Internal Structure of the Reinforcement Learning Codebase

Derin Theiventhiram
https://github.com/DerinHD/Reinforcement_Learning_OpenAI

This chapter guides through the internal structure of the RL codebase. This includes details about the project structure, how to set up the project, the documentation about the environment and model interfaces, and so on. I recommend reading this chapter to understand how to use the codebase, especially if one attempts to extend the project by adding new environments and models or integrating new features.

0.1 Project Structure

Figure 1 illustrates the hierarchical structure of the project folder. It was designed such that there is a clear separation between the different components, such as environment setup, model definition, data storage, core scripts, etc. In the following, each module is presented.

environment

The environment folder contains definitions of the environment where the models operate. It contains custom and OpenAI Gymnasium environments. There is also a base class at the root of the custom folder called `baseEnvironment.py`, which contains an interface for integrating new environments.

model

All custom and Stable-baseline3 implementations of RL algorithms are located in the model folder. The custom model folder is subdivided into two separate folders to differentiate between model-free and model-based methods. The model folder also contains a base class located at `baseModel.py` so that new models can be added to the project.

inverseRL

Methods of the inverse reinforcement learning problem, such as the approximate IRL algorithm, are saved in the inverseRL folder.

core

This folder contains the main logic of the project with two runnable Python files: `main.py` and `visualization.py`. Executing `main.py` allows the user to run the various features of the RL codebase (e.g., train an agent in a specific environment). `visualization.py` contains the code to visualize the results of the trained models. The visualizations can either be universal, such that they can be used for all environments, or only be available for a specific environment.
`settings.py` is used for the configuration of the models and environments. Whenever new models or environments are added to the project, this file needs to be modified so that the components can be loaded properly.

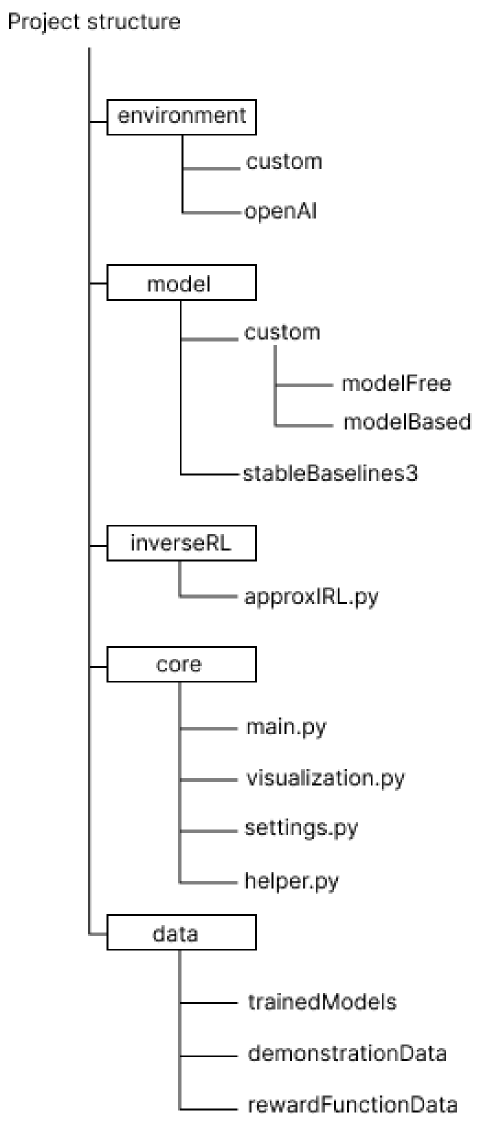

Figure 1: Illustration of the hierarchical file structure of the project. Source: Own illustration.

Finally, the file `helper.py` contains some utility functions to easily handle user interaction with the command line tool.

data

All project-related data is stored in the data folder. It is subdivided into three different components:

- `trainedModels`: contains the result of an RL model trained in a specific environment
  - `rewards.data` saves the total rewards per episode
  - `<environment_name>.environment` contains the parameter settings of the environment where the model operates
  - `<model_name>.model` contains data of the trained model
- `demonstrationData`: contains expert demonstration trajectories performed in a specific environment

  - `<environment_name>.environment` contains the parameter settings of the environment where the trajectories were performed
  - `demonstration.data` saves the trajectories. Since the format of the states and actions differentiate between the environment, I have chosen the package pickle to store the data in the following format: `demonstration.data` is an array with multiple dictionaries indicating the trajectories for each episode. Each dictionary is set up as follows: The key is the current step of the episode. The value is an array where the first element is the current state, the second element is the action that was performed at the current state and the third element states the reward which was received after taking that action.

