XRL Environment Technical Documentation

Allan

August 2023

1 Introduction

This document details the custom environment and software stack created for obtaining explainable reinforcement learning (RL) via a reward decomposed Deep Q-Network (DQN).

1.1 Software

This project makes extensive use of the PyTorch machine learning framework. PyTorch is used to create and train the policy and target networks for the DQN learning process. Additionally, the DQN learning code makes use of the prioritized memory replay buffer provided by the PyTorch reinforcement learning library: torchrl.

The custom environment was created using the Gymnasium library, an updated and maintained fork of the OpenAI Gym library. The reference documentation used to create the environment can be found here on the Gymnasium website. Internally, the custom environment uses the PyGame library to display the graphical state. This rendering occurs only if the user specifies the program to use human rendering mode. PyGame is also used to handle the keyboard and mouse inputs of the user in the graphical evaluation mode.

1.2 The Environment

This project considers a custom-made RL environment. An image of a rendered frame of the environment is shown in Figure 1. This graphical rendering of the environment only occurs when the human rendering mode is specified by the user on program launch. The RL agent is the green character. The primary objective of the agent is to destroy the target represented by the blue character. An obstacle in this environment is present in the form of the hard-coded red enemy character. Both the RL agent and the enemy can move around and fire bullets at each other (note the smaller owner-colored bullets in the render). The RL agent can also shoot at the target to destroy it, resulting in a victory

and successful episode termination. If the agent collides with the enemy or the enemy bullet, then the agent is destroyed resulting in a failure termination state.

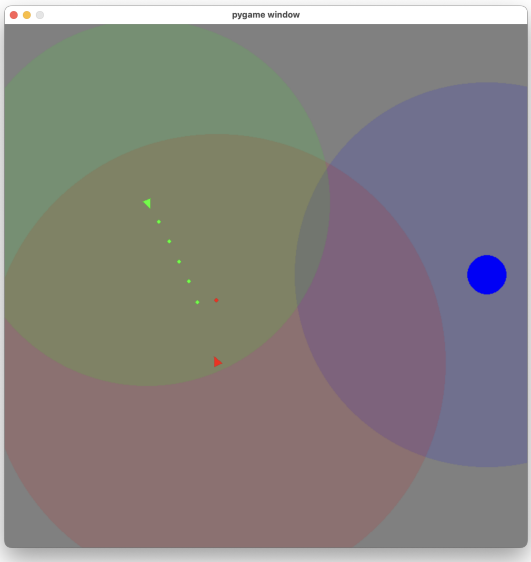

Figure 1: A screenshot of the environment

To complexify the environment and the RL process, some constraints have been included in the design of the environment. The agent can only successfully hit the target by firing within that blue targeting zone that surrounds the target. Both the agent and the enemy have limited observation ranges—the enemy will not react to the agent if is is outside of the red outer radius, and the agent won't receive valid observations of the enemy if it is outside of the green outer sphere. Additionally, any moves made by the agent that would result in it leaving the visible environment area are nullified, leaving it in place.

When the program is in the graphical evaluation mode, the user is able to interact with the environment by using the keyboard and mouse. The user can pause and unpause the simulation, rewind, and propose counterfactual trajectories for the RL agent. Proposing a counterfactual scenario is done by drawing an arbitrarily complex trajectory for the RL agent to consider. This drawing is done by clicking on the graphical environment window, which will extend a proposed trajectory path starting from the agent. Along the path, the user can also specify points at which the agent should shoot the target or the enemy. An example image of a proposed counterfactual trajectory is shown in Figure 2. The counterfactual proposal system also comes with basic editing features, including removing the most recent component added to the trajectory, and clearing the proposal altogether.

