Data flow for Highway-env DQN agent training

Legend: **Filenames, Paths,** Functions, Line numbers

To train the DQN agent navigate to the **rl-agents/scripts/** subdirectory and run the command:

```
python3 experiments.py evaluate configs/HighwayEnv/env.json
configs/HighwayEnv/agents/DQNAgent/dqn.json --train --episodes=2000 --name-from-config
```

The environment config file **rl-agents/scripts/configs/HighwayEnv/env.json** contains the gym id of the environment and the module to be imported.

The agent config file **rl-agents/scripts/configs/HighwayEnv/agents/DQNAgent/dqn.json** includes the name of class of the agent to be used (class **'rl_agents.agents.deep_q_network.pytorch.DQNAgent**), the type of the network to be used (a class from **rl-agents/rl_agents/agents/common/models.py**) as well as hyperparameters of the network.

In **experiments.py**, the function which incorporates the entire loop of the training/testing is the function evaluate on line 48. In evaluate, the environment is loaded based on the environment config file and the DQN agent is loaded based on the agent config file. The creation of the agent and the loading of the environment happens in **rl-agents/rl_agents/agents/common/factory.py**.

Next, an instance of the class Evaluation is instantiated on line 67 and the input arguments include the environment and the agent loaded from before. This class is imported from **rl-agents/rl_agents/trainer/evaluation.py**.

In this class in **evaluation.py**, an observation variable takes the value None on line 99. The function train is run on line 101, which calls the function run_episodes on line 106.

In the function run_episodes on line 122, a for loop runs for the total number of episodes. In this loop, there is another while loop where rewards are stored by calling the function step.

The function step (line 145) plans a sequence of actions according to the agent policy and steps the environment. It starts by calling the function plan on line 150 with the input argument of the observation variable from before.

This plan function can be traced to **rl-agents/rl_agents/agents/common/abstract.py** on line 38. It returns a list of optimal actions, given the initial state (observation) which is still None. More specifically, it returns a call to the function act (line 45), with the input argument of the state.

The function act can be found in **rl-agents/rl_agents/agents/deep_q_network/abstract.py**, on line 60. It returns an action given the state-action value model and the exploration policy. Initially, it sets a variable previous_state equal to the current state and then it calls the function get_state_action_values with the input argument of the current state, assigning the returned value on a variable named values.

The get_state_action_values function is in the same file on line 121 and returns the array of its action-values for each action. Namely, it returns a call to the function get_batch_state_action_values with an input argument of the current state in a list, and taking the 0th element of this list.

The get_batch_state_action_values function can be traced to **rl-agents/rl_agents/agents/deep_q_network/pytorch.py**, and is located on line 78. Its input is mentioned to be a variable called states. This function returns with a call to value_net with the states as input. The value_net is assigned a value on line 17, by calling the function model_factory, with the input being the part of the agent config file corresponding to the type of network to be used.

Therefore, value_net gets the form of an MLP instance from **rl-agents/rl_agents/agents/common/models.py** on line 49 and the input states get propagated along the network.

Going back to the function act (line 60) in **rl-agents/rl_agents/agents/deep_q_network/abstract.py**, the state-action values returned from the network are stored to a variable called values. After that, these values are fed to a function called exploration_policy.update.

The exploration policy leads us to **rl-agents/rl_agents/agents/common/exploration/abstract.py**, where, given the selected policy in the agent config file, the function exploration_factory (line 41) instantiates the selected policy (in this example EpsilonGreedy).

The function exploration_policy.update takes place in **rl-agents/rl_agents/agents/common/exploration/epsilon_greedy.py** (line 35), where the action distribution parameters are updated.

Finally, the act function in **rl-agents/rl_agents/agents/deep_q_network/abstract.py** (line 60), returns a sample from the action distribution computed from above, by calling the function exploration_policy.sample on line 69. The sample function is in **rl-agents/rl_agents/agents/common/exploration/abstract.py** (line 20).

This leads us back to **rl-agents/rl_agents/trainer/evaluation.py**, where we get the selected action values on line 150. After that the environment is updated on lines 160, 161, setting the previous observation equal to the current observation and setting the value of the variable action equal to the 0th element of actions that we got from the plan function above.

The environment is updated by obtaining values for the new observation, reward, query of terminal state on line 162, through the step function which traces back to **highway-env/highway_env/envs/highway_env.py** on line 48 and eventually in **highway-env/highway_env/envs/common/abstract.py** on line 165. This step function receives as input an action, steps the environment dynamics and returns the reward, observation and whether the current state is a terminal state.

Next, the agent.record function is called in **rl-agents/rl_agents/trainer/evaluation.py** (line 167) if training mode is on. It receives as input the previous observation, the action, reward, current observation and whether the current state is a terminal state. The function definition is located in **rl-agents/rl_agents/agents/deep_q_network/abstract.py** on line 36. It records a transition by performing a Deep Q-network iteration. First, it pushes the transition to memory on line 53 with the function push.

Then, it samples a minibatch with the function sample_minibatch (line 71) which calls the function memory.sample, located in **rl-agents/rl_agents/agents/common/memory.py** (line 37). It returns a batch of transitions.

Back in **rl-agents/rl_agents/agents/deep_q_network/abstract.py**, the loss is computed by calling the function compute_bellman_residual (line 56). This function is in **rl-agents/rl_agents/agents/deep_q_network/pytorch.py** (line 39). With the sampled batch as input, it computes the target state-action value (if not provided) and returns the loss over the input batch.

Next, in **rl-agents/rl_agents/agents/deep_q_network/abstract.py** the step_optimizer function is called (line 57) with the computed loss as input. This function is in **pytorch.py** (line 31) and performs gradient descent on the mlp.

Finally, back at **rl-agents/rl_agents/agents/deep_q_network/abstract.py**, the function update_target_network is called (line 58). This function is on line 77 and tracks the policy network with the target network.

Back at **evaluation.py**, step function returns the collected reward and whether the current state is terminal (line 171), run_episodes function aggregates the rewards on a list (line 132) and ends the episode by calling the functions after_all_episodes and after_some episodes.

Function after_all_episodes (line 303) does some storing of values in files, and function after_some_episodes (line 312) saves the best model.

This concludes the loop.