Graphical User Interface for OpenAI Gym Project Documentation

Özgür Kara, Mehul Sinha

August 2022

Contents

1 Project Description

The project aims to develop a GUI for facilitating the experimentation of Reinforcement Learning for the users. Particularly, researchers can be able to test and implement their ideas and algorithms about reinforcement learning with the GUI easily even though they are not proficient in coding.

1.1 Features

- PyQt5 based open source Graphical User Interface for visually testing the RL agents
- Dragging and dropping the components allow users to create flows easily
- Flows can be saved (in json format), loaded and new graphs can be opened using toolbar options
- Flows can also be created in DHGWorkflow, saved in GraphML format and loaded in VisualRLComposer as a new graph
- Users can create multiple Workflow graphs by opening a new scene tab
- During testing, the relevant values such as rewards, states and actions are updated in a real-time manner
- There are seven built-in RL environments and reward functions from OpenAI Gym and six RL agents that are imported from stable-baselines3 library
- Program allows users to both perform training and testing. Also, users can save their trained models as well as load their pretrained models according to their preferences
- Users can design custom policies for their RL models using the custom policy network designer
- The program allows users to train or test models with multi-environment experimentation in parallel
- The program allows user to keep track of model training via Tensorboard
- Users are able to integrate their custom environments and reward functions to the program by following the procedure in the documentation
- Detailed documentation and demo videos are provided in the GitHub page

1.2 Installation

In order to install the program, run the following codes:

```
$ git clone https://github.com/NISYSLAB/VisualRLComposer
$ python -m venv env
$ source env/bin/activate
$ pip install -r /path/to/requirements.txt
```

1.3 Usage

After the installation is done, you can run the program by running the following command.

```
$ python main.py
```

2 Interface Components

The components that constitute the interface are introduced in this section. In the **interface.py** module, all the components are initialized in the **initUI** function as follows:

Code 1: Initializing interface components. (interface.py)

```
from window_widget import RLComposerWindow
from tensorboard_widget import Tensorboard
from plot_widget import TestingWidgetPlot, TrainingWidgetPlot
from custom_network_widget import NetConfigWidget
from treeview_widget import FunctionTree
from plot_button import PlotButton

    def initUI(self):
        self.window_widget = RLComposerWindow(self)
        self.tensorboard = Tensorboard()
        self.testing_reward_widget = TestingWidgetPlot(name="Reward")
        self.testing_state_widget = TestingWidgetPlot(name="State")
        self.testing_action_widget = TestingWidgetPlot(name="Action")
        self.training_reward_widget = TrainingWidgetPlot(name="Reward")
        self.training_action_widget = TrainingWidgetPlot(name="Action")
        self.plot_button_widgets = PlotButton(self.testing_reward_widget, self.testing_action_widget,
                    self.testing_state_widget, self.training_reward_widget, self.training_action_widget)
        self.netconf = NetConfigWidget(self, '')
        self.plot_tab = QTabWidget(self)
        self.plot_tab.addTab(self.tensorboard, 'Tensorboard')
        self.plot_tab.addTab(self.plot_button_widgets, "Plots")
        self.plot_tab.addTab(self.netconf, "Custom Network")
        self.tree = FunctionTree(self.window_widget)
        self.img_view = QLabel(self)
        self.pauseButton = QPushButton("Pause/Continue", self)
        self.saveModelButton = QPushButton("Save Model", self)
        self.createButton = QPushButton("Create Instance", self)
        self.trainButton = QPushButton("Train Instance", self)
        self.testButton = QPushButton("Test Instance", self)
        self.closeButton = QPushButton("Close Instance", self)
```

Most of the components (except for the buttons) are imported from their corresponding modules where each of them will be explained in the next sections in detail. The layout (e.g. size, position, etc.) of these components are arranged and organized in *createLayout* function:

Code 2: Layout organization. (interface.py)

```
    def createLayout(self):
        layout = QGridLayout(self)
        layout.setRowStretch(0, 6)
        layout.setRowStretch(1, 4)
        layout.setRowStretch(2, 1)
        layout.setColumnStretch(0, 70)
        layout.setColumnStretch(1, 5)
        layout.setColumnStretch(2, 5)
        layout.setColumnStretch(3, 5)
        layout.setColumnStretch(4, 5)
        layout.setColumnStretch(5, 5)
        layout.setColumnStretch(6, 5)

        layout.addWidget(self.window_widget, 0, 0)
        layout.addWidget(self.plot_tab, 1, 0, 2, 1)
        layout.addWidget(self.tree, 0, 1, 1, 6)
        layout.addWidget(self.img_view, 1, 1, 1, 6)
```

```
layout.addWidget(self.createButton, 2, 1)
layout.addWidget(self.trainButton, 2, 2)
layout.addWidget(self.saveModelButton, 2,3)
layout.addWidget(self.testButton, 2, 4)
layout.addWidget(self.pauseButton, 2, 5)
layout.addWidget(self.closeButton, 2, 6)
```

The following steps should be done when one wants to add a new component to the interface:

1. Create a class for your widget and code it according to your preferences
2. Import the class in the *interface.py* and create the object in *initUI()* function.
3. Then add your widget to the layout in *createLayout()* function accordingly.
4. layout.setRowStretch() and layout.setColumnStretch() functions control the size of each column and row as well as layout.addWidget(widget, row_number, col_number, row_span, col_span) determines the position of the widget that is going to be add to the layout.

In figure 1, the interface of the GUI is shown and the corresponding positions are entitled with red texts.

