Project Repository

Simulator for Reinforcement Learning-based Fleet Management

GitHub Repository

`https://github.com/illidanlab/Simulator`

A large-scale ride-hailing simulator supporting IDQN, contextual DQN, contextual A2C, and LP-cA2C algorithms.

algorithm/alg_utility.py I

Purpose: Provides **helper functions** and **math utilities** that all RL algorithms (IDQN, cDQN, cA2C, LP-cA2C) use inside the simulator.

Key Components

- **Probability Distribution Classes (Pd, DiagGaussianPd)**
  Used to sample actions from a **Gaussian policy** (for Actor-Critic). Computes log-probabilities, entropy, sampling.
- **Reward Normalization**
  Makes rewards stable by converting them to **zero-mean & unit-variance**.
- **Continuous Quadratic Knapsack / Projection Functions**
  Used to **optimize driver allocation** while keeping constraints (e.g., limited idle drivers). Important for **LP-cA2C** to adjust actions using linear programming ideas.

algorithm/alg_utility.py II

- **Fully-Connected Layer Function (fc)**
  Simple wrapper to build neural network layers.
- **Sampling Utilities (categorical_sample_split)**
  Helps sample discrete actions for each agent.
- **State/Reward Conversion Helpers**
  Maps grid states $\rightarrow$ node IDs that the RL model understands.
- **Collision Detection**
  Counts action conflicts (two agents moving into each other's grid).
- **Q-table Summation**
  Utility for debugging: checks overall Q-value magnitude.

alg_utility.py provides math tools, distribution functions, sampling, projections, and helper utilities that the RL algorithms need to make decisions, update networks, and enforce constraints.

algorithm/IDQN.py I

Purpose: Implements **Independent Deep Q-Learning** for ride-sharing grids — each grid learns its own Q-value without coordination.

Key Ideas (Very Simple):

- **Estimator (DQN Network)**
  3-layer neural network → outputs **Q(s, a)** for each possible move. Learns using **Bellman update:** $Q \leftarrow r + \gamma \max(Q_{\text{next}})$.
- **Action Selection**
  Uses **valid neighbor mask** (can only move to connected grids). ϵ-greedy: mostly best action, sometimes explore. Converts Q-values → number of drivers moving to neighbors.
- **State Processor**
  Converts full city state → **per-grid state vectors**. Adds time one-hot, driver counts, order counts. Computes rewards & next states for training.

algorithm/IDQN.py II

- **Replay Memory**
  Stores experience (s, a, r, s'). Samples random mini-batches for stable training.
- **ModelParameterCopier**
  Copies online network $\rightarrow$ target network for stable Q-learning.

IDQN.py trains a separate DQN for each grid to decide where its drivers should go next, without any coordination between grids.

algorithm/cDQN.py I

Purpose: Implements **Contextual Deep Q-Network (cDQN)** — *one shared value network* for **all grids**, coordinating driver movement based on context.

Key Ideas (Very Simple):

- **One Shared Value Network**
  Instead of 504 separate DQNs (IDQN), cDQN uses **one model** to value all grid states. This makes learning **faster and more coordinated**.
- **Context = Number of Idle Drivers**
  If a grid has more idle drivers, it can take more actions. Action distribution based on value of neighbor grids + context.
- **Expected SARSA Update**
  Computes targets using **probability-weighted next Q-values**, not max Q.

algorithm/cDQN.py II

- **Neighbor Masking**
  Each grid can only move drivers to **its connected neighbors**.
- **Action Generation**
  For each grid: look at Q-values of neighbor grids $\rightarrow$ pick best neighbor (ϵ-greedy) $\rightarrow$ split idle drivers among neighbors.
- **Replay Memory + Target Network**
  Stores past experiences and stabilizes training using parameter copying.

cDQN.py learns a single global value function that directs driver movements across all grids, using context and neighbor structure to coordinate actions better than IDQN.

cA2C.py I

Purpose: Smartly dispatch drivers across city grids using **A2C (Actor–Critic)** with a **central coordinator**.

Key Idea:

- One centralized Value Network (Critic) – learns how "good" each grid state is.
- One centralized Policy Network (Actor) – learns where to send drivers next.
- Uses **Advantage = (reward + future value − current value)** to update policy.

