning_adaptive_limited(config):
    """
    Tabular Q-learning using the DroneCoverageEnvAdaptiveLimited environment.
    We read environment params and RL hyperparams from 'config'.

    In addition to printing each episode's reward, we'll write it to 'training_log.txt'.
    """

    # 1. Create the environment
    env = DroneCoverageEnvAdaptiveLimited(config)

    # 2. Extract RL hyperparams
    num_episodes = config["num_episodes"]
    gamma        = config["gamma_rl"]
    alpha        = config["alpha_rl"]
    epsilon      = config["epsilon_rl"]
    epsilon_decay= config["epsilon_decay"]
    epsilon_min = config["epsilon_min"]

    # 3. Initialize Q-table
    Q_table = {}

    # Open a text file for logging
    with open("training_log.txt", "w") as log_file:

        # 4. Run episodes
        for ep in range(num_episodes):
            obs = env.reset()
            s_str = state_to_str(obs)
            done = False
```

```python
        episode_reward = 0.0

        while not done:
            acts = possible_actions(env)

            # Epsilon-greedy selection
            if random.random() < epsilon:
                act = random.choice(acts)
            else:
                best_q = float("-inf")
                chosen = acts[0]
                for a in acts:
                    q_val = safe_get(Q_table, s_str, str(a))
                    if q_val > best_q:
                        best_q = q_val
                        chosen = a
                act = chosen

            next_obs, reward, done, _ = env.step(act)
            episode_reward += reward
            sp_str = state_to_str(next_obs)

            old_q = safe_get(Q_table, s_str, str(act))
            if sp_str not in Q_table:
                Q_table[sp_str] = {}

            if not done:
                # compute best Q in the next state
                next_acts = possible_actions(env)
                best_next = float("-inf")
                for na in next_acts:
                    val = safe_get(Q_table, sp_str, str(na))
                    if val > best_next:
                        best_next = val
                td_target = reward + gamma * best_next
            else:
                td_target = reward

            new_q = old_q + alpha * (td_target - old_q)
            Q_table[s_str][str(act)] = new_q

            s_str = sp_str

        # Print the episode summary
        print(f"Episode {ep+1}/{num_episodes} - total reward: {episode_reward:.3f}")
        # Write to file
        log_file.write(f"Episode {ep+1}/{num_episodes} - total reward: {episode_reward
            :.3f}\n")

        # decay epsilon
        if epsilon > epsilon_min:
            epsilon *= epsilon_decay

    return Q_table
```

```python
################################################################################
# RUNNING THE LEARNED POLICY
################################################################################

def run_trained_policy(Q_table, config):
    """
    Run the learned policy in a fresh environment using the same environment parameters
    from 'config'. Return the total reward and final observation.
    """
    env = DroneCoverageEnvAdaptiveLimited(config)

    obs = env.reset()
    s_str = state_to_str(obs)
    done = False
    total_r = 0.0
    steps = 0

    max_steps_run = config["max_steps_env"]

    while not done and steps < max_steps_run:
        acts = possible_actions(env)
        best_q = float("-inf")
        chosen = acts[0]
        for a in acts:
            q_val = safe_get(Q_table, s_str, str(a))
            if q_val > best_q:
                best_q = q_val
                chosen = a

        next_obs, reward, done, _ = env.step(chosen)
        total_r += reward
        s_str = state_to_str(next_obs)
        steps += 1

    return total_r, env._get_observation()

################################################################################
# MAIN DEMO FUNCTION
################################################################################

def demo_training():
    Q_table = Q_learning_adaptive_limited(CONFIG)

    # Save Q_table
    with open('Q_table_adaptive_limited.pickle', 'wb') as f:
        pickle.dump(Q_table, f, protocol=pickle.HIGHEST_PROTOCOL)

