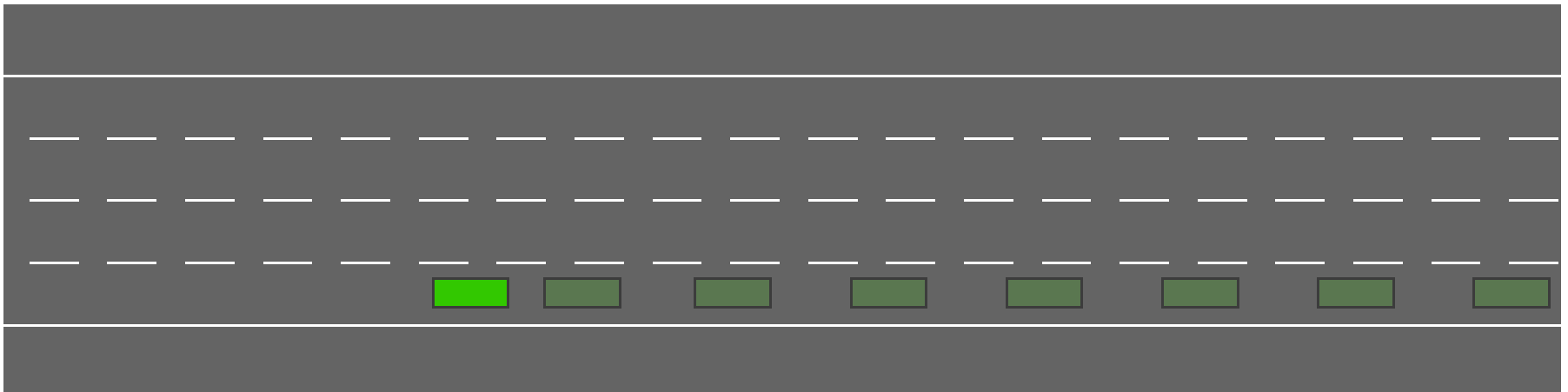


Lab3 DQN for Highway Driving

Zonghua Gu 2021

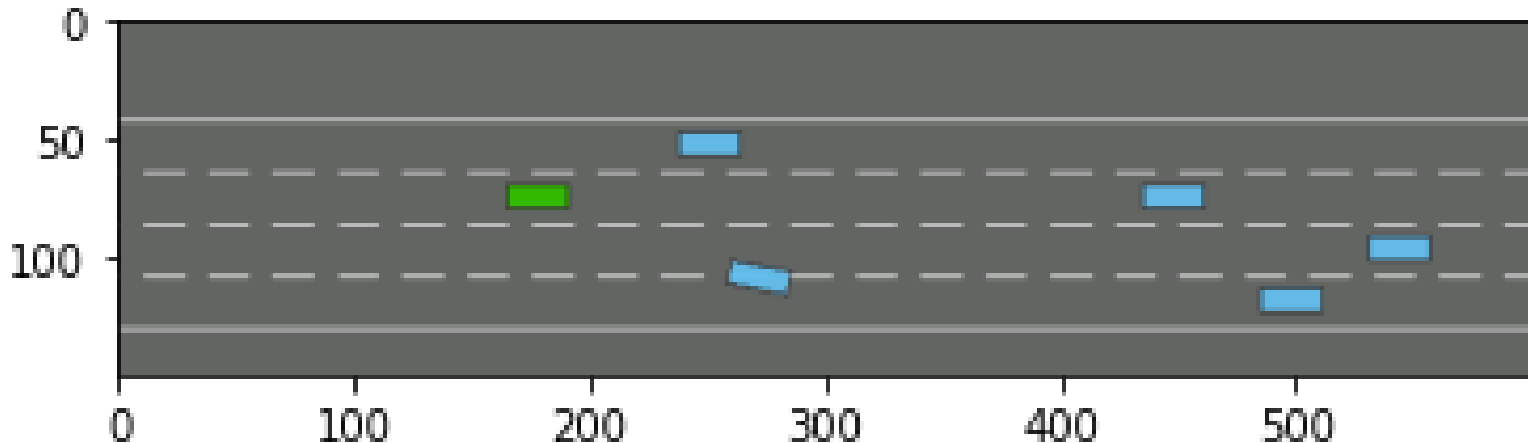
Highway Env

- A collection of environments for autonomous driving and tactical decision-making tasks, by Edouard Leurent
 - Source code:
<https://github.com/eleurent/highway-env>
 - Documentation:
<https://eleurent.github.io/highway-env/>



