环境

python 3.9

windows 10

架构

- `BaselineAstar.py` 是基准算法
- `DNQ.py` 是无障碍物路径规划强化学习算法
- `DNQObstacle.py` 加入了障碍物
- `DoubleDNQ.py` 在有障碍物的基础上对DNQ进行了改进，引入Double DNQ减少计算目标Q值的过估计，引入Dueling DQN改进网络结构。
- 上述文件可分别直接运行。

项目目标

1. **多智能体路径规划与拥堵控制**
   设计一个网格化交通路由环境（ `TrafficRoutingEnv` ），在其中放置若干智能体（agent）和各自的目标位置。智能体需要学会在遵守网格边界、避免与其他智能体同格或交叉移动（碰撞）的前提下，尽快从自己的起点到达目的地。
2. **基于深度强化学习的决策**
   使用 DQN（Deep Q-Network）算法，让每个智能体独立地通过与环境不断交互，学习一套策略。每个智能体拥有自己的一对网络（策略网和目标网）以及独立的经验回放缓冲，在观测到“当前位置 + 目标位置 + 周围邻居占据信息 + 归一化曼哈顿距离”等状态后，输出 5 种可能动作（上、下、左、右、停留）中最优的一项。
3. **奖励塑形（Potential Shaping）以加速收敛**
   在原始“到达目标得 +10，未到得 −1”的基础上，引入“曼哈顿距离潜力塑形”项。具体地，每一步：
   - 先计算当前位置到目的地的旧距离（dist_old）和新距离（dist_new），
   - 令 $\phi = -$ 曼哈顿距离，在奖励里加上“$\gamma\phi_{\text{new}} - \phi_{\text{old}}$”这一项，促使智能体向距离更近的方向移动，从而加快训练。
4. **可视化训练后策略**
   在训练结束后，用 ε = 0（完全贪心）策略，让所有智能体再跑一次，收集它们的移动轨迹并在一个网格图里绘制出来。这样可以直观地看到：各个智能体在学会避免碰撞后，是如何规划路径、到达各自目标的。

整体架构

整个项目可以分为三大模块：**环境模块**、**训练模块**（MADQNTrainer）、**可视化模块**。

环境模块： `TrafficRoutingEnv` 类

- **核心职责**
  - 定义一个 H×W 的离散网格环境，记录 N 个智能体的位置与各自的目标。
  - 处理每一步的动作：根据传入的 `actions` （字典形式， `{agent_id: action}` ），先计算所有智能体“期望新位置”，再做碰撞检测（包括“同格碰撞”和“交换格子碰撞”），让冲突方退回原位，其余智能体移动到新位置。
  - 给所有智能体计算“原始奖励”：当某个智能体第一次步入目标格时 +10，之后停留在目标格给 0；尚未到达时每步 -1。
  - 将“奖励塑形”逻辑加到原始奖励中：

$$\text{shaped\_reward} \;=\; \text{raw\_reward} \;+\; \big(\gamma \cdot \phi_{\text{new}} \;-\; \phi_{\text{old}}\big), \quad \phi = -\,(\text{曼哈顿距离})$$

  - 对“已到达”状态做冻结：一旦 `agent_i` 到达目标，后续它的动作始终被当成“停留”，不会再移动也不会拿到负奖励。
- **主要方法**
  i. `__init__(self, grid_size, num_agents)`
    - 接受网格大小和智能体数量，在构造里随机生成（并且只生成一次）每个智能体的起点（ `self._initial_positions` ）和各自的目标（ `self.destinations` ）。
    - 同时初始化内部状态（ `self.agent_positions = self._initial_positions` 、“已到达”字典、步数计数）。
  ii. `reset(self)`
    - 每当一个 episode 开始时，直接把 `self.agent_positions` 还原到构造时那套初始点、把 “已到达” 全部设为 False、把 `self.steps=0` ，并返回一个观测字典 `{i: {'position':…, 'destination':…}}` ，用于智能体做初始决策。
    - ◇ **关键点：这里不再在每次 reset 时随机坐标，而是把位置复位到“固定的初始点”。**
  iii. `step(self, actions)`
    - 输入：一个 `{agent_id: action}` ，动作编号 0=上、1=下、2=左、3=右、4=停留。
    - 步骤：
      a. 记录所有智能体的旧位置。
      b. 根据每个智能体的 `action` 算出 “期望新位置”，如果该智能体已经到达则保持不动。
      c. 检测“交换碰撞”：若 A 想去 B 的旧位置，而 B 想去 A 的旧位置，则两者都退回原位。
      d. 对剩余的“期望新位置”做“同格碰撞”检测：若有多人想去同一个格子，则这些人都退回原位。
      e. 得到最终位置后，更新 `self.agent_positions` ，并把步数 `self.steps` 加 1。
      f. 计算每个智能体的 “原始 reward” 与 “done”（是否到达），并做潜力塑形得到 `shaped_rewards[i]` 。
      g. 如果所有智能体都已到达或步数到 50，就把 `done=True` ，否则 `done=False` 。
      h. 返回 `(next_obs, shaped_rewards, done, info={})` 。