    # Test run
    total_r, final_obs = run_trained_policy(Q_table, CONFIG)
    print(f"\\nTest run => total reward: {total_r:.3f}. Final drones: {final_obs['drones
        ']}")

if __name__ == "__main__":
```

```
demo_training()
```

## 3.1 Function-by-Function Explanation

### 3.1.1 `CONFIG`

A single dictionary holding environment hyperparameters and Q-learning hyperparameters. For instance:

- `N`, `M`: Grid size.
- `alpha_env`, `beta_env`, `gamma_penalty_env`: Coefficients for reward shaping.
- `num_episodes`, `alpha_rl`, `epsilon_rl`: RL parameters such as the number of episodes, learning rate, and exploration rate.

### 3.1.2 `state_to_str(obs)`

- Takes an `obs` which is a dictionary with `"drones"` and `"obstacles"`.
- Focuses on `obs["drones"]`, which is a list of tuples $(x, y, radius, active)$.
- Sorts them by $(radius, x, y, active)$ to get a canonical order (otherwise, permutations of the same set of drones would produce different strings).
- Returns the string representation to serve as a key in the Q-table dictionary.

### 3.1.3 `possible_actions(env)`

- Enumerates all valid top-level actions.
- Always includes:
  - `{"type":"NOOP"}` for doing nothing.
- If the environment has fewer drones than `max_drones`, we add `SPAWN` actions for each possible radius in `env.available_sizes`.
- For each existing drone index `i`:
  - `REMOVE`: remove the drone
  - `TOGGLE`: toggle `active` on/off (unless code was modified to remove toggling).
  - Movement: `0..4` meaning `UP, DOWN, LEFT, RIGHT, STAY`.
- The final list of dictionaries is returned to the Q-learner, which picks from them.

### 3.1.4 `safe_get(Q_table, s, a)`

- A small helper ensuring `Q_table[s][a]` always exists. Initializes to 0.0 if absent.
- Returns that Q-value.

#### 3.1.5 `Q_learning_adaptive_limited(config)`

Main Q-learning routine:

1. Creates the environment with `DroneCoverageEnvAdaptiveLimited(config)`.
2. Extracts RL hyperparams like `num_episodes`, `gamma_rl`, `alpha_rl`, etc.
3. Initializes an empty dictionary `Q_table`.
4. Loops over episodes:
   - Resets environment, obtains initial `obs`.
   - Converts `obs` to a string `s_str`.
   - While not `done`:
     (a) Compute `acts = possible_actions(env)`.
     (b) Pick an action by epsilon-greedy from `Q_table[s_str]`.
     (c) Step the environment, get `next_obs`, `reward`, `done`.
     (d) Convert next observation to `sp_str`.
     (e) Q-update:
     $$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big].$$
   - Print and log the episode reward.
   - Decay $\epsilon$.
5. Returns the learned `Q_table`.

#### 3.1.6 `run_trained_policy(Q_table, config)`

- Creates a new environment with the same config.
- Resets to get initial state string.
- Takes deterministic actions by picking the action with highest Q-value in the current state.
- Accumulates reward until `done` or `max_steps_run` steps are reached.
- Returns the total reward and the final observation (drone positions).

#### 3.1.7 `demo_training()`

- Calls `Q_learning_adaptive_limited` to train and get a `Q_table`.
- Saves `Q_table` to `Q_table_adaptive_limited.pickle`.
- Tests the policy via `run_trained_policy`.
- Prints final reward and drone configuration.

# 4 Key Observations and Summary

- **Adaptive Drone Approach.** The ability to spawn or remove drones, plus toggling them, gives the policy flexibility to maintain an optimal set of active drones. Drones that add more overlap or cost than benefit can be turned off or removed.
- **Reward Shaping.** The environment returns a composite reward balancing coverage, penalizing overlap, penalizing uncovered areas, and penalizing using many active drones.
- **Terminal Criteria.** Episode ends if:
  1. Full coverage is achieved.
  2. Coverage does not improve for `stall_threshold_env` steps.
  3. `max_steps_env` is reached.
- **Epsilon-Greedy and Q-Table.** The code uses tabular Q-learning with state strings. For bigger grids or more drones, the state space becomes huge, so in practice, one might switch to function approximation or deep RL.

This system demonstrates how to integrate a multi-action adaptive environment with a straightforward RL agent. By carefully constructing the environment, enumerating possible actions, and storing Q-values in a dictionary keyed by a canonical state string, one can systematically learn improved coverage policies over many episodes.

*End of Document*