Making an environment

- `import gym`
- `import highway_env`
- `from matplotlib import pyplot as plt`
- `%matplotlib inline`
- `env = gym.make('highway-v0')`
- `# 5 environments: Highway, Merge, Roundabout, Parking, Intersection,`
- `env.reset()`
- `for _ in range(3):`
 - `action = env.action_type.actions_indexes["IDLE"]`
 - `obs, reward, done, info = env.step(action)`
 - `env.render()`
- `plt.imshow(env.render(mode="rgb_array"))`
- `plt.show()`



Training an agent

- RL agents can be trained using libraries such as rl-agents (by Leurent), OpenAI baselines or stable-baselines3.

rl-agents

- A collection of RL agents authored by Leurent: <https://github.com/eleurent/rl-agents>
- Planning
 - [Value Iteration](#)
 - [Cross-Entropy Method](#)
 - Monte-Carlo Tree Search
 - [Upper Confidence Trees](#)
 - [Deterministic Optimistic Planning](#)
 - [Open Loop Optimistic Planning](#)
 - [Trailblazer](#)
 - [PlatyPOOS](#)
- Safe planning
 - [Robust Value Iteration](#)
 - [Discrete Robust Optimistic Planning](#)
 - [Interval-based Robust Planning](#)
- Value-based
 - [Deep Q-Network](#)
 - [Fitted-Q](#)
- Safe value-based
 - [Budgeted Fitted-Q](#)

Stable Baselines

- A set of improved implementations of RL algorithms based on OpenAI Baselines:
<https://github.com/DLR-RM/stable-baselines3>
- Training a PPO (Proximal Policy Gradient) agent with Stable Baselines:

```
import gym

from stable_baselines.common.policies import MlpPolicy
from stable_baselines import PPO2

env = gym.make('CartPole-v1')

model = PPO2(MlpPolicy, env, verbose=1)
# Train the agent
model.learn(total_timesteps=10000)

# Enjoy trained agent
obs = env.reset()
for i in range(1000):
    action, _states = model.predict(obs, deterministic=False)
    obs, reward, done, info = env.step(action)
    env.render()
    if done:
        obs = env.reset()
env.close()
```

```

import gym
import highway_env
import numpy as np

from stable_baselines import HER, SAC, DDPG, TD3
from stable_baselines.ddpg import NormalActionNoise

env = gym.make("parking-v0")

# Create 4 artificial transitions per real transition
n_sampled_goal = 4

# SAC hyperparams:
model = HER('MlpPolicy', env, SAC, n_sampled_goal=n_sampled_goal,
            goal_selection_strategy='future',
            verbose=1, buffer_size=int(1e6),
            learning_rate=1e-3,
            gamma=0.95, batch_size=256,
            policy_kwargs=dict(layers=[256, 256, 256]))

model.learn(int(2e5))
model.save('her_sac_highway')

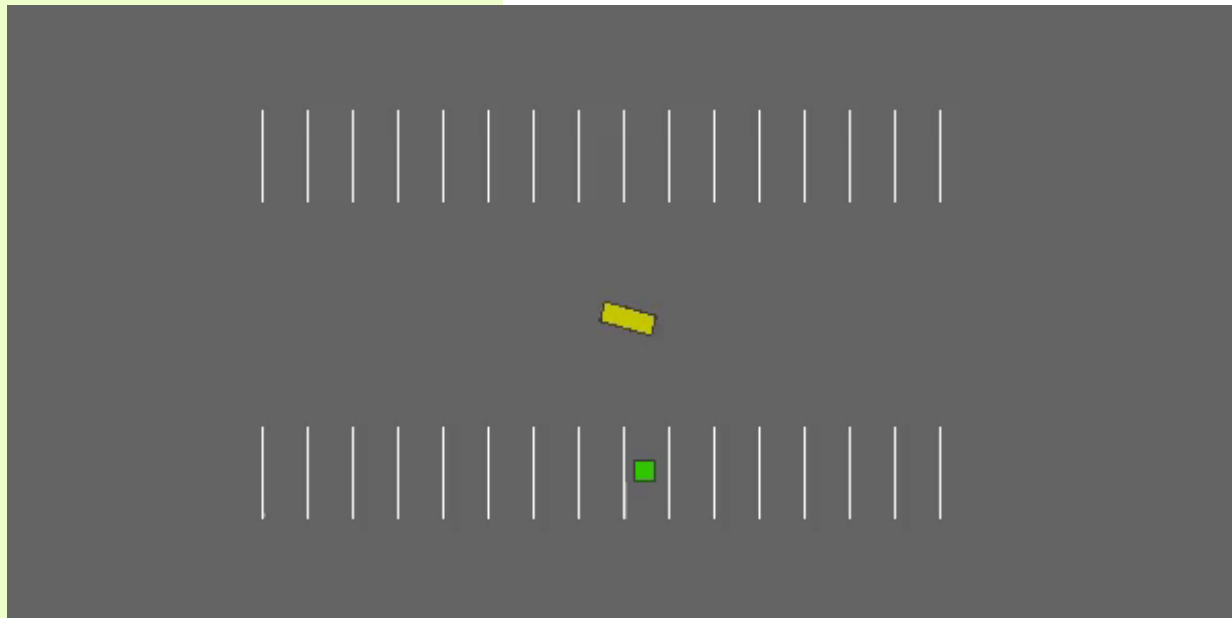
# Load saved model
model = HER.load('her_sac_highway', env=env)

obs = env.reset()

# Evaluate the agent
episode_reward = 0
for _ in range(100):
    action, _ = model.predict(obs)
    obs, reward, done, info = env.step(action)
    env.render()
    episode_reward += reward
    if done or info.get('is_success', False):
        print("Reward:", episode_reward, "Success?", info.get('is_success', False))
        episode_reward = 0.0
        obs = env.reset()