  Note that the expert demonstration trajectories are generated by the feature "Create demonstration data," where the user plays the game in one or multiple episodes. If there is already external trajectory data available, then it needs to be converted to the format as described above. Also, the environment parameter settings need to be saved manually. Therefore, a separate script needs to be created for the conversion. For serialization of the data, the package pickle was used.

- `rewardFunction`: contains the result of the inverse reinforcement learning problem.
  - `<model_name>.model`: contains the RL model, which was used to imitate the expert
  - `<environment_name>.environment`: contains the parameter settings of the environment where the model operates
  - `<rewardFunction>.data`: contains the data of the reconstructed reward function (e.g., if the approx. IRL method was applied, then the data file contains the weight vector)

Note that for data storage, the Python package pickle was used. One advantage of this package compared to other packages, such as JSON or CSV, is that it allows serializing complex data types like classes, dictionaries, functions, etc., automatically. As an example, the RL model class can be saved directly without decomposing it into basic data types. Additionally, with pickle, Python objects can be loaded and saved efficiently, such that even classes with neural networks can be stored. However, one disadvantage of this package is that it is not human-readable. Therefore, the developer needs to ensure that the data is saved and loaded properly.

0.2 Setup Project

The GitHub version control system was used to maintain the framework code. It allows the developer to view the history of the code development and easily add new features to the codebase. The repository is public and can be found under the following link: https://github.com/DerinHD/Reinforcement_Learning_OpenAI.git. The following instructions describe setting up the project on different operating systems (Windows, OS, Linux).

0. Installation requirements

The RL codebase is a Python framework. Therefore, to run it, a Python release needs to be installed from the following download page https://www.python.org/downloads/. Note that the library Stable-baselines3, which is used to test external RL models, requires Python 3.9+. The maximum compatible version that I tried was 3.12.0. Version 3.13.1 was not compatible with Stable-baselines3.
Additionally, C++ Compiler Support is needed to use Stable-baselines3. For Windows users, I have used the Build tools from Visual Studio https://visualstudio.microsoft.

com/de/downloads/?q=build+tools to install the needed tools, but there could be alternative ways. For OS users, the swig package can be used, which provides a scripting interface for code written in C/C++. Therefore, run the following commands:

```
xcode-select --install
```

```
brew install swig cmake
```

Finally, to install the RL codebase, I recommend downloading git to always have the latest version of the project. Install git under the following link: https://git-scm.com/downloads

1. Create a virtual Python environment.

The idea of having a virtual Python environment is to isolate a project from others to avoid dependency issues with packages. Imagine that there is another project that needs the newest Python version, but as mentioned above, Stable-baselines3 may not be compatible with that version. Therefore, if we run both projects on the same system, it could lead to conflicts. So, using a virtual environment keeps the environment clean and reusable.
A virtual environment can be created by running the following commands on the terminal.

1. Install the package virtualvenv:

   ```
   pip install virtualenv
   ```

2. Run the following command to create a virtual environment folder. I recommend creating a root folder where all virtual environments are saved.

   ```
   python<version> -m venv <virtual-environment-name>
   ```

3. Run the following command to activate the virtual environment

   ```
   source <virtual-environment-name>/bin/activate
   ```

   The command above works for users with the operating systems Linux and OS. For Windows users, run the following command:

   ```
   .\<virtual-environment-name>\Scripts\activate
   ```

The virtual environment can be deactivated with the command `deactivate`. For more details about virtual environments, I recommend reading the Python documentation https://docs.python.org/3/library/venv.html.

2. Initialize project

After all necessary dependencies were installed, the RL codebase can now be downloaded from the GitHub repository. If git was installed in the previous steps, then the project can be cloned with the command

```
git clone  https://github.com/DerinHD/Reinforcement\_Learning\_OpenAI.git
```

or if SSH is used, then run the command

```
git clone git@github.com:DerinHD/Reinforcement\_Learning\_OpenAI.git.
```

Note that cloning with SSH requires an SSH key. A documentation of generating such a key can be found here. When the download finishes, go to the root of the project and run the following command to install all necessary packages:

```
pip install -e.
```

3. Run the code

The project is now ready, and the code can be run by executing the files `main.py` or `visualization.py`, which are located in the core folder. E.g., run the `main.py` file as follows:

```
python main.py
```

Via the terminal, the user will now be guided to run different features of the codebase.

0.3 Base Class Implementation

This section provides a clear understanding of how the components, the environment, and the RL model are implemented. To have a common structure between custom and external implementations, the design of the Python classes is mainly based on the ones defined in OpenAI Gymnasium and Stable-baselines3. The main goal is to provide a clean interface to easily integrate new models and environments into the RL codebase.