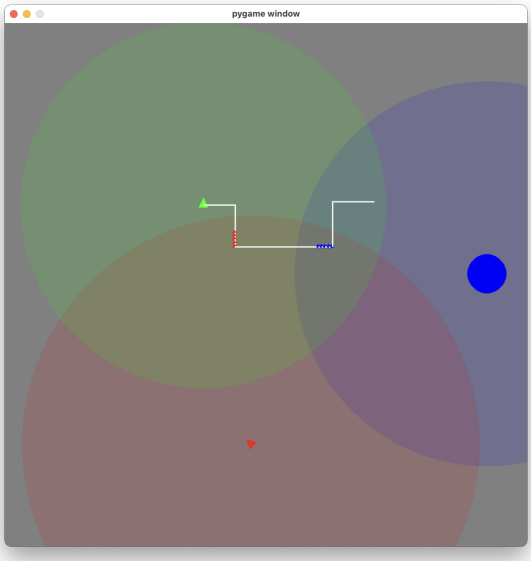

Figure 2: Interacting with the environment to propose a counterfactual trajectory

Once a satisfactory counterfactual scenario is proposed, the user can order the agent to perform the split factual / counterfactual trajectories from that point. The agent would first perform the factual trajectory, reset to the split point, and then perform the proposed counterfactual trajectory. Based on the states observed along each path, the program will determine the minimum sufficient explanation behind the decisions of the agent—including the advantages and disadvantages to its factual trajectory compared to the counterfactual.

The complete list of keybindings for the interactive explainable environment are:

- **spacebar**: toggle the environment pause state
- **r**: rewind the environemnt simulation (only works from a paused state, can be held down)
- **x**: clear most recent component of counterfactual (does nothing if no counterfactual present)
- **c**: clear all of the counterfactual (does nothing if no counterfactual present)
- **left mouse**: append a segment to the counterfactual trajectory from the previous endpoint to the mouse pointer (starts a new counterfactual if it isn't present)

- **e**: append a segment to the counterfactual trajectory for shooting the enemy (starts a new counterfactual if it isn't present)
- **t**: append a segment to the counterfactual trajectory for shooting the target (starts a new counterfactual if it isn't present)

TODO: MAKE THIS KEYBINDING LIST A PROPERLY FORMATTED TABLE

1.3 Installation

All the required packages to use this software can be installed using the pip package manager for Python:

- **PyTorch**: `https://pypi.org/project/torch/`
- **torchrl**: `https://pypi.org/project/torchrl/`
- **PyGame**: `https://pypi.org/project/pygame/`

Installation instructions can be found at the provided links. Assuming you have access to the pip package manager in a terminal shell environment, you can simply run: **pip install PACKAGE_NAME**.

Due to the way the Gymnasium library works, the environment itself must be registered and installed as a local pip package. To do this, navigate to the top level program directory in your terminal and enter the command **pip install -e gym-env**.

1.4 Running

Running the environment is simply a matter of executing the main file, DQN_DogFight.py, in Python. That can be done in the terminal using the command: **python DQN_DogFight.py**. There are many additional parameters that can be passed to the main program that will alter its execution—a full list of them and their explanations can be provided by running: **python DQN_DogFight.py –help**.

TODO: MAKE THE COMMANDS DISPLAY ON A SEPARATE LINE IN A CODE BLOCK, EXPLAIN THE CODE BLOCK FORMAT NEAR THE TOP—PERHAPS IN A NEW PREAMBLE SECTION (I.E. "READING THIS DOCUMENT")

2 Repository Directory Structure

The directory structure of the repository is shown below in Figure 3. Following that is a detailed breakdown of the significance and usage of each subdirectory and file.

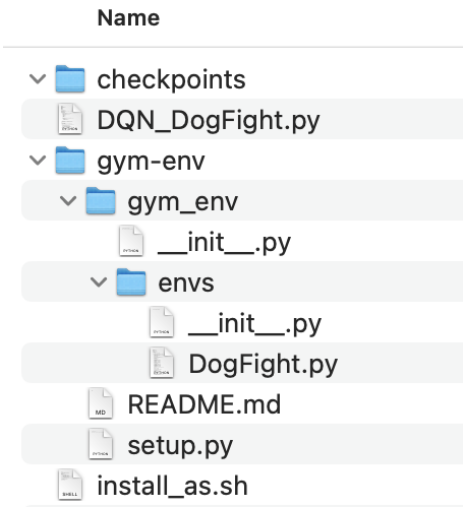

Figure 3: Repository directory structure

2.1 checkpoints

For each training run, a subdirectory is created within this upper-level directory. As the name suggests, these subdirectories will contain the training checkpoints for the policy network, target network, replay memory, episodic reward values, episodic reward plots, and the training parameters for the RL agent during that run. The checkpoints will be stored as:

- **X_policy.pt**: policy network at the end of episode X
- **X_target.pt**: target network at the end of episode X
- **X_memory.pt**: replay memory buffer at the end of episode X
- **X_parameters**: dictionary of training parameters for the agent at the end of episode X (**TODO: these parameters do not change as the episodes go on—we should make it so that only 1 copy of this parameter dict is saved at the start of a new run**)
- **X_plot.png**: graphical plot of the episodic training rewards earned by the RL agent up until the end of episode X (**TODO: we already store the vector of training accuracies up until that point, and these plots can simply be derived from those—should save some space by not saving these images...this would also remove the direct matplotlib / pandas dependencies from the DQN_DogFight.py file**)
- **X_rewards**: Python list containing the episodic training rewards earned by the RL agent up until the end of episode X

2.2 DQN_DogFight.py

This file contains the main program for this project. From execution of this file, you can: train the RL agent (from scratch or from a pervious checkpoint), evaluate the RL agent, and interact with the graphical environment in evaluation mode in order to obtain the explanations behind the agent's actions.

There are four primary functions in this file—they are:

- **main()**: execution of the program begins here. From this main function, we handle the initialization of the environment along with all major components to the DQN learning process (target and policy networks and the prioritized replay buffer). The main training / evaluation loop is in this function. A high level description of the execution from main goes as follows:
  - Set PyTorch computing device (CPU or CUDA / GPU)
  - Parse command line arguments to get main program parameters
  - Initialize the environment, target and policy networks, and the prioritized replay buffer
  - **If loading from a checkpoint**, set the state of the models and memory to those stored in the checkpoint files
  - FOR EACH EPISODE IN TRAINING / EVALUATION:
    - Reset the environment, state rewind buffer, shooting / shooting flag buffer, and factual / counterfactual buffers
    - UNTIL THE EPISODE ENDS:
      - **IF THE USER IS REWINDING** (can only be done in graphical evaluation mode), pop the latest state from the rewind state buffer, and set the environment to that state
      - Handle user pausing / unpausing of the environment (can only be done in graphical evaluation mode)
      - **IF PERFORMING FACTUAL TRAJECTORY** (after being proposed a counterfactual), execute the factual trajectory (simply based on agent's policy)
      - **IF PERFORMING COUNTERFACTUAL TRAJECTORY** (after being proposed a counterfactual), execute the counterfactual trajectory
      - **IF DONE WITH FACTUAL AND COUNTERFACTUAL**, obtain differences in Q values experienced by each path, obtain the scalar "disadvantage" value, and use that to determine and print the minimum sufficient explanation

        - **IF NOT PERFORMING FACTUAL / COUNTERFACTUAL SPLIT AND EVALUATING**, choose next action based on learned policy
        - **IF NOT PERFORMING FACTUAL / COUNTERFACTUAL SPLIT AND TRAINING**, choose next action based on epsilon-greedy method
        - Take a step in the environment, get a "transition consisting of: state, action, next state, reward
        - **IF THE STEP INVOLVED A MISSILE RELATED EVENT (i.e. shooting, hitting, or missing)**, push the transition to a separate buffer to handle the delayed rewards at the end of the episode
        - **ELSE IF TRAINING** push the transition to the prioritized replay buffer and call the optimize_model() method
    - Handle assigning the delayed missile rewards to the proper states
    - **IF TRAINING**, push the transition to the replay memory and optimize the policy and target models
- **optimize_model()**: train the online policy network by sampling a batch of transitions from the replay buffer. This uses the same optimization method as standard double DQN training, but to handle the decomposed rewards, we send each reward component through the common network trunk and then through the respective ending branch. This ensures that only the appropriate sections of the network are being updated and optimized based on the specific reward components.
- **update_target()**: perform a soft-update on the target network (i.e. take a small gradual step in updating the target model's weights and parameters to be like the policy network's)
- **parse_arguments()**: parse the command-line arguments passed to the program. Arguments are given default values if not explicitly set by the user. This function will return all the main program parameters that will be used within the main method.