```

highway-parking-v0
environment trained with HER
(Hierarchical Experience
replay).



highway_env.py

- The vehicle is driving on a straight highway with several lanes, and is rewarded for reaching a high speed, staying on the rightmost lanes and avoiding collisions.
- The observations, actions, dynamics and rewards of an environment are parametrized by a configuration, defined as a config dictionary. After environment creation, the configuration can be accessed using the config attribute. Here are the default config values:

observation.py

- GrayscaleObservation(ObservationType)
 - Observes the image rendered by the simulator (top-down view)
- KinematicObservation(ObservationType)
 - Observes the kinematics (position and speed) of all nearby vehicles within PERCEPTION_DISTANCE=6.0*MDPVehicle.SPEED_MAX
- LidarObservation(ObservationType)
 - Observes direction and distance to obstacles within line of sight

```
def observation_factory(env: 'AbstractEnv', config: dict) -> ObservationType:
    if config["type"] == "TimeToCollision":
        return TimeToCollisionObservation(env, **config)
    elif config["type"] == "Kinematics":
        return KinematicObservation(env, **config)
    elif config["type"] == "OccupancyGrid":
        return OccupancyGridObservation(env, **config)
    elif config["type"] == "KinematicsGoal":
        return KinematicsGoalObservation(env, **config)
    elif config["type"] == "GrayscaleObservation":
        return GrayscaleObservation(env, **config)
    elif config["type"] == "AttributesObservation":
        return AttributesObservation(env, **config)
    elif config["type"] == "MultiAgentObservation":
        return MultiAgentObservation(env, **config)
    elif config["type"] == "LidarObservation":
        return LidarObservation(env, **config)
    elif config["type"] == "ExitObservation":
        return ExitObservation(env, **config)
    else:
        raise ValueError("Unknown observation type")
```