训练模块： `MADQNTrainer` 类

- **核心职责**
  - 管理 N 个智能体各自的网络、目标网络、优化器、回放缓冲区，执行并行的 DQN 训练。
  - 采用 ε-贪心策略选动作，存储 $(s, a, r, s', \text{done\_before})$ 五元组到各自的回放缓冲中，定期从缓冲随机抽样进行 Q-learning 更新。
  - 每经过若干回合，把策略网络的参数同步到目标网络里。
- **主要组件**
  i. `__init__(self, env, num_agents, state_dim, action_dim, buffer_capacity, batch_size, gamma, lr, target_update)`
    - `env`：上述 `TrafficRoutingEnv` 实例。
    - `num_agents`：智能体数量（与环境保持一致）。

- `state_dim`：输入给 DQN 的状态向量维度（本项目是 14：4 维坐标信息 + 9 维邻居占据矩阵 + 1 维归一化曼哈顿距离）。
- `action_dim`：动作个数（本项目固定为 5）。
- `gamma`：折扣因子。
- `target_update`：每隔多少回合把策略网权重复制到目标网。
- 内部为每个 `i` 分别创建：
  - `policy_nets[i] = DQN(state_dim, action_dim)`
  - `target_nets[i] = DQN(state_dim, action_dim)` 并且 `target_nets[i].load_state_dict(policy_nets[i].state_dict())`
  - `optimizers[i] = Adam(policy_nets[i].parameters(), lr)`
  - `replay_buffers[i] = ReplayBuffer(buffer_capacity)`

ii. `select_action(self, agent_id, state, eps, done)`
- 若 `done=True`（智能体已到达目标），直接返回停留动作 4。
- 否则以 ε 概率随机返回 `[0, action_dim)` 中的一个数；以 (1−ε) 概率通过 `policy_nets[agent_id](state)` 输出 Q 值，选最大值索引动作。

iii. `optimize_agent(self, agent_id)`
- 从 `replay_buffers[agent_id]` 中随机抽 `batch_size` 条 $(s, a, r, s', \text{done})$。
- 转成张量后：
  - 当前网络输出 `current_q = policy_net(s).gather(a)`；
  - 目标网络输出 `next_q = target_net(s').max(a')`，用公式 `target_q = r + γ * next_q * (1 - done)`。
  - 计算 MSE 损失，反向传播更新 `policy_nets[agent_id]` 的参数。

iv. `update_targets(self)`
- 每当回合数整除 `target_update` 时，就把 `policy_nets[i].state_dict()` 复制给 `target_nets[i]`。

v. `train(self, num_episodes, max_steps, eps_start, eps_end, eps_decay)`
- 主循环：

```
eps = eps_start
episode_returns = []
for episode in 1..num_episodes:
    obs = env.reset()
    state_dict = obs_to_state(obs, env.grid_size)
    done_dict = {i: False for i in agents}
    total_reward = 0

    for step in 0..max_steps-1:
        # 1) ε-贪心选出所有智能体的 actions
        for i in range(num_agents):
            actions[i] = select_action(i, state_dict[i], eps, done_dict[i])

        # 2) 与环境交互
        next_obs, rewards, done_all, _ = env.step(actions)
        next_state_dict = obs_to_state(next_obs, env.grid_size)

        # 3) 把 transition 存入各自缓冲，并更新 done_dict
        for i in range(num_agents):
            replay_buffers[i].push(
                state_dict[i],
                actions[i],
                rewards[i],
                next_state_dict[i],
                done_dict[i]
            )
            if rewards[i] == 10:
                done_dict[i] = True
            total_reward += rewards[i]

        state_dict = next_state_dict

        # 4) 分别对每个 agent 调用 optimize_agent(i)
        for i in range(num_agents):
            optimize_agent(i)

        # 5) 如果 done_all=True，就结束本回合
        if done_all:
            break

    episode_returns.append(total_reward)

    # 6) ε 衰减
    eps = max(eps * eps_decay, eps_end)

    # 7) 每 target_update 个 episode，同步目标网络
    if episode % target_update == 0:
        update_targets()

    # 8) 每 10 个 episode 打印一次最近 10 回合的平均回报
    if episode % 10 == 0:
```

```
                print(episode, eps, avg(episode_returns[-10:]))
        # 9) 训练结束后绘制 episode_returns 曲线
```

