CqSim - Gym.Env

Design Document

About

This documentation focuses on coupling CqSim Simulator with OpenAi Gym Interface to implement a Custom Gym Environment which can enable training of Reinforcement Learning Models using Libraries such as Tensorflow.

OpenAI - Gym.Env

OpenAI's gym is a package to create custom reinforcement learning agents. An environment contains all the necessary functionality to run an agent and allow it to learn. Each custom environment must implement the following gym interface :

```
import gym
from gym import spaces

class CustomEnv(gym.Env):
  """Custom Environment that follows gym interface"""
  metadata = {'render.modes': ['human']}

  def __init__(self, arg1, arg2, ...):
    super(CustomEnv, self).__init__()
    # Define action and observation space
    # They must be gym.spaces objects
    # Example when using discrete actions:
    self.action_space = spaces.Discrete(N_DISCRETE_ACTIONS)
    # Example for using image as input:
    self.observation_space = spaces.Box(low=0, high=255,
                                        shape=(HEIGHT, WIDTH, N_CHANNELS),
dtype=np.uint8)

  def step(self, action):
    ...
    return observation, reward, done, info
  def reset(self):
    ...
    return observation  # reward, done, info can't be included
  def render(self, mode='human'):
    ...
  def close (self):
    ...
```

CQSim

Is a Trace-based Event-Driven Scheduling Simulator.

Details about the CQSim implementation can be found in the following documents:

- CQSim Manual
- CQSim High-Level Design Document
- CQSim Low-level Design Document

CQSIM is formed by several modules, each one is implemented as a class. This document will focus on the component CqSim.Cqsim_sim.

Implementation using Multithreading:

Overview:

The notion behind the implementation is similar to the Producer-Consumer problem where the CqSim behaves as Consumer and Gym.Env.Step provides the Producer. Both CqSim and Gym.Env run on 2 separate threads and interact using shared variables.

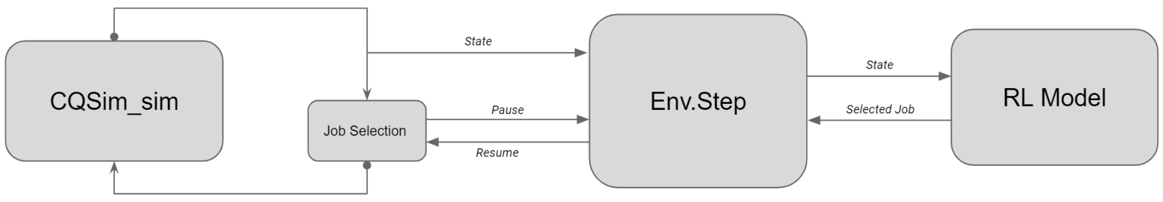

Fig.1 Overview of the flow for the multithreading approach.

Code Structure:

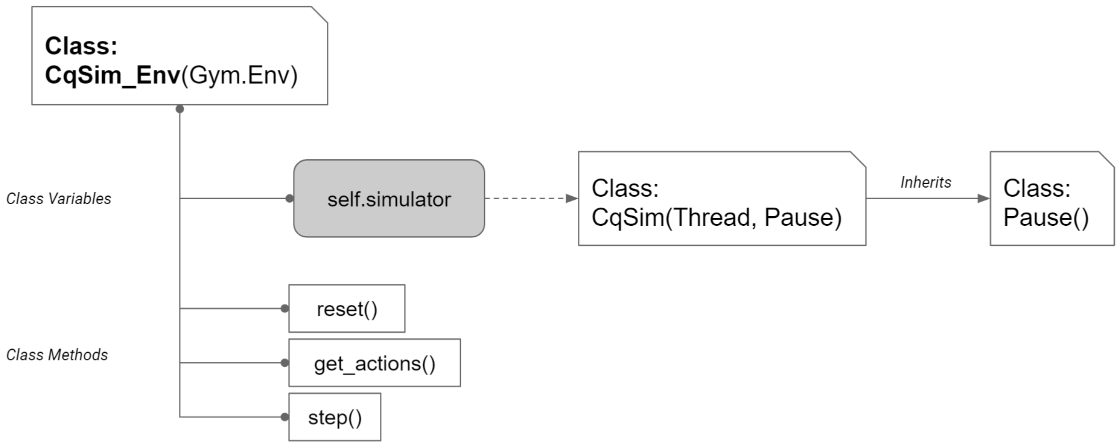

Fig.2 CqSim_Env Class structure.

The class **CqSim_Env** inherits the Gym.Env interface and implements all the abstract methods. It holds the **CqSim.Cqsim_sim** class object as its variable - **self.simulator**.

From the original implementation in CQSim Code, the class Cqsim.Cqsim_sim is altered in the following ways :

- Cqsim_sim class inherits the classes **Thread** and **Pause.**
- **Thread** class enables implementation of the object to be run on a parallel thread.
- The **Thread.run** method from the parent class is overwritten and executes the class method - **Cqsim_sim.cqsim_sim().** Which is the main function that coordinates the different components of the simulator.
- Class **Pause()** is implemented to control the synchronization between the **Cqsim_Env** main thread and **Cqsim_Env.simulator** thread.

Implementation:

- Once **Cqsim_Env** is initialised, it initiates the thread for **Cqsim_sim.**
- **Cqsim_sim** thread runs through the simulation steps :
    - Imports submit event.
    - Reaches event_job().
    - Initiates start_scan().

- At this point the **Cqsim_Env** thread is notified and **Cqsim_sim** thread gets paused and waits for the *notification* for resuming from **Cqsim_Env.step()**
- **Cqsim_Env** thread is initially paused and waits for the notification from Cqsim_sim thread.
- Once **Cqsim_Env** is resumed, it takes the current state of **Cqsim_sim** using the shared variable **Cqsim_Env.simulator.simulator_state.**
- This state is passed to the Model for making the selection of the job, and the shared variable is updated.
- At this point the **Cqsim_Env** thread is paused and the simulator thread resumes and reaches to the next event or next state.

Implementation of Thread Pause:

The **Pause** class helps implement the pausing and resuming of the Cqsim_Env(Producer) and Cqsim_sim(Consumer) threads and synchronize their communication.

The **Pause** class makes use of python Conditional Variables. It maintains 2 conditional variables - **Pause.prod_cv** and **Pause.cons_cv.**

In order to implement Producer-Consumer approach, when one cond. variable is paused the other one is resumed and vice-versa.

The **Pause** class also maintains a special variable - intial_check (boolean).
This variable is used to pause the Cqsim_env.get_state() method to the point when the Cqsim_sim thread is initiated and the first set of jobs is loaded.