How it Works:

1. **State Processor** – Converts city-wide state → per-grid features (drivers, orders, time, grid ID).

cA2C.py II

2. **Value Network (Critic)** – 3-layer MLP $\rightarrow$ outputs $V(s)$ for each grid. Trained with TD target: $target = reward + \gamma V(next\_state)$.
3. **Policy Network (Actor)** – 3-layer MLP $\rightarrow$ outputs action probabilities. Invalid actions removed using neighbor masks. Trains using $policy\_loss = -\log \pi(a|s) * advantage$.
4. **Central Advantage Calculation** – Critic computes future value $\rightarrow$ advantage for each move.
5. **Replay Buffers** – Stores experiences for policy and value updates.

Actor and Critic look at **all grids together**, not independently (unlike IDQN). Ensures **better global coordination** across the city.

run/run_baseline_nopolicy.py I

Purpose: Runs the simulator **without any RL policy**. Drivers do **not move**. Used as a **baseline** to compare with IDQN, cDQN, cA2C.

What It Does:

- Loads real data (orders, drivers, grid map).
- Creates the CityReal simulator (full-day, 144 steps).
- Runs 10 episodes with **empty actions** $\rightarrow$ no rebalancing.
- Environment handles orders: drivers only serve orders in their own grid.
- Collects metrics: reward/GMV, response rate, idle drivers, order count.
- Saves outputs to `results.pkl`.

The baseline runs the city simulation with **zero movement**, giving the **worst-case benchmark** for evaluating all RL algorithms.

run/run_IDQN.py I

Purpose: Train **Independent DQN (IDQN)** where each grid acts as an **independent agent** with its own Q-value decision.

What It Does:

- Loads real simulator data (orders, drivers, grid map).
- Initializes IDQN: one neural network, but decisions are **per-grid and uncoordinated**.
- Runs 25 training episodes:
  - Reset city for a full day (144 steps).
  - Convert city state $\rightarrow$ per-grid state features.
  - For each step: choose grid actions via Q-network, execute in simulator, compute reward, store experience, train from replay, update target network.
- Records metrics: GMV, response rate, dispatched drivers, conflicts.
- Saves logs and model checkpoints.

run/run_IDQN.py II

Trains separate Q-learning agents on each grid to move drivers without any coordination.

run/run_cDQN.py I

Purpose: Train **Contextual DQN (cDQN)** where **all grids share one Q-network** and decisions use **city-level context**.

What It Does:

- Loads real city data (orders, drivers, grid map).
- Initializes cDQN: one shared Q-network using a **context vector**.
- Runs 25 training episodes:
  - Reset city for a full day.
  - Build state = grid features + city context.
  - At each step: network outputs coordinated driver movement, simulator updates state, reward stored in replay memory.
- After each episode: large training phase (4000 updates), target network update, save model.
- Tracks metrics: GMV, response rate, conflicts, dispatched drivers.

run/run_cDQN.py II

Trains one shared Q-network that coordinates all grids using city context for better global rebalancing.

run/run_cA2C.py I

Purpose: Train **Contextual A2C (cA2C)** where each grid uses an **Actor** to choose actions and a **Critic** to evaluate them, both conditioned on **city context**.

What It Does:

- Loads real city data (orders, drivers, grid map).
- Initializes Actor network (driver movement) and Critic network (state value).
- Runs 25 training days:
  - At each step: Actor outputs actions; simulator applies them; reward + next state collected.
  - Stores data in two buffers: Critic replay (value targets) and Policy replay (advantages).
- After each episode: **4000 Critic updates + 4000 Actor updates**.

run/run_cA2C.py II

- Saves model and logs performance (GMV, response rate, conflicts, dispatched drivers).

Trains an Actor–Critic system that uses contextual city information to coordinate driver dispatch across all grids.

simulator/envs.py (CityReal) I

Purpose: Simulates a real city for ride-hailing dispatch using grids, drivers, orders, and time-based distributions.

Core Idea: CityReal is a **time-driven simulation** that models order arrivals, driver movement, ride durations, and prices, enabling RL algorithms (cDQN, cA2C, IDQN) to learn dispatching.

