

E:\ONEU\CS5100\Mohammed\train\_agent.py

```
1 import random
2 import matplotlib.pyplot as plt
3 import numpy as np
4
5 # -----
6 # 1. FIX SEEDS GLOBALLY
7 # -----
8 random.seed(1234)
9 np.random.seed(1234)
10
11 from drone_env_limited import DroneCoverageEnvAdaptiveLimited
12
13 #####
14 # ALL CONFIGURATION IN ONE PLACE
15 #####
16 CONFIG = {
17     "N": 10,
18     "M": 10,
19     "available_sizes": [3,5],
20     "max_drones": 25,
21     "obstacle_percent": 0.0,
22
23     # Increase coverage more strongly
24     "coverage_multiplier": 70.0,
25
26     # Stronger overlap penalty
27     "alpha_env": 25.0,
28     "beta_env": 2.0,
29     # Lower drone penalty so we can place more
30     "gamma_penalty_env": 0.01,
31
32     "stall_threshold_env": 500, # more forgiving
33     "max_steps_env": 1000,
34
35     "num_episodes": 2000,
36     "gamma_rl": 0.9,
37     "alpha_rl": 0.05,
38     "epsilon_rl": 1.0,
39     "epsilon_decay": 0.999,
40     "epsilon_min": 0.01,
41
42     "test_mode": False
43 }
44
45
46 def state_to_str(obs):
47     """
48     Convert environment's observation => canonical string, sorting the drones
```

```

49     by (size, cx, cy, active).
50     """
51     drones = obs["drones"] # each is (cx, cy, size, active)
52     canon = []
53     for (cx, cy, sz, act) in drones:
54         a_bit = 1 if act else 0
55         canon.append((sz, cx, cy, a_bit))
56     canon.sort()
57     return str(canon)
58
59
60 def possible_actions(env, random_spawns=True):
61     """
62     - NOOP
63     - SPAWN_RANDOM for each size in env.available_sizes (if len(env.drones)<env.max_drones)
64       => if random_spawns=True, that means we won't require (cx,cy).
65     - ACT for each drone => 'REMOVE' or 'STAY' only
66       (we removed up/down/left/right toggles).
67     """
68
69     acts = [{"type": "NOOP"}]
70
71     # 1) SPawns: Instead of enumerating x,y, do single "SPAWN_RANDOM" per size
72     if len(env.drones) < env.max_drones:
73         for s in env.available_sizes:
74             acts.append({"type": "SPAWN_RANDOM", "size": s})
75
76     # 2) For existing drones => "REMOVE" or "STAY"
77     for i in range(len(env.drones)):
78         acts.append({"type": "ACT", "drone_index": i, "move": "REMOVE"})
79         # "STAY" = do nothing but keep active
80         acts.append({"type": "ACT", "drone_index": i, "move": "STAY"})
81
82     return acts
83
84
85 def safe_q(Q_table, s, a):
86     if s not in Q_table:
87         Q_table[s] = {}
88     if a not in Q_table[s]:
89         Q_table[s][a] = 0.0
90     return Q_table[s][a]
91
92
93 #####
94 # Q-LEARNING
95 #####
96 def Q_learning_adaptive_limited(config):
97     """
98     Train a tabular Q-table with epsilon-greedy exploration.

```

```

99     Keep track of coverage fraction as well as reward.
100     """
101     env = DroneCoverageEnvAdaptiveLimited(config)
102     Q_table = {}
103
104     best_Q_table = {}
105     best_coverage_fraction = -1.0
106
107     num_episodes = config["num_episodes"]
108     gamma = config["gamma_r1"]
109     alpha = config["alpha_r1"]
110     epsilon = config["epsilon_r1"]
111     eps_decay = config["epsilon_decay"]
112     eps_min = config["epsilon_min"]
113
114     ep_rewards = []
115     ep_coverages = [] # store coverage fraction each episode
116
117     # (A) Count how often the agent visits the "empty" state => "[]"
118     empty_visits = 0
119
120     with open("training_output.txt", "w") as log_file:
121
122         for ep in range(num_episodes):
123             obs = env.reset()
124             s_str = state_to_str(obs)
125             done = False
126             ep_reward = 0.0
127             steps = 0
128
129             if s_str == "[]":
130                 empty_visits += 1
131
132             while not done:
133                 acts = possible_actions(env, random_spawns=True)
134
135                 # Epsilon-greedy for training
136                 if random.random() < epsilon:
137                     act = random.choice(acts)
138                 else:
139                     best_val = float("-inf")
140                     chosen = None
141                     for a in acts:
142                         val = safe_q(Q_table, s_str, str(a))
143                         if val > best_val:
144                             best_val = val
145                             chosen = a
146                     act = chosen
147
148                 next_obs, reward, done, info = env.step(act)