0.3.1 Class Diagram

The class diagram shown in Figure 2 illustrates the structure and relationship within model and environment classes. All model and environment classes inherit from the abstract classes BaseModel and BaseEnvironment to have a common interface. Unlike programming languages as C++, where abstract classes are realized by having pure virtual functions, Python has no direct definition for an abstract class. However, this functionality can be achieved by using the module Abstract Base Class (ABC). By inheriting from the ABC class, abstract methods can be implemented by decorating the methods with @abstractmethods. In addition to that, the base class of the environment inherits from `gymnasium.Env`, the environment class definition of the OpenAI Gymnasium library. In the following, a detailed overview of the attributes and methods of the two classes is shown.

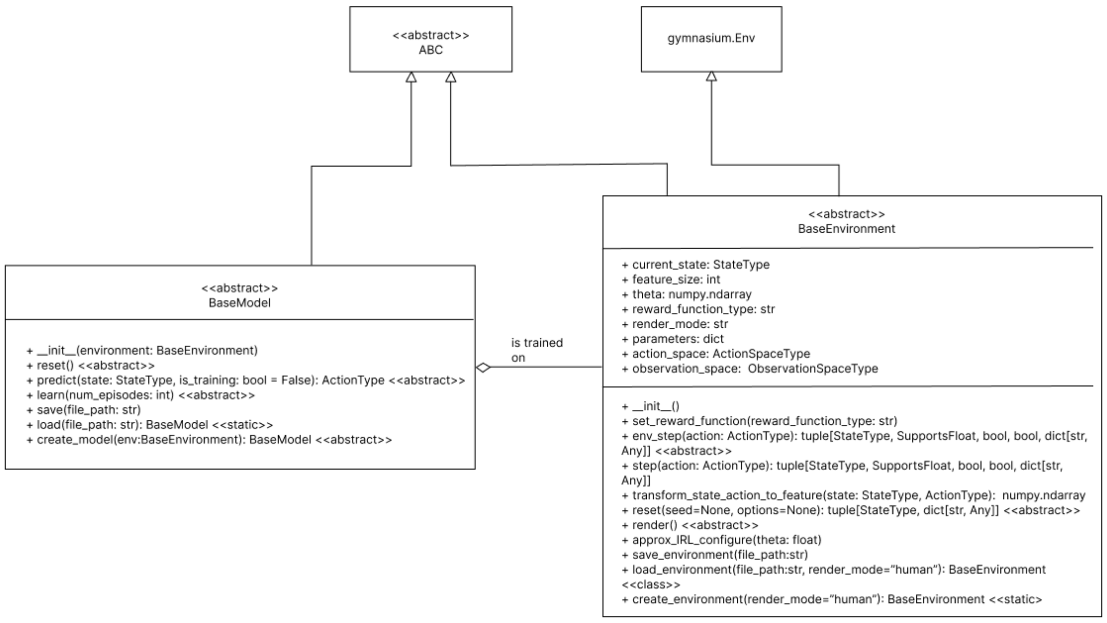
<<abstract>>
ABC
gymnasium.Env
<<abstract>>
BaseModel
+ __init__(environment: BaseEnvironment)
+ reset() <<abstract>>
+ predict(state: StateType, is_training: bool = False): ActionType <<abstract>>
+ learn(num_episodes: int) <<abstract>>
+ save(file_path: str)
+ load(file_path: str): BaseModel <<static>>
+ create_model(env:BaseEnvironment): BaseModel <<abstract>>
is trained on
<<abstract>>
BaseEnvironment
+ current_state: StateType
+ feature_size: int
+ theta: numpy.ndarray
+ reward_function_type: str
+ render_mode: str
+ parameters: dict
+ action_space: ActionSpaceType
+ observation_space: ObservationSpaceType
+ __init__()
+ set_reward_function(reward_function_type: str)
+ env_step(action: ActionType): tuple[StateType, SupportsFloat, bool, bool, dict[str, Any]] <<abstract>>
+ step(action: ActionType): tuple[StateType, SupportsFloat, bool, bool, dict[str, Any]]
+ transform_state_action_to_feature(state: StateType, ActionType): numpy.ndarray
+ reset(seed=None, options=None): tuple[StateType, dict[str, Any]] <<abstract>>
+ render() <<abstract>>
+ approx_IRL_configure(theta: float)
+ save_environment(file_path:str)
+ load_environment(file_path:str, render_mode="human"): BaseEnvironment <<class>>
+ create_environment(render_mode="human"): BaseEnvironment <<static>

Figure 2: Illustration of the structure and relationship within the two base classes for model and environment presented in a class diagram. Source: Own illustration.