What CityReal Does:

- Represents city as $M \times N$ grid nodes (valid/invalid areas).
- Tracks drivers (idle, offline, online).
- Generates orders using historical distributions (Poisson, Gaussian).
- Simulates order durations, destinations, and prices.
- Controls online/offline drivers based on real data.
- Provides observations for RL (grid-wise idle drivers and active orders).

simulator/envs.py (CityReal) II

- Steps forward in time (10-minute intervals) for a full day.

Key Components:

- `construct_node_real` → creates nodes in valid areas.
- `utility_bootstrap_oneday_order` → samples one day's orders.
- `initial_order_random` / `step_generate_order_real` → generates timestep orders.
- `utility_add_driver_real` / `set_drivers_offline` → controls driver counts and locations.
- `get_observation` → returns 2-channel state for RL (drivers + orders).
- `reset` / `reset_clean` → initializes city for simulation.

Provides RL models a realistic, data-driven city environment to learn dispatching strategies matching real-world conditions.

simulator/objects.py I

Purpose:

- Simulates a ride-hailing environment with **Nodes**, **Drivers**, **Orders**, and **Order Distributions**.
- Handles order generation, driver status update, and simple order–driver assignment.

1. Distribution (Abstract Class)

- Defines interface for sampling random order counts.

PoissonDistribution

- Samples number of orders from Poisson(λ).

GaussianDistribution

- Samples number of orders from a Normal(μ, σ).

2. Node

- Represents a grid-cell (location) in the city.

simulator/objects.py II

- Stores:
  - Neighboring nodes
  - Active orders
  - Drivers (idle/offline)
- Generates orders (random or real distribution).
- Assigns orders to drivers within node or neighboring nodes.
- Updates/removes completed or expired orders.

3. Driver

- Represents a driver in the system.
- Tracks online/offline and serving/not-serving status.
- Takes orders and moves to destination when completed.
- Updates its internal clock and state each timestep.

4. Order

- Represents a customer request.

simulator/objects.py III

- Stores origin node, destination node, start time, duration, price.
- Tracks assigned time and waiting time.

simulator/utilities.py |

Purpose:

- Provides helper utilities for the simulation: file handling, time range generation, grid indexing, and neighbor search.

1. File & Time Utilities

- `datetime_range`: Generates timestamps between start and end with a fixed step.
- `mkdir_p`: Safe directory creation (like Linux `mkdir -p`).

2. Grid Index Conversion

- `ids_2dto1d`: Converts (row, col) to 1D index in an $M \times N$ grid.
- `ids_1dto2d`: Inverse mapping from 1D index back to (row, col).

3. Neighbor Computation

simulator/utilities.py II

- `get_neighbor_list`: Returns actual neighbor **Node objects** for hexagonal (6-side) or square (4-side) grid.
- `get_neighbor_index`: Returns neighbor coordinates around a cell.
- `get_layers_neighbors`: Finds neighbors layer-by-layer (1-hop, 2-hop, ..., L hops).

4. Driver Node State Utilities

- `get_driver_status`: Builds a matrix of idle drivers over the grid.
- `debug_print_drivers`: Prints debugging info of all drivers in a node.

tests/run_example.py I

Purpose

- Runs a small-scale simulation using the `CityReal` environment.
- Tests how orders, drivers, and grid locations interact over 144 time steps.
- Prints order response rate and driver statistics at each step.

Classes

- **CityReal**: Simulates city-level ride-hailing operations. Manages orders, drivers, grid states, and time evolution.
- **Grid/Mapping Utilities (from `simulator.envs`)**: Convert city map into grid representation; support movement, neighborhood, and indexing.
- **Driver State Manager**: Tracks idle, active, and on/off drivers during simulation.

tests/run_example.py II

- **Order Generator**: Produces synthetic orders based on duration, price distribution, and grid locations.

Core Simulation Flow

1. Load a 3×3 grid map and generate:
   - order distribution per grid,
   - idle driver distribution,
   - real orders for each time step.
2. Create the `CityReal` environment with these inputs.
3. Reset the environment and run 144 iterations:
   - At each step call `env.step()` with empty dispatch action.
   - Environment internally:
     3.1 Generates new orders.
     3.2 Matches drivers (if possible).
     3.3 Updates city state and metrics.
4. Print order response rate and driver statistics at each step.
5. At the end, compute mean order response rate.

tests/run_example.py III

Outcome

- Demonstrates basic driver-order dynamics.
- Shows how response rate changes under no-dispatch policy.