```

```

149         ep_reward += reward
150
151         sp_str = state_to_str(next_obs)
152         old_q = safe_q(Q_table, s_str, str(act))
153
154         if sp_str not in Q_table:
155             Q_table[sp_str] = {}
156
157         # Q-learning update
158         if not done:
159             nxt_acts = possible_actions(env, random_spawns=True)
160             best_next = float("-inf")
161             for na in nxt_acts:
162                 v = safe_q(Q_table, sp_str, str(na))
163                 if v > best_next:
164                     best_next = v
165             td_target = reward + gamma * best_next
166         else:
167             td_target = reward
168
169         new_q = old_q + alpha*(td_target - old_q)
170         Q_table[s_str][str(act)] = new_q
171
172         s_str = sp_str
173         steps += 1
174
175         if s_str == "[]":
176             empty_visits += 1
177
178     # end of episode
179     if epsilon > eps_min:
180         epsilon *= eps_decay
181
182     ep_rewards.append(ep_reward)
183
184     coverage_fraction_episode = 0.0
185     if env.num_free_cells > 0:
186         coverage_fraction_episode = (
187             env.previous_coverage / float(env.num_free_cells)
188         )
189     ep_coverages.append(coverage_fraction_episode)
190
191     # If better coverage => copy entire Q-table
192     if coverage_fraction_episode > best_coverage_fraction:
193         best_coverage_fraction = coverage_fraction_episode
194         best_Q_table = {}
195         for st in Q_table:
196             best_Q_table[st] = {}
197             for ac in Q_table[st]:
198                 best_Q_table[st][ac] = Q_table[st][ac]

```

```

199         print(f" --> Found new best coverage: {100.0*coverage_fraction_episode:.1f}%")
200
201         line_str = (f"Episode {ep+1}/{num_episodes} => "
202                     f"steps={steps}, reward={ep_reward:.3f}, "
203                     f"coverage={env.previous_coverage}/{env.num_free_cells} "
204                     f"({coverage_fraction_episode*100:.1f}%")
205         print(line_str)
206         log_file.write(line_str + "\n")
207
208     # Plot training curve
209     plt.figure(figsize=(10,5))
210
211     plt.subplot(1,2,1)
212     plt.plot(ep_rewards, label="Episode Reward")
213     plt.xlabel("Episode")
214     plt.ylabel("Reward")
215     plt.title("Training Rewards Over Episodes")
216     plt.legend()
217
218     plt.subplot(1,2,2)
219     plt.plot(ep_coverages, label="Coverage Fraction")
220     plt.xlabel("Episode")
221     plt.ylabel("Coverage Fraction")
222     plt.title("Coverage Fraction Over Episodes")
223     plt.legend()
224
225     plt.tight_layout()
226     plt.savefig("training_progress.png", dpi=100)
227     plt.close()
228
229     print(f"\n[INFO] Visited the empty state '[' {empty_visits} times in training!")
230
231     # Return the best Q-table found
232     return best_Q_table
233
234
235 def evaluate_policy(Q_table, config):
236     """
237     Evaluate purely greedily from Q_table until done or max_steps.
238     We'll do a short forced spawn loop, and also do some mild epsilon exploration
239     so that we actually try spawns if the Q-values are not well formed.
240     """
241
242     random.seed(1234)
243     np.random.seed(1234)
244
245     config = dict(config)
246     config["test_mode"] = True
247     config["max_steps_env"] = 500

```

```

248
249 env = DroneCoverageEnvAdaptivelimited(config)
250 obs = env.reset()
251 s_str = state_to_str(obs)
252
253 done = False
254 total_r = 0.0
255 steps = 0
256 max_steps = config["max_steps_env"]
257
258 # We'll do a short forced spawn loop: 5 spawns
259 # but we'll pick the best action among spawn actions
260 # if it doesn't exist, we do a random spawn.
261 # or we do an epsilon approach.
262
263 # Let's do a small epsilon approach in final test:
264 # (Even though "purely greedy" was the original, we want guaranteed coverage.)
265 test_epsilon = 0.3
266
267 for i in range(5):
268     if done:
269         break
270     acts = possible_actions(env, random_spawns=True) # random spawn available
271     if random.random() < test_epsilon:
272         act = random.choice(acts)
273     else:
274         best_val = float("-inf")
275         chosen = acts[0]
276         for a in acts:
277             val = safe_q(Q_table, s_str, str(a))
278             if val > best_val:
279                 best_val = val
280                 chosen = a
281         act = chosen
282
283     next_obs, r, done, _ = env.step(act)
284     total_r += r
285     s_str = state_to_str(next_obs)
286     steps += 1
287
288 # Now do the normal Q-based loop
289 while not done and steps < max_steps:
290     acts = possible_actions(env, random_spawns=True)
291
292     if random.random() < test_epsilon:
293         chosen = random.choice(acts)
294     else:
295         best_val = float("-inf")
296         chosen = acts[0]
297         for a in acts:

```

```
298         val = safe_q(Q_table, s_str, str(a))
299         if val > best_val:
300             best_val = val
301             chosen = a
302
303         next_obs, r, done, _ = env.step(chosen)
304         total_r += r
305         s_str = state_to_str(next_obs)
306         steps += 1
307
308     return total_r, env._get_observation()
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
```