BaseModel

The BaseModel class serves as an interface for any RL model in this project. It inherits from the ABC class and therefore can not be instantiated directly. Whenever a new model is integrated into the system, it needs to inherit from this base class. BaseModel expects an environment as a member instance, indicating that a model is trained in an environment.

Attributes:

- `env (baseEnvironment)`: Environment where the model operates

Methods:

- `__init__(environment: BaseEnvironment): void`: Initialize BaseModel class
  Parameters:
  - `environment (BaseEnvironment)`: Environment where the model operates
- `reset(): <<abstract>>`: Resets the model (e.g., for a Q-Learning model, Q-values are reset to initial values). This method is abstract. Therefore, it needs to be overwritten by the child class.
- `predict(state: StateType, is_training: bool=False): ActionType <<abstract>>`: Given a state, the method predicts the next action.
  Parameters:
  - `state (StateType)`: Current state. Type of state can be discrete, multi-discrete, etc.
  - `is_training (bool=False)`: Boolean which states if the model is currently trained or not. This method is abstract. Therefore, it needs to be overwritten by the child class.

  Returns:
  - `action (ActionType)`: Next Action. Type of action can be discrete, multi-discrete, etc.
- `learn(num_episodes: int): <<abstract>>`: Train model for a specific number of episodes. This method is abstract. Therefore, it needs to be overwritten by the child class.
  Parameters:
  - `num_episodes (int)`: Total number of episodes where the model is trained.
- `save(file_path: str)`: Saves the model. The base class already contains an implementation for the data storage using the package pickle. However, if this package does not work to save model data, this method needs to be overwritten by the child class.
  Parameters:
  - `file_path (str)`: Path to file where model data will be stored
- `load(file_path:str): BaseModel <<static>>`: Loads the model. The base class already contains an implementation for loading the data using the package pickle. However, if this package does not work to load the model data, this method needs to be overwritten by the child class. This method is static.
  Parameters:
  - `file_path (str)`: Path to file where the data will be loaded from

  Returns:

  - `model (BaseModel)`: model

- `create_model(env: BaseEnvironment): BaseModel <<static>>`: Creates a model via terminal. This method is static but needs to be overwritten by the child class to configure how the model is created.
  Parameters:
  - `env (BaseEnvironment)`: Environment where the model operates

  Returns:
  - `model (BaseModel)`: model

BaseEnvironment

This abstract class serves as an interface for defining reinforcement learning environments based on the OpenAI Gymnasium library. It inherits from the ABC class and the Env class from the gymnasium package. Whenever a new environment is integrated into the system, it needs to inherit from the base class.

Attributes:

- `current_state (StateType)`: Current State of the environment
- `feature_size (int)`: Size of the feature vector $\phi(s, a)$
- `theta (numpy.ndarray)`: Weight vector w which is used to compute the reward for a (state, action) pair $R(s, a) = w^T \phi(s, a)$
- `reward_function_type (str)`: Name of the reward function. At the current moment, there are two options:
  - "Default": Use standard reward function from the environment
  - "Approx_IRL": Use reconstructed reward function generated by the approximated IRL algorithm
- `render_mode (str)`: Render mode (e.g., "human", "rgb_array", etc.)
- `parameters (dict)`: A dictionary that contains the parameters of the environment
- `action_space (ActionSpaceType)`: Action space (e.g., gym.spaces.Discrete)
- `observation_space (ObservationSpaceType)`: Observation space (e.g., gym.spaces.Discrete)

Methods:

- `__init__()` : Initialize environment
- `set_reward_function(reward_function_type: str)`: Set reward function
  Parameters:
  - `reward_function_type (str)`: Type of reward function
- `reset(seed=None, options=None): tuple[StateType, dict[str, Any]]<<abstract>>`: Reset the environment. The environment returns to its initial state. This method is abstract. Therefore, it needs to be overwritten by the child class.
  Parameters:
  - `seed (int)`: Define a seed that can be used to control initialization of the environment
  - `options (dict[str, Any])`: Optionally, add more info on how the environment was reset

Returns:

  - `state (StateType)`: initial state
  - `info (dict[str, Any])`: info about how the environment was reset

- `env_step(action: ActionType):tuple[StateType, SupportsFloat, bool, bool dict[str, Any]] <<abstract>>`: Perform an action in the environment. This method is abstract. Therefore, it needs to be overwritten by the child class.
  Parameters:

  - `action (ActionType)`: Next Action|

  Returns:

  - `next_state (StateType)`: Next state received from the environment
  - `reward (SupportsFloat)`: Reward received from the environment
  - `termination (bool)`: Boolean indicating the termination of the episode
  - `truncation (bool)`: Boolean indicating the truncation of the episode (e.g., maximum number of steps possible)
  - `info (dict[str, Any)`: Additional info about taking a step.

