Traffic Light Control RL Code

This markdown file contains a condensed version of the `sumo_traffic_rl.py` script, suitable for inclusion in a report.

```
"""
Improved Traffic Light Control using Deep Q-Learning
Condensed version for report inclusion.
"""
import os, sys, random, time
import numpy as np
from collections import deque
import matplotlib.pyplot as plt

# SUMO Setup
SUMO_HOME = r"C:\Program Files (x86)\Eclipse\Sumo"
os.environ['SUMO_HOME'] = SUMO_HOME
sys.path.append(os.path.join(SUMO_HOME, 'tools'))
import traci, sumolib

class ImprovedSUMOEnv:
    def __init__(self, config_file, use_gui=False):
        self.sumo_cmd = [sumolib.checkBinary('sumo-gui' if use_gui else 'sumo'), '-
c', config_file, '--no-warnings']
        self.state_size, self.action_size, self.max_steps = 20, 4, 3600
        self.yellow_time, self.green_time = 3, 10
        self.reset()

    def reset(self):
        if traci.isLoaded(): traci.close()
        traci.start(self.sumo_cmd)
        self.tl_id = traci.trafficlight.getIDList()[0] if
traci.trafficlight.getIDList() else None
        self.step_count = self.total_waiting_time = self.total_vehicles = 0
        return self._get_state()

    def _get_state(self):
        state = []
        if self.tl_id:
            for lane in
list(dict.fromkeys(traci.trafficlight.getControlledLanes(self.tl_id)))[:5]:
                state.extend([traci.lane.getLastStepHaltingNumber(lane)/10.0,
traci.lane.getWaitingTime(lane)/100.0,
                              traci.lane.getLastStepVehicleNumber(lane)/10.0,
traci.lane.getLastStepMeanSpeed(lane)/15.0])
        return np.array((state + [0]*20)[:20], dtype=np.float32)

    def step(self, action):
        old_wait, old_queue = self._get_metrics()
```

```
        if self.tl_id:
            curr = traci.trafficlight.getPhase(self.tl_id)
            if action != curr:
                traci.trafficlight.setPhase(self.tl_id, (curr + 1) % 8)
                for _ in range(self.yellow_time): traci.simulationStep();
self.step_count += 1
                traci.trafficlight.setPhase(self.tl_id, action * 2)
        for _ in range(self.green_time): traci.simulationStep(); self.step_count +=
1
        new_wait, new_queue = self._get_metrics()
        reward = (old_wait - new_wait)/10.0 + (old_queue - new_queue)*2.0 -
new_wait/100.0 - new_queue/5.0
        done = self.step_count >= self.max_steps or
traci.simulation.getMinExpectedNumber() <= 0
        self.total_waiting_time += new_wait; self.total_vehicles +=
self._get_veh_count()
        return self._get_state(), reward, done

    def _get_metrics(self):
        lanes = set(traci.trafficlight.getControlledLanes(self.tl_id)) if
self.tl_id else []
        return sum(traci.lane.getWaitingTime(l) for l in lanes),
sum(traci.lane.getLastStepHaltingNumber(l) for l in lanes)

    def _get_veh_count(self):
        return sum(traci.lane.getLastStepVehicleNumber(l) for l in
set(traci.trafficlight.getControlledLanes(self.tl_id)))

class ImprovedDQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size, self.action_size = state_size, action_size
        self.gamma, self.epsilon, self.epsilon_min, self.epsilon_decay, self.lr =
0.99, 1.0, 0.01, 0.9995, 0.0005
        self.memory, self.batch_size = deque(maxlen=5000), 64
        self.model = self._build_model()
        self.target_model = [w.copy() for w in self.model]
        self.train_count = 0

    def _build_model(self):
        sizes = [self.state_size, 128, 128, 64, self.action_size]
        return [np.random.randn(sizes[i], sizes[i+1]) * np.sqrt(2.0/sizes[i]) for i
in range(len(sizes)-1)] + \
               [np.zeros((1, s)) for s in sizes[1:]]

    def act(self, state):
        if np.random.rand() <= self.epsilon: return
random.randrange(self.action_size)
        return np.argmax(self._forward(state)[0])

    def _forward(self, x, model=None):
        model = model or self.model
        activations = [x.reshape(1, -1)]
        for i in range(3):
            z = np.dot(activations[-1], model[i]) + model[i+4]
            activations.append(np.maximum(0, z))
        return np.dot(activations[-1], model[3]) + model[7], activations
```

```
    def replay(self):
        if len(self.memory) < self.batch_size: return
        batch = random.sample(self.memory, self.batch_size)
        for s, a, r, ns, d in batch:
            target = r + (0 if d else self.gamma * np.max(self._forward(ns,
self.target_model)[0]))
            curr_q, acts = self._forward(s)
            target_q = curr_q.copy(); target_q[0, a] = target
            self._train(acts, target_q)
        self.train_count += 1
        if self.train_count % 10 == 0: self.target_model = [w.copy() for w in
self.model]
        if self.epsilon > self.epsilon_min: self.epsilon *= self.epsilon_decay

    def _train(self, acts, target):
        d_out = 2 * (acts[-1] @ self.model[3] + self.model[7] - target) /
self.action_size
        grads = [None]*8
        grads[3], grads[7] = acts[3].T @ d_out, np.sum(d_out, axis=0,
keepdims=True)
        delta = (d_out @ self.model[3].T) * (acts[3] > 0)
        for i in range(2, -1, -1):
            grads[i], grads[i+4] = acts[i].T @ delta, np.sum(delta, axis=0,
keepdims=True)
            if i > 0: delta = (delta @ self.model[i].T) * (acts[i] > 0)
        for i in range(8): self.model[i] -= self.lr * np.clip(grads[i], -1, 1)

def train(config, episodes=30):
    env = ImprovedSUMOEnv(config); agent = ImprovedDQNAgent(env.state_size,
env.action_size)
    for ep in range(episodes):
        state = env.reset(); total_reward = 0
        while True:
            action = agent.act(state)
            ns, r, done = env.step(action)
            agent.memory.append((state, action, r, ns, done))
            if len(agent.memory) >= agent.batch_size: agent.replay()
            state, total_reward = ns, total_reward + r
            if done: break
        print(f"Ep {ep+1}: Reward {total_reward:.2f}, Wait
{env.total_waiting_time/env.step_count:.2f}s")
    env.close()

if __name__ == "__main__":
    train("cross3ltl.sumocfg")
```