Baseline Simulation: run_example.py I

Purpose: Evaluate city simulator **without any RL policy** (drivers do not move). Serves as **benchmark**.

Simulation Setup:

- City grid: 3×3 (mapped_matrix_int)
- Time steps: 144 (full day)
- Idle driver distribution: 2 per grid per timestep
- Orders: predefined set per timestep
- No rebalancing actions (dispatch_action $= []$)

Metrics Collected:

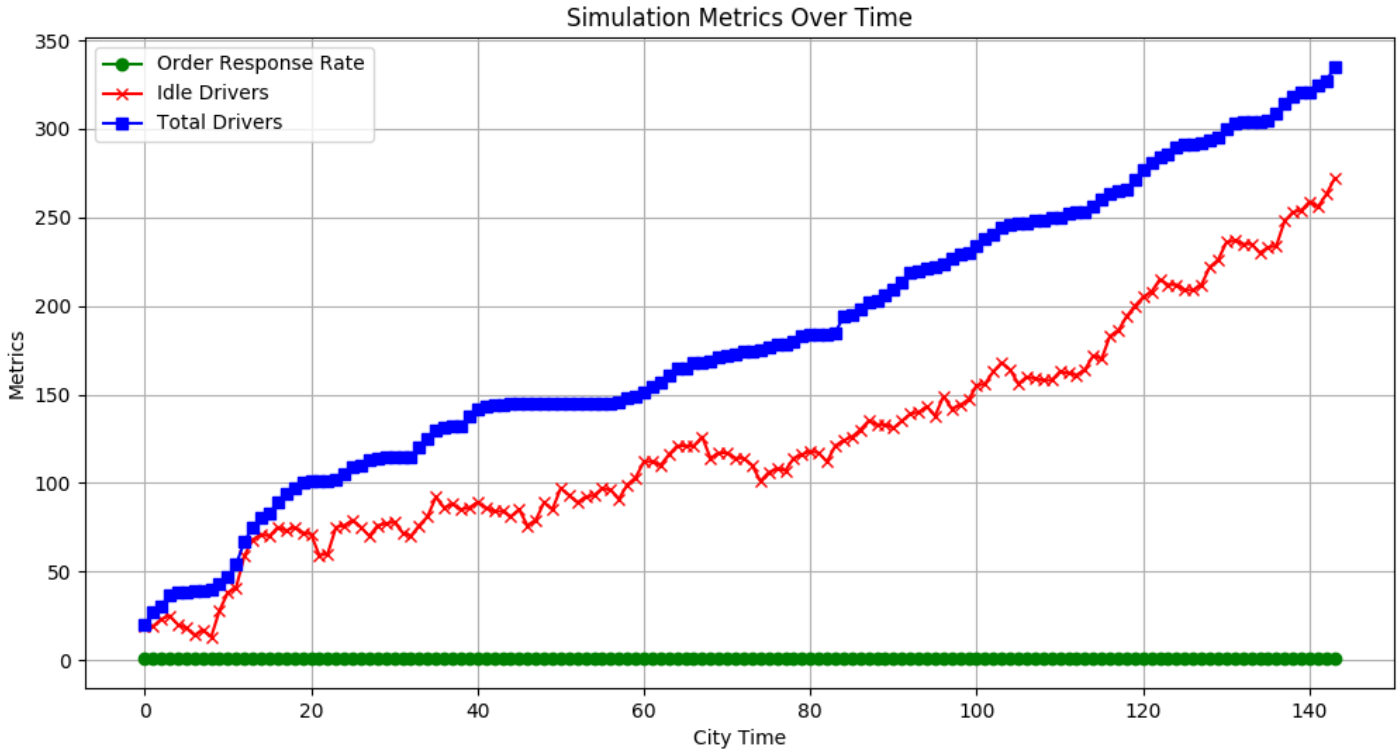
- Order Response Rate
- Idle Drivers
- Total Drivers

Baseline Simulation: run_example.py II

Results:

- Order response rate: **avg** $\approx$ **0.998** over 144 steps
- Idle drivers fluctuate as drivers serve orders within grids
- Total drivers remain constant (no new drivers added)

Visualization: Plotted **Order Response Rate**, **Idle Drivers**, and **Total Drivers** over time for quick analysis.