- `step(action: ActionType):tuple[StateType, SupportsFloat, bool, bool dict[str, Any]] <<abstract>>`: Perform an action in the environment. It calls the method `env_step()` that needs to be implemented in the inherited class.
  Parameters:

  - `action (ActionType)`: Next Action

  Returns:

  - `next_state (StateType)`: Next state received from the environment
  - `reward (SupportsFloat)`: Reward received from the environment
  - `termination (bool)`: Boolean indicating the termination of the episode
  - `truncation (bool)`: Boolean indicating the truncation of the episode (e.g., maximum number of steps possible)
  - `info (dict[str, Any])`: Additional info about taking a step

- `render(): void <<abstract>>`: render the environment via GUI or terminal output

- `transform_state_action_to_feature(state: StateType, action: ActionType) <<abstract>>`: Transforms a (state, action) pair into a feature vector. This function can be used to reduce the state complexity of the environment or for inverse Reinforcement learning.
  Parameters:

  - `state (StateType)`: State
  - `action (ActionType)`: Action

  Returns:

  - `feature_vector (numpy.ndarray)`: feature vector

- `approx_IRL_configure(theta: numpy.ndarray)`: Configure the weight vector of the approx. IRL method
  Parameters:

  - `theta (numpy.ndarray)`: Weight vector w which is used to compute the reward for a (state, action) pair $R(s, a) = w^T \phi(s, a)$

- `save_environment(file_path: str)`: Saves the parameters of the environment in a file. Note that the whole environment class is not serializable
  Parameters:

  - `file_path (str)` Path to file where environment data will be stored

- `load_environment(file_path:str, render_mode=’’human’’): baseEnvironment <<class>>:` Loads the environment
  Parameters:
  - `file_path (str)`: Path to file where the environment data will be loaded from
  - `render_mode (str)`: Render mode

  Returns:
  - `environment (BaseEnvironment)`: environment

- `create_environment(render_mode="human"): <<static>>` : Creates an environment via terminal. This method is static but needs to be overwritten by the child class to configure how the environment is created.
  Returns:
  - `environment (BaseEnvironment)`: Environment
  - `render_mode (str)`: Render mode

0.4 Integrate New Models and Environments into the Codebase

This section describes the workflow of adding new models and environments to the codebase.

Create a new environment

1. Create a new environment class

Go to the folder environment and create a new Python file `<environment_name>.py`. Open the file and create a new class that inherits from the ABC and `gymnasium.Environment` class. Example:

```
import gymnasium as gym
from abc import ABC, abstractmethod

class Gridworld(ABC, gym.Env):
    ...
```

Ensure that all abstract methods are overwritten by the child class. Note that if an environment from the OpenAI Gymnasium library is added to the system, then the process is very similar, except that the child class also inherits from the original environment class. As an example, view the implementation of the FrozenLake class in the folder environment/openAI.
Note that for the method `create_environment`, the `helper.py` file can be used to configure the parameters of the environment. Again, view the implementation of the FrozenLake class to see how the method can be created.

2. Configure settings.py

Open core/settings.py and configure the file as follows. At the top of the file, the new environment class needs to be imported. Example:

```
    from environment.custom.GridWorld import GridWorld
```

Afterwards, define a name for the environment that will be shown to the user when running the project. Example:

```
    gridWorld = "GridWorld"
```

Locate to the dictionary `list_of_environments` and add a new entry for the new environment. The key is the name of the environment, and the value is an array defining the type of observation and action space. At the current moment, the project contains the types "Discrete", "Multi-Discrete", "Box" and "Multi-Binary". Example:

```
    list_of_environments = {
        ...
        # first element:=type of observation space, second element:=type of action space
        gridWorld : ["Discrete", "Discrete"]

    }
```

Locate to the function `create_environment` and add a new entry for the new environment class. Example:

```
def create_environment(environment_name, folder_path):
    env = None
    parameters = None

    if environment_name == "..." :
        ...
    elif environment_name == gridWorld:
        env = Gridworld.create_environment(render_mode="rgb_array")
    ...
    return env, parameters
```

Locate the function `load_environment` and add a new entry for the new environment class. Example:

```
def load_environment(folder_path, render_mode ="human"):
    ...
    env = None
    path = f"{folder_path}/{content[idx]}"

    if environment_name == "...":
        ...
    elif environment_name == gridWorld:
        env, parameters = GridWorld.load(path, render_mode)

    return env, parameters, environment_name
```

In case that the new environment class has a discrete action space, the function `get_action_names` needs to be modified so that a new entry for the environment class is added. This is important for the feature "Create demonstration data", where the user can play the game and needs to know which actions are available. Example:

```
def get_action_names(env_name):
    ...
    if env_name ==  ...:
        ...
    elif env_name == gridWorld:
        # ensure that the new environment class contains the method get_action_names
        return GridWorld.get_action_names()
    ...
```

The environment is now ready and can be tested by running the core/main.py file and running the features "Train an agent" or "Create demonstration data".