- **注意点**
  - `done_dict[i]`：标记第 i 个智能体是否已经到达过终点，用于让它后续始终“停留”而不再拿到负奖励。
  - `env.step(actions)` 中返回的 `rewards[i]` 已包含潜力塑形项，所以 `total_reward` 累计的是“塑形后”总奖励。
  - `episode_returns` 存储每个回合（所有智能体一次完整交互）的累积奖励，便于画出训练曲线。

可视化模块：`animate(trainer, env)`

- **核心职责**
  - 在训练结束后，用 “ε = 0（纯贪心）” 策略演示一次完整回合，收集每个智能体的位置轨迹，并绘制实时动画到同一个网格图上。
  - 目的是直观地展示：训练好的策略在给定初始状态下，三个以上智能体如何规划不碰撞的路径，并到达各自的目标。
- **主要流程**

```
H, W = env.grid_size
agent_trajectories = {i: [] for i in range(env.num_agents)}
done_dict = {i: False for i in range(env.num_agents)}

# 重置环境并记录初始位置
obs = env.reset()
state_dict = obs_to_state(obs, env.grid_size)
for i in range(env.num_agents):
    agent_trajectories[i].append(env.agent_positions[i])

# 收集所有步骤的数据用于动画
all_steps = []
step = 0
done = False

while not done and step < 50:
    actions = {
        i: trainer.select_action(i, state_dict[i], eps=0.0, done=done_dict[i])
        for i in range(env.num_agents)
    }
    next_obs, rewards, done, _, num_collisions = env.step(actions)
    next_state_dict = obs_to_state(next_obs, env.grid_size)

    for i in range(env.num_agents):
        if rewards[i] == 10 or env.agent_positions[i] == env.destinations[i]:
            done_dict[i] = True
        agent_trajectories[i].append(env.agent_positions[i])

    # 记录当前步骤的所有智能体位置
    all_steps.append({i: env.agent_positions[i] for i in range(env.num_agents)})
    state_dict = next_state_dict
    step += 1

print("Final Greedy Trajectories (row, col) for each agent:")
for i, traj in agent_trajectories.items():
    print(f"Agent {i}: {traj}")

# 创建图形
fig, ax = plt.subplots(figsize=(8, 8))

# 1. 绘制网格线
for x in range(H + 1):
    ax.plot([0, W], [x, x], color='gray', linewidth=0.5)
for y in range(W + 1):
    ax.plot([y, y], [0, H], color='gray', linewidth=0.5)

# 2. 绘制障碍物（填满整个格子）
for (ox, oy) in env.obstacles:
    rect = Rectangle((oy, H - 1 - ox), 1, 1, facecolor='black', edgecolor='black')
    ax.add_patch(rect)

# 3. 绘制每个智能体的起点和终点
```

```
base_colors = ['r', 'g', 'b', 'c', 'm', 'y', 'k', '#FFA500']
colors_list = [base_colors[i % len(base_colors)] for i in range(env.num_agents)]

# 绘制目的地（终点）
destinations = {}
for i in range(env.num_agents):
    goal = env.destinations[i]
    gx, gy = goal[1] + 0.5, (H - 1 - goal[0]) + 0.5
    destinations[i] = ax.scatter([gx], [gy],
                                 color=colors_list[i],
                                 marker='*',
                                 s=120,
                                 zorder=3)

# 初始化智能体位置标记
agents = {}
for i in range(env.num_agents):
    start = agent_trajectories[i][0]
    sx, sy = start[1] + 0.5, (H - 1 - start[0]) + 0.5
    agents[i] = ax.scatter([sx], [sy],
                           color=colors_list[i],
                           marker='s',
                           s=80,
                           zorder=3)

# 初始化轨迹线
lines = {}
for i in range(env.num_agents):
    lines[i], = ax.plot([], [],
                        color=colors_list[i],
                        marker='o',
                        markersize=6,
                        linewidth=2,
                        zorder=2)

# 设置坐标轴
ax.set_xlim(0, W)
ax.set_ylim(0, H)
ax.set_aspect('equal')
ax.set_xticks([])
ax.set_yticks([])
ax.set_title("Multi-Agent Trajectories Animation (Greedy Policy)")

# 添加步骤计数器文本
step_text = ax.text(W/2, H+0.5, "Step: 0", ha='center', va='center', fontsize=12)

# 自定义图例
legend_elements = [
    Rectangle((0, 0), 1, 1, facecolor='black', edgecolor='black', label='Obstacle'),
    Line2D([0], [0], marker='s', color='gray', label='Start',
           markerfacecolor='none', linestyle='None', markersize=10),
    Line2D([0], [0], marker='*', color='gray', label='Goal',
```

```
            markerfacecolor='none', linestyle='None', markersize=12)
    ]
    for i, color in enumerate(colors_list):
        legend_elements.append(Line2D([0], [0], color=color, lw=2, label=f'Agent {i}'))

    ax.legend(handles=legend_elements,
              bbox_to_anchor=(1.01, 1),
              loc='upper left',
              fontsize=9)

    # 动画更新函数
    def update(frame):
        nonlocal step_text

        # 更新步骤文本
        step_text.set_text(f"Step: {frame+1}")

        # 更新每个智能体的位置和轨迹
        for i in range(env.num_agents):
            # 只更新到当前帧的位置
            current_traj = agent_trajectories[i][:frame+2]  # +2因为初始位置在0帧

            # 更新智能体位置
            if frame < len(all_steps):
                x, y = all_steps[frame][i]
                agents[i].set_offsets([y + 0.5, (H - 1 - x) + 0.5])

            # 更新轨迹线
            xs = [pos[1] + 0.5 for pos in current_traj]
            ys = [(H - 1 - pos[0]) + 0.5 for pos in current_traj]
            lines[i].set_data(xs, ys)

        return list(agents.values()) + list(lines.values()) + [step_text]

    # 创建动画
    ani = FuncAnimation(
        fig,
        update,
        frames=len(all_steps),  # 使用实际步骤数作为帧数
        interval=500,  # 每500毫秒更新一帧
        blit=True,
        repeat=False
    )

    plt.tight_layout()
    plt.show()

    # 保存为GIF
    ani.save('animate_results/greedy_trajectories_DQNObstacle.gif', writer='pillow', fps=2, dpi=100)
    return ani
```