2.3 gym-env

This folder contains the code for the gym environment itself. The structure within this directory is set up such that the gym environment can be locally installed as a Python pip package as is standard for the OpenAI Gym / Gymnasium library.

2.3.1 gym_env

The upper-level ‘gym-env‘ directory specifies the package to be installed, while this specifies the Python module to be installed (to be used in an ‘import‘ statement i.e. ‘import gym_env‘)

2.3.2 __init__.py

Irrelevant to the environment or learning code, this is used for registering the gym environment as a package for pip installation.

2.3.3 envs

This folder contains the custom environment itself. Everything in this directory is used to register the environment with the Gym / Gymnasium APIs

2.3.4 __init__.py

Used to register the environment with Gymnasium

2.3.5 DogFight.py

The code for the custom environment is defined here. Following standard Gymnasium environment guidelines, the custom environment is defined as a Python class. There are four main methods associated with the class / environment:

- **__init__()**: the constructor for the environment that is called in the main ‘DQN_DogFight.py‘ program file via the ‘gym.make()‘ function call. This initializes all the environment constants that are used elsewhere in the code, including the action and observation spaces.
- **step(action)**: takes a single step within the environment simulation. First, handle the agent's actions, move the entities, check for terminating conditions, and return the observations and rewards for the step. Additional information is returned in a dictionary for the decomposed reward components, along with any flags that indicate whether this step had missile / bullet-related events (to be used in the main training code to handle delayed rewards)
- **render()**: used to render the environment to a PyGame window when the user starts the program in human rendering mode. Also contains the code for interacting with the environment (only works when the program was started in evaluation mode), including pausing, rewinding, and proposing counterfactuals.
- **reset()**: reset the environment. Used in the main DQN_DogFight.py code at the start of each episode.

2.3.6 README.md

A README file for displaying on the git repository. Serves the same basic function as this document that you are reading.

2.3.7 setup.py

Additional file used for registering the environment as a pip package.

2.4 install_as.sh

A shell script (can be used on Linux and macOS) to install the gym environment under a new package name. This is used if you want to have multiple versions of the environment (with slight changes, for example) running. They need to be installed under different package names due to certain constraints placed by the Gymnasium library, such as environments being installed as Python pip packages. **DEPRECATED: VERY INFLEXIBLE—THIS FUNCTIONALITY SHOULD EITHER BE ABANDONED OR REPLACED**

3 Suggested Changes & Future Work

- Move away from using the Gym / Gymnasium libraries and APIs. Currently, they introduce some inflexibility that makes the code more complicated, especially with the interoperability between the training code, rendering, and input handling. Moving away from using the API and making the environment independent can be done within a day or so, and would allow for much more flexibility, especially in running multiple versions simultaneously.
- Expand the environment. The environment began in a more complex state, but was scaled down due to issues with training an RL agent within the earlier versions. There are still some issues with training the agent and obtaining desirable behaviors. Once this is resolved, expanding the environment should be very easy. We could include more enemies, more targets, stationary artillery and turrets as additional obstacles.
- Improve rendering. This is of course purely cosmetic. This should be done later as it is of low priority.
- Actually introduce natural language processing (NLP). As it stands right now, the RL explanations are hard-coded string arrays for the advantages and disadvantages. These are not flexible enough to provide detailed and dynamic explanations.
- New, more extensive and gradual reward function. This should improve the learning process itself, and it could provide more granularity for the provided explanations between the factuals and counterfactuals.