Create a new model

1. Create a new model class

Go to the folder model/custom and create a new Python file `<model_name>.py`. Open the file and create a new class that inherits from the ABC class. Example:

```
class QLearning(ABC):
    ...
```

Ensure that all abstract methods are overwritten by the child class. As an example, view the implementation of the DQN class in the folder model/stableBaselines3/dqn.py.
Note that for the method `create_model`, the `helper.py` file can be used to configure the parameters of the model. Again, view the implementation of the DQN class to see how the method can be created.

2. Configure settings.py

Open core/settings.py and configure the file as follows. At the top of the file, the new model class needs to be imported. Example:

```
from models.custom.modelfree_RL.tabular.QLearning import QLearning
```

Afterwards, define a name for the model that will be shown to the user when running the project. Example:

```
qLearning = "QLearning"
```

Locate to the dictionaries `models_compatibility_observation_space` and `models_compatibility_action_space` and add a new entry for the model class if the types of the spaces of the environment are compatible with the new model. Example:

```
models_compatibility_observation_space = {
    "Box": [...],
    "Discrete": [qLearning],
    "MultiDiscrete": [...],
    "MultiBinary": [...],
}
models_compatibility_action_space = {
    "Box": [...],
    "Discrete": [qLearning],
    "MultiDiscrete": [...],
    "MultiBinary": [...],
}
```

In the example above, the QLearning class is only compatible with an environment that has a discrete observation and action space. Locate the function `create_model` and add a new entry for the new model class. Example:

```
def create_model(model_name, env):
    model = None
    if model_name == ...:
        ...
    elif model_name == qLearning:
        model = QLearning.create_model(env)
    ...
    return model
```

Locate the function `load_model` and add a new entry for the new model class. Example:

```
def load_model(folder_name):
    ...
    if model_name == ...:
        ...
    elif model_name == qLearning:
        model = QLearning.load(path)
    ...
    return model
```

The model is now ready and can be tested by running the core/main.py file and running the feature "Train an agent".

0.5 Workflow of the Features

In the following, the workflow of each feature that is contained in the RL codebase will be described.

0.5.1 Train an Agent

This feature deals with the control problem of finding an optimal policy. A model of the user's choice is trained on a specific environment for a given number of episodes. Figure 3 illustrates the workflow of the feature. At the beginning, the user is asked to input the name of the folder where the model will be saved. Afterward, the user has to choose an environment. The system will now run the static method `create_environment` of the corresponding environment class to instantiate the environment by configuring its parameters via the terminal. These parameters will be saved in a separate file called `<environment_name>.environment`. The system now searches for models that are compatible with the environment. More precisely, all models that work with the observation and action type of the environment are now shown and the user is asked to choose a model. Similar to the creation of the environment, the system will now instantiate the model and the user can configure its parameters via the terminal. Finally, the user has to specify how many episodes the model will be trained on and if the model should be recorded at some fixed episodes. The record of an episode can be saved as an .mp4 file or as a text file, depending on how the render method was implemented. Additionally, the total number of rewards per reward will be monitored automatically. The results (parameters of the environment, trained model and rewards per episode) are saved in the folder trainedModels.

0.5.2 Create Demonstration Data

To deal with problems such as maximum likelihood estimation or inverse reinforcement learning, demonstration data are needed. This feature can be used to create trajectories by playing one or multiple episodes in the environment via terminal inputs (see figure 4 for illustration). The user will first be asked to input the name of the folder where the demonstration data is saved. Afterward, a list of environments that have a discrete action space will be shown to the user. The feature can not be run on environments with other types of action spaces. After the user has chosen an environment and configured its parameters, the environment will be instantiated and its parameters will be stored in a separate file. The user can now play the game which is rendered on the GUI via the Python package pygame or on the terminal as text. At the terminal, the user can see which keyboard keys correspond to which action. Whenever an episode is terminated or truncated, the user can decide whether to play another episode or to finish it. When the user wants to play another episode, the environment will be reset to its initial state. The results will be saved in the folder data/demonstrationData.