- **要点**
  - `eps=0.0`：整个演示过程切换到“纯贪心”策略，不做随机动作。

- `agent_trajectories[i]`：一个列表，记录第 i 个智能体从初始到终止时刻，每一步的格子坐标。
- 网格线保证画面可视化成一个直观的方格；每条曲线用不同颜色，起点用方块标记、终点用星形标记。

整体流程串联

1. **创建环境和 Trainer**

```
env = TrafficRoutingEnv(grid_size=(8,8), num_agents=6)
state_dim = 14    # 4坐标 + 9 邻居 + 1 归一化距离
action_dim = 5    # 上、下、左、右、停留
trainer = MADQNTrainer(env, num_agents=6,
                       state_dim=state_dim,
                       action_dim=action_dim)
```

2. **训练阶段**

```
episode_returns = trainer.train(num_episodes=350)
```

- 每个 episode 调用 `env.reset()`（会把 agent 恢复到构造时固定的起点/终点），
- 在一系列交互步骤里（每步 `env.step(actions)` & 存储经验 & 优化网络），直到“所有 agent 都到达” 或 “步数到 50” 为止，将这段累积奖励记入 `episode_returns`。
- 同时 ε 衰减、定期同步目标网。

3. **可视化阶段**

```
animate(trainer, env)
```

- `env.reset()`（把 agent 再次恢复到最初起点）
- 用“纯贪心”动作（ε=0）让模型演示一次完整回合，收集位置并绘图。
- 最后在看到 N 条彩色折线代表智能体运动的动画，直观展示训练后智能体的行走轨迹和避让策略。

总结

- **项目目标**：通过多智能体 DQN，在离散网格里让多个智能体学会不碰撞地从固定起点到达指定终点；同时引入曼哈顿距离塑形，加速训练收敛。
- **环境架构**：
  i. `__init__` 随机且仅随机一次起点与终点，之后每次 `reset()` 都把位置复位到那套初始点。
  ii. `step()` 处理“同格碰撞”“交换碰撞”，更新位置、计算原始奖励＋塑形奖励、返回 `shaped_rewards`。
- **训练架构**：
  i. 为每个 agent 独立维护 DQN 策略网 + 目标网 + 回放缓冲区 + 优化器。
  ii. 主循环里每个 episode 调 `env.reset()`、逐步与环境交互并存储经验、批量更新网络、定期同步目标网、记录回报。
- **可视化架构**：
  i. 训练结束后，用 ε=0（纯贪心）策略演示一次完整回合。

ii. 收集并打印每个 agent 的位置序列，然后在网格上绘制它们的移动动画、起点和终点标记。

通过上述模块化设计，实现了一个相对完整的“多智能体网格路径规划 + DQN 训练 + 运动可视化”流水线。