action.py

- class ContinuousAction(ActionType)
 - Continuous action space for throttle and/or steering angle. If both throttle and steering are enabled, they are set in this order: [throttle, steering]. The space intervals are always $[-1, 1]$, but are mapped to throttle/steering intervals through configurations.
 - ACCELERATION_RANGE = $(-5, 5.0)$
 - $[-x, x]$, in m/s^2
 - STEERING_RANGE = $(-\text{np.pi} / 4, \text{np.pi} / 4)$
 - $[-x, x]$, in rad
- class DiscreteMetaAction(ActionType)
 - Discrete action space of meta-actions: lane changes, and cruise control set-point.
 - ACTIONS_ALL = {0: 'LANE_LEFT', 1: 'IDLE', 2: 'LANE_RIGHT', 3: 'FASTER', 4: 'SLOWER'}
 - A mapping of action indexes to labels.
 - ACTIONS_LONGI = {0: 'SLOWER', 1: 'IDLE', 2: 'FASTER'}
 - A mapping of longitudinal action indexes to labels.
 - ACTIONS_LAT = {0: 'LANE_LEFT', 1: 'IDLE', 2: 'LANE_RIGHT'}
 - A mapping of lateral action indexes to labels.

```
:param env: the environment
:param longitudinal: include longitudinal actions
:param lateral: include lateral actions
"""
```

```
def action_factory(env: 'AbstractEnv', config: dict) -> ActionType:
    if config["type"] == "ContinuousAction":
        return ContinuousAction(env, **config)
    elif config["type"] == "DiscreteMetaAction":
        return DiscreteMetaAction(env, **config)
    else:
        raise ValueError(f"Unknown action type: {config['type']}")

    super().__init__(env)
    self.longitudinal = longitudinal
    self.lateral = lateral
    self.actions = self.ACTIONS_ALL if longitudinal and lateral \
        else self.ACTIONS_LONGI if longitudinal \
        else self.ACTIONS_LAT if lateral \
        else None
```

Actions are controller targets

- The `:py:class:`~highway env.envs.common.action.DiscreteMetaAction`` type adds a layer of `:ref:`speed and steering controllers <vehicle controller>`` on top of the continuous low-level control, so that the ego-vehicle can automatically follow the road at a desired velocity. Then, the available **meta-actions** consist in *changing the target lane and speed* that are used as setpoints for the low-level controllers.

vehicle/controller.py

- A vehicle piloted by two low-level controllers, allowing high-level actions such as cruise control and lane changes.
 - The longitudinal controller is a speed controller;
 - The lateral controller is a heading controller cascaded with a lateral position controller.
 - Control algorithm is Proportional control.
 - Vehicle model is dynamical bicycle model, with tire friction and slipping.

```
def act(self, action: Union[dict, str] = None) -> None:
    """
    Perform a high-level action to change the desired lane or speed.

    - If a high-level action is provided, update the target speed and lane;
    - then, perform longitudinal and lateral control.

    :param action: a high-level action
    """
    self.follow_road()
    if action == "FASTER":
        self.target_speed += self.DELTA_SPEED
    elif action == "SLOWER":
        self.target_speed -= self.DELTA_SPEED
    elif action == "LANE_RIGHT":
        _from, _to, _id = self.target_lane_index
        target_lane_index = _from, _to, np.clip(_id + 1, 0, len(self.road.network.graph[_from][_to]) - 1)
        if self.road.network.get_lane(target_lane_index).is_reachable_from(self.position):
            self.target_lane_index = target_lane_index
    elif action == "LANE_LEFT":
        _from, _to, _id = self.target_lane_index
        target_lane_index = _from, _to, np.clip(_id - 1, 0, len(self.road.network.graph[_from][_to]) - 1)
        if self.road.network.get_lane(target_lane_index).is_reachable_from(self.position):
            self.target_lane_index = target_lane_index

    action = {"steering": self.steering_control(self.target_lane_index),
             "acceleration": self.speed_control(self.target_speed)}
    action['steering'] = np.clip(action['steering'], -self.MAX_STEERING_ANGLE, self.MAX_STEERING_ANGLE)
    super().act(action)
```

highway_env.py default_config

- In `def default_config(cls) -> dict`:
 - `"collision_reward": -1,` # The reward received when colliding with a vehicle.
 - `"right_lane_reward": 0.1,` # The reward received when driving on the right-most lanes, linearly mapped to zero for other lanes.
 - `"high_speed_reward": 0.4,` # The reward received when driving at full speed, linearly mapped to zero for lower speeds according to `config["reward_speed_range"]`.
 - `"lane_change_reward": 0,` # The reward received at each lane change action.
 - `"reward_speed_range": [20, 30],`