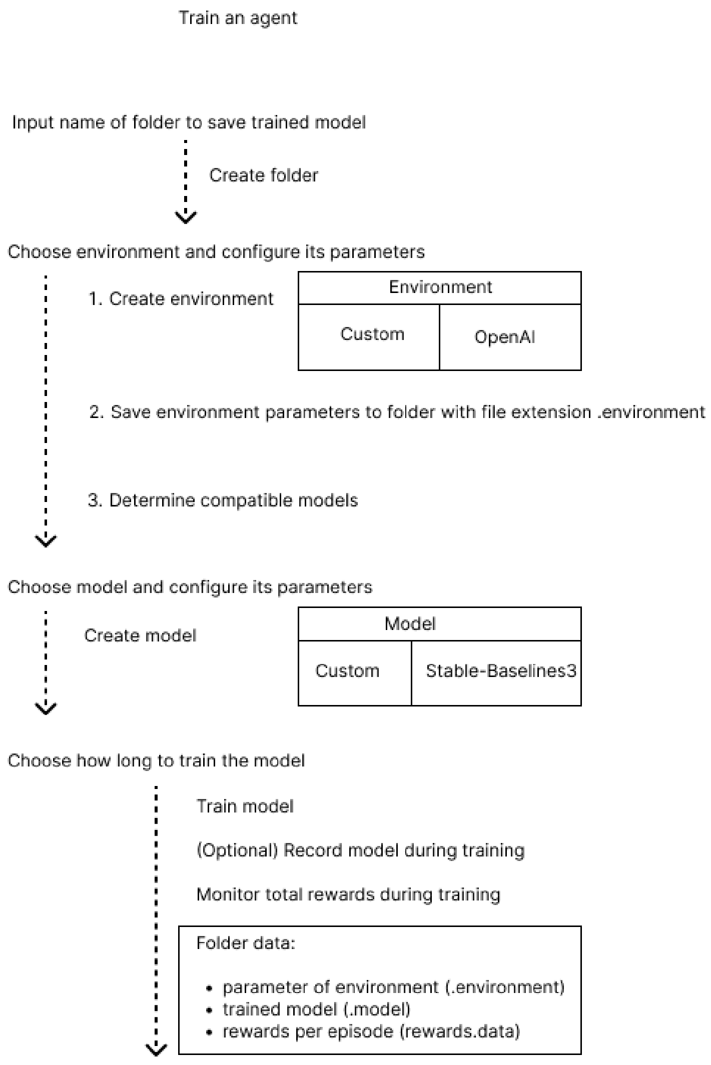
Train an agent
Input name of folder to save trained model
Create folder
Choose environment and configure its parameters
1. Create environment
Environment
Custom
OpenAI
2. Save environment parameters to folder with file extension .environment
3. Determine compatible models
Choose model and configure its parameters
Create model
Model
Custom
Stable-Baselines3
Choose how long to train the model
Train model
(Optional) Record model during training
Monitor total rewards during training
Folder data:
• parameter of environment (.environment)
• trained model (.model)
• rewards per episode (rewards.data)

Figure 3: Illustration of the workflow of the feature "Train an agent". Source: Own illustration.

0.5.3 Perform Inverse Reinforcement Learning

This feature describes how to perform inverse reinforcement learning to reconstruct the reward function. The workflow is illustrated in Figure 5. The user is first asked to input the name of the folder that contains the demonstration data and the corresponding environment parameters. The system will then load the demonstration data and the environment. Afterward, it will determine compatible models that can be used to sample the trajectories by learning a policy based on the current reward function. The user now needs to choose which model to use and configure its parameters. Finally, the user has to configure the parameters of the inverse RL algorithm (e.g., number of training steps). When the training is finished, the results will be stored in the folder data/rewardFunctionData.