Simulation Metrics Over Time
Order Response Rate
Idle Drivers
Total Drivers
Metrics
City Time
0
50
100
150
200
250
300
350
0
20
40
60
80
100
120
140

Frame Title

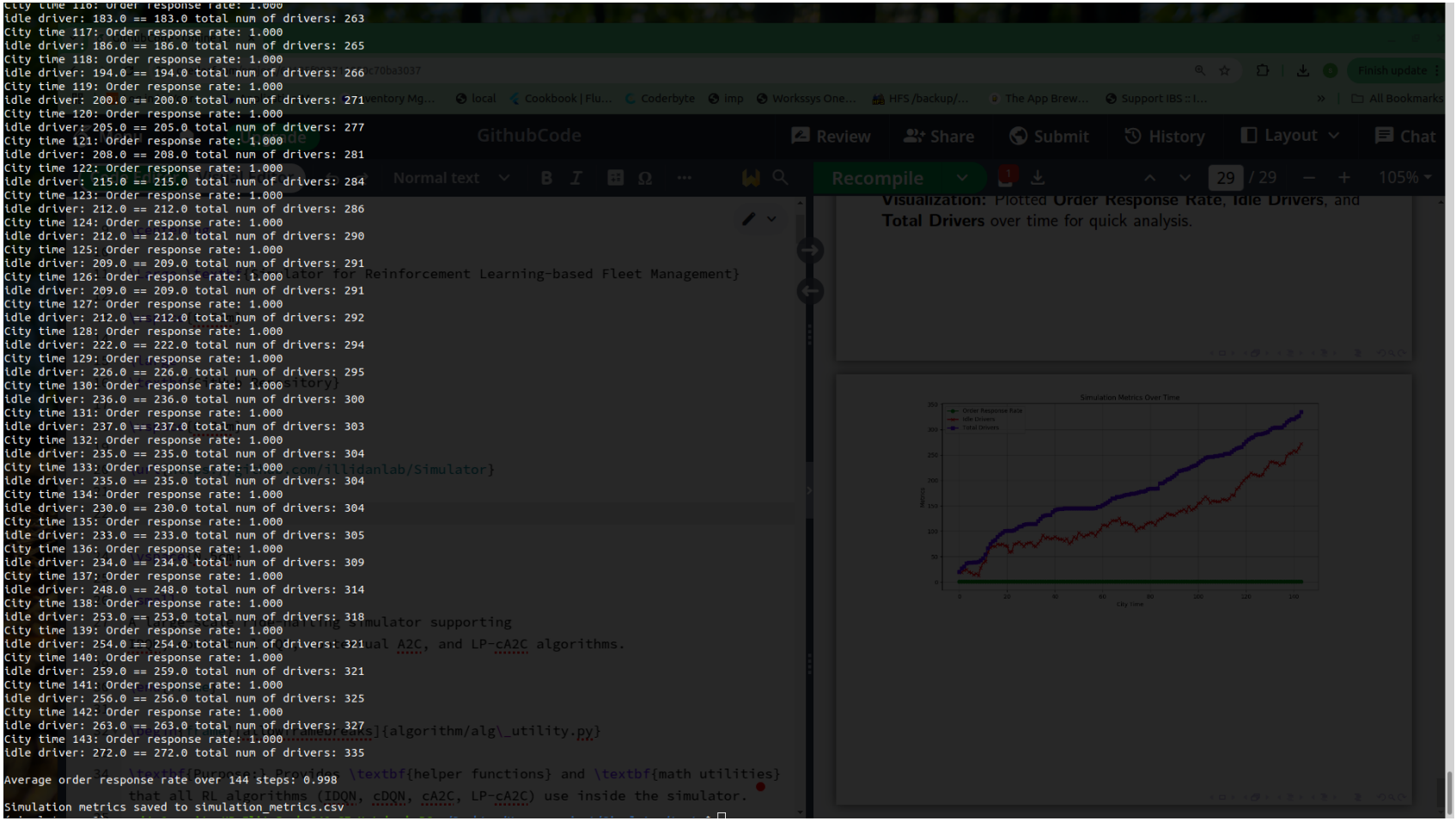


Figure: Screenshot of result from tests/run_example.py