highway_env.py _reward()

```
def _reward(self, action: Action) -> float:
    """
    The reward is defined to foster driving at high speed, on the rightmost lanes, and to avoid collisions
    :param action: the last action performed
    :return: the corresponding reward
    """
    neighbours = self.road.network.all_side_lanes(self.vehicle.lane_index)
    lane = self.vehicle.target_lane_index[2] if isinstance(self.vehicle, ControlledVehicle) \
        else self.vehicle.lane_index[2]
    scaled_speed = utils.lmap(self.vehicle.speed, self.config["reward_speed_range"], [0, 1])
    reward = \
        + self.config["collision_reward"] * self.vehicle.crashed \
        + self.config["right_lane_reward"] * lane / max(len(neighbours) - 1, 1) \
        + self.config["high_speed_reward"] * np.clip(scaled_speed, 0, 1)
    reward = utils.lmap(reward,
                        [self.config["collision_reward"],
                         self.config["high_speed_reward"] + self.config["right_lane_reward"]],
                        [0, 1])
    reward = 0 if not self.vehicle.on_road else reward
    return reward
```

highway_env.py _reward()

Explanations

- If crashed, add collision_reward (-1)
- Add right_lane_reward * Lane / max(nLanes-1, 1)
 - lane_index has 3 elements (from, to, id), so lane_index[2] is the lane id. For self.vehicle, consider the target lane; for other vehicles, consider the current lane
 - neighbours contains all lanes in the same road.
 - Suppose nLanes=2, if lane=0 (left lane), then $.1 * \frac{0}{1} = 0$; if lane=1 (right lane), then $.1 * \frac{1}{1} = .1$
- utils.lmap(v: float, x: Interval, y: Interval) -> float
 - Linear map of value v within range $x=[x_0, x_1]$ to desired range $y=[y_0, y_1] = [0,1]$, returns $y_0 + \frac{(v-x_0)(y_1-y_0)}{(x_1-x_0)} \in [0,1]$
- Add high_speed_reward * scaled_speed = .4 * np.clip(scaled_speed, 0, 1)
 - scaled_speed = utils.lmap(self.vehicle.speed, self.config["reward_speed_range"], [0, 1])
 - np.clip(scaled_speed, 0, 1) (if $v \notin [20,30]$, clip output to within [0,1])
- reward = utils.lmap(reward, [-1, .5], [0,1])
 - Min reward=-1 (collision_reward); Max reward=.1+.4 (right_lane_reward+ high_speed_reward);

roundabout_env.py _reward(self, action: int)

- In def default_config(cls) -> dict: "collision_reward": -1, "high_speed_reward": 0.2, "right_lane_reward": 0, "lane_change_reward": -0.05
- If crashed, add collision_reward(-1)
- Add high_speed_reward*scaled speed index
- Add lane_change_reward*lane_change
- reward = utils.lmap(reward, [-1.05, .2], [0,1])

```
def _reward(self, action: int) -> float:
    lane_change = action == 0 or action == 2
    reward = self.config["collision_reward"] * self.vehicle.crashed \
        + self.config["high_speed_reward"] * \
            MDPVehicle.get_speed_index(self.vehicle) / max(MDPVehicle.SPEED_COUNT - 1, 1) \
        + self.config["lane_change_reward"] * lane_change
    return utils.lmap(reward,
                      [self.config["collision_reward"] + self.config["lane_change_reward"],
                       self.config["high_speed_reward"]], [0, 1])
```


create_road(), create_vehicles()

```
def _create_road(self) -> None:
    """Create a road composed of straight adjacent lanes."""
    self.road = Road(network=RoadNetwork.straight_road_network(self.config["lanes_count"], speed_limit=30),
                     np_random=self.np_random, record_history=self.config["show_trajectories"])











def _create_vehicles(self) -> None:
    """Create some new random vehicles of a given type, and add them on the road."""
    other_vehicles_type = utils.class_from_path(self.config["other_vehicles_type"])
    other_per_controlled = near_split(self.config["vehicles_count"], num_bins=self.config["controlled_vehicles"])

    self.controlled_vehicles = []
    for others in other_per_controlled:
        controlled_vehicle = self.action_type.vehicle_class.create_random(
            self.road,
            speed=25,
            lane_id=self.config["initial_lane_id"],
            spacing=self.config["ego_spacing"]
        )
        self.controlled_vehicles.append(controlled_vehicle)
        self.road.vehicles.append(controlled_vehicle)

    for _ in range(others):
        self.road.vehicles.append(
            other_vehicles_type.create_random(self.road, spacing=1 / self.config["vehicles_density"])
        )
```