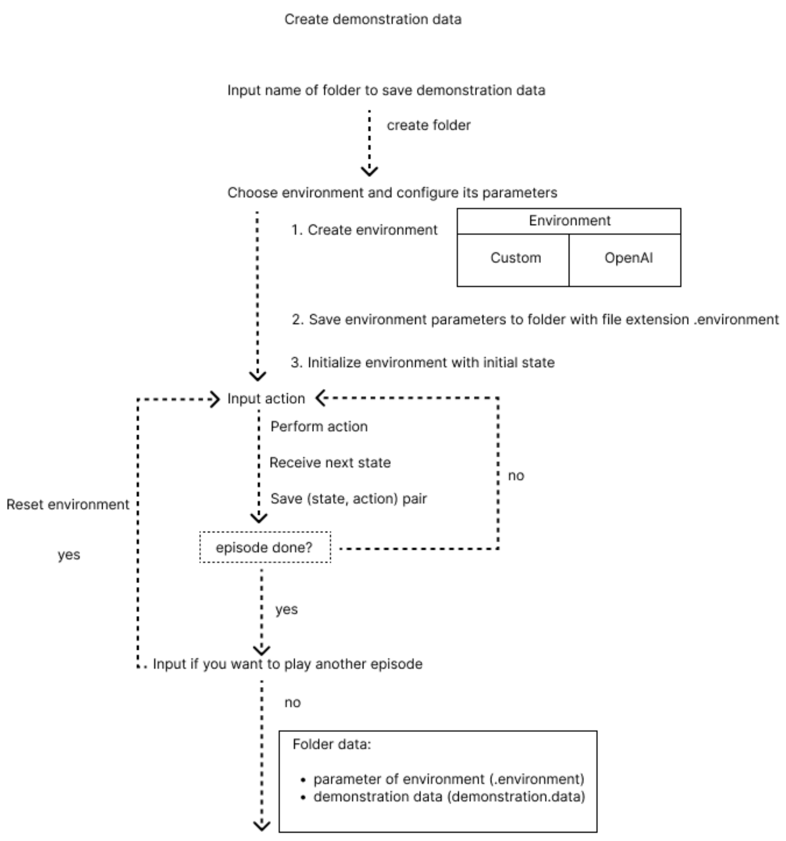
Create demonstration data
Input name of folder to save demonstration data
create folder
Choose environment and configure its parameters
1. Create environment
Environment
Custom
OpenAI
2. Save environment parameters to folder with file extension .environment
3. Initialize environment with initial state
Input action
Perform action
Receive next state
Save (state, action) pair
no
Reset environment
yes
episode done?
yes
Input if you want to play another episode
no
Folder data:
parameter of environment (.environment)
demonstration data (demonstration.data)

Figure 4: Illustration of the workflow of the feature "Create demonstration data". Source: Own illustration.

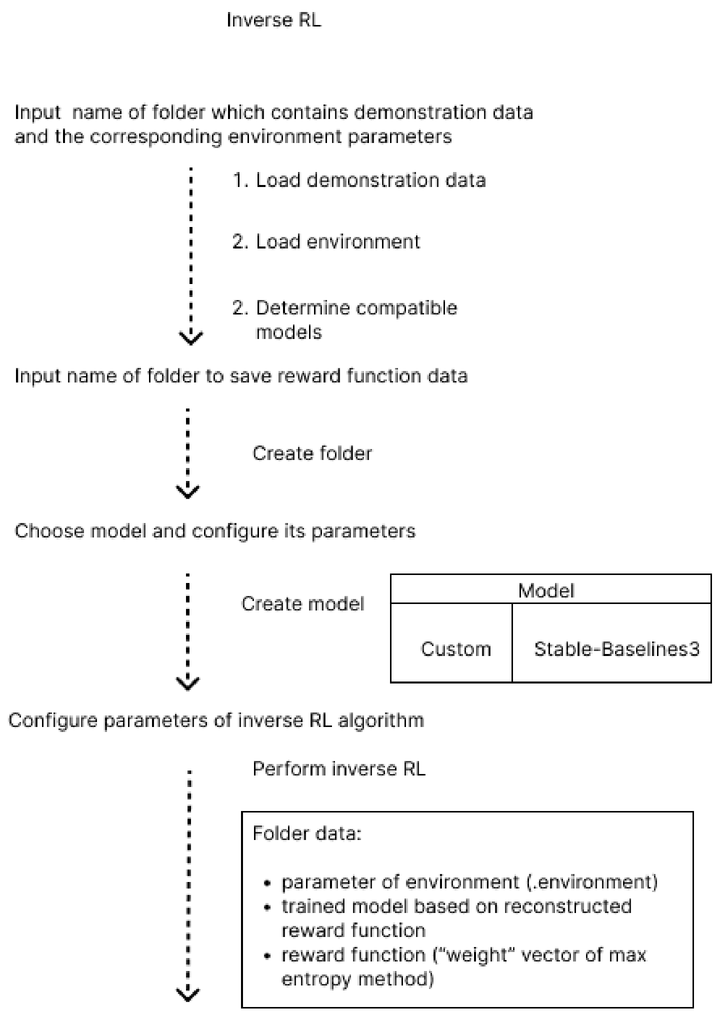
Inverse RL
Input name of folder which contains demonstration data and the corresponding environment parameters
1. Load demonstration data
2. Load environment
2. Determine compatible models
Input name of folder to save reward function data
Create folder
Choose model and configure its parameters
Create model
Model
Custom
Stable-Baselines3
Configure parameters of inverse RL algorithm
Perform inverse RL
Folder data:
• parameter of environment (.environment)
• trained model based on reconstructed reward function
• reward function ("weight" vector of max entropy method)

Figure 5: Illustration of the workflow of the feature "Perform inverse reinforcement learning". Source: Own illustration