agents

- In random.py:
- `def act(self, state):`
 `return`
 `self.env.action_space.sample()`
- In deep_q_network/abstract.py:
- `def act(self, state,`
 `step_exploration_time=True):`
 `if step_exploration_time:`
 `self.exploration_policy.step_time()`
 `values =`
 `self.get_state_action_values(state)`
 `self.exploration_policy.update(values)`
 `return self.exploration_policy.sample()`

	<code>budgeted_ftq</code>
	<code>common</code>
	<code>control</code>
	<code>cross_entropy_method</code>
	<code>deep_q_network</code>
	<code>dynamic_programming</code>
	<code>fitted_q</code>
	<code>robust</code>
	<code>simple</code>
	<code>tree_search</code>

Env and Agent Configs

- `env_config = 'configs/HighwayEnv/env.json'`
- `agent_config = 'configs/HighwayEnv/agents/DQNAgent/dqn.json'`

```
1  {
2      "id": "highway-v0",
3      "import_module": "highway_env"
4  }
    env.json

1  {
2      "__class__": "<class 'rl_agents.agents.deep_q_network.pytorch.DQNAgent'>",
3      "model": {
4          "type": "MultiLayerPerceptron",
5          "layers": [256, 256]
6      },
7      "double": false,
8      "gamma": 0.8,
9      "n_steps": 1,
10     "batch_size": 32,
11     "memory_capacity": 15000,
12     "target_update": 50,
13     "exploration": {
14         "method": "EpsilonGreedy",
15         "tau": 6000,
16         "temperature": 1.0,
17         "final_temperature": 0.05
18     },
19     "loss_function": "l2"
20 }
```

dqn.json

abstract.py exploration_factory

```
def exploration_factory(exploration_config, action_space):  
    """  
        Handles creation of exploration policies  
    :param exploration_config: configuration dictionary of the policy, must contain a "method" key  
    :param action_space: the environment action space  
    :return: a new exploration policy  
    """  
  
    from rl_agents.agents.common.exploration.boltzmann import Boltzmann  
    from rl_agents.agents.common.exploration.epsilon_greedy import EpsilonGreedy  
    from rl_agents.agents.common.exploration.greedy import Greedy  
  
    if exploration_config['method'] == 'Greedy':  
        return Greedy(action_space, exploration_config)  
    elif exploration_config['method'] == 'EpsilonGreedy':  
        return EpsilonGreedy(action_space, exploration_config)  
    elif exploration_config['method'] == 'Boltzmann':  
        return Boltzmann(action_space, exploration_config)  
    else:  
        raise ValueError("Unknown exploration method")
```

epsilon_greedy.py get_distribution

- n : discrete action space size (# actions)
- For each action: $dist = \frac{\epsilon}{n}$
- For the optimal action: $dist = \frac{\epsilon}{n} + 1 - \epsilon$

```
def get_distribution(self):  
    distribution = {action: self.epsilon / self.action_space.n for action in range(self.action_space.n)}  
    distribution[self.optimal_action] += 1 - self.epsilon  
    return distribution
```

epsilon_greedy.py

- $\epsilon = finalT + (T - finalT)e^{-\frac{time}{\tau}}$
- $\frac{time}{\tau} = 0 \Rightarrow \epsilon = T = 1.0$
- $\frac{time}{\tau} = \infty \Rightarrow \epsilon = finalT = .1$
- Hyperparam τ determines the speed of change from T to $finalT$

```
def default_config(cls):
    return dict(temperature=1.0,
                final_temperature=0.1,
                tau=5000)

def update(self, values):
    """
        Update the action distribution parameters
    :param values: the state-action values
    :param step_time: whether to update epsilon schedule
    """
    self.optimal_action = np.argmax(values)
    self.epsilon = self.config['final_temperature'] + \
        (self.config['temperature'] - self.config['final_temperature']) * \
        np.exp(- self.time / self.config['tau'])
```