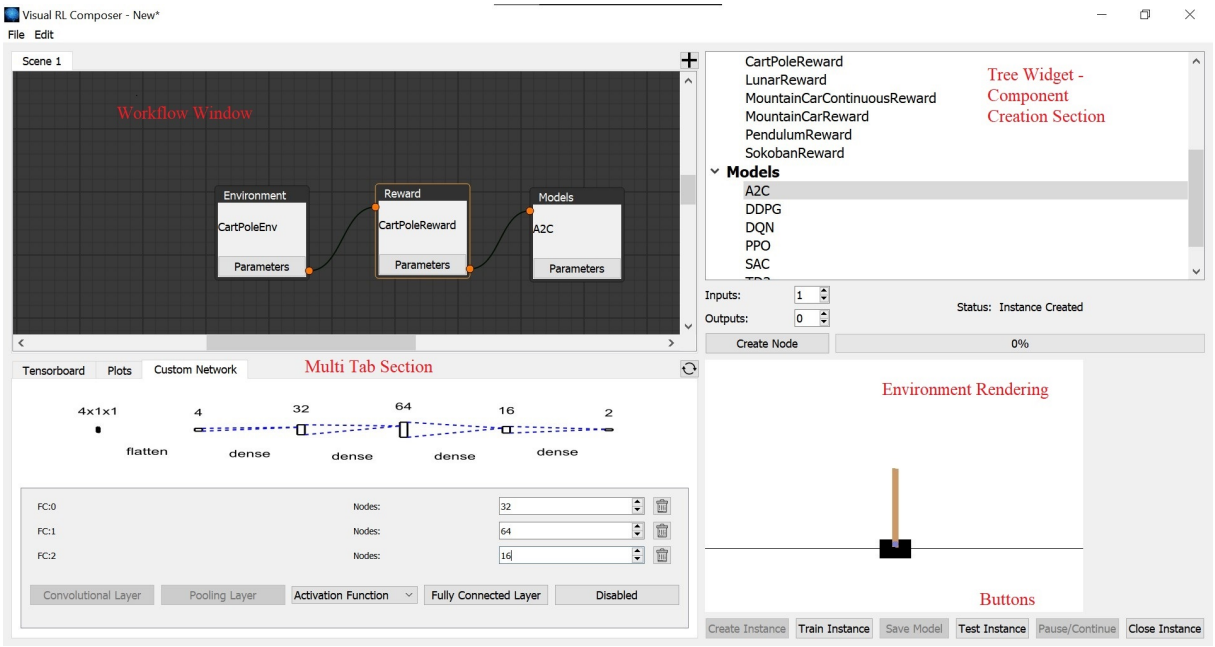


Figure 1: Interface of GUI

2.1 Workflow Window

In this part, we introduce the sub components in the workflow window where users can form their graphs visually by dragging and dropping the graphical blocks.

2.1.1 Scene

Scene is the page of the workflow window where there is a background (black grid background in Figure 1) and abstract functions in `scene.py` that are responsible for adding and removing the node and edge objects. Scene object also has an attribute called "self.grScene" which is imported from the class in the module namely `graphics/graphics_scene.py` and it is inherited from "QGraphicsScene" class of PyQt5.

Code 3: Scene initialization. (scene.py)

```
from graphics_scene import QDMGraphicsScene
    def initUI(self):
        self.grScene = QDMGraphicsScene(self)
        self.grScene.setGrScene(self.width, self.height)
```

Once you want to add a new object to the workflow and you want it to be visually seen on the workflow window, do the following steps:

1. Create your own QGraphicsItem object. (Like the classes in `graphics/graphics_edge.py` or `graphics/graphics_node.py`
2. Create a list in the Scene class (`scene.py`) and add it's corresponding function for addition and removing as in Code 4

Code 4: Scene class (scene.py)

```
from edge import Edge
from node import Node
class Scene(Serialize):
    def __init__(self):
        super().__init__()
        self.nodes = []
        self.edges = []

    def addNode(self, node):
        self.nodes.append(node)

    def addEdge(self, edge):
        self.edges.append(edge)

    def removeNode(self, node):
        self.nodes.remove(node)

    def removeEdge(self, edge):
        self.edges.remove(edge)
```

3. Then pass the scene object to your new class and use "QGraphicsScene.addItem(QGraphicsItem)" function to visually see your new item on the board. Code 5 shows an example for node object.

Here we first passed the scene object to our Node class, then we assigned the QGraphicsItem as "self.grNode = QDMGraphicsNode(self)" where we design the graphical and visual features of the Node object at QDMGraphicsNode class in `graphics/graphics_node.py`. After, we use "self.scene.addNode(self)" to add our abstract node object to the scene object and used "self.scene.grScene.addItem(self.grNode)" line to add our QGraphicsItem to QGraphicsScene which draws it on the scene when the program runs.

Code 5: Node class (node.py)

```
from graphics_node import QDMGraphicsNode

class Node():
    def __init__(self, scene, title="Undefined Node", inputs=[], outputs=[],
                 nodeType=None, model_name=None):
        super().__init__()
        self.scene = scene
        self.grNode = QDMGraphicsNode(self)
        self.scene.addNode(self)
        self.scene.grScene.addItem(self.grNode)
```

2.1.2 Scene Tabs

Scene tabs are a functionality that can be used to make, train and test multiple RL models by creating model graphs on different Scene tabs. On creating a new scene tab, Scene object and QDMGraphicsView object are appended to a scene list and view list respectively. On changing tabs, scene and view list pass the corresponding objects to the class scene and view variables.

Code 6: RLComposerWindow class (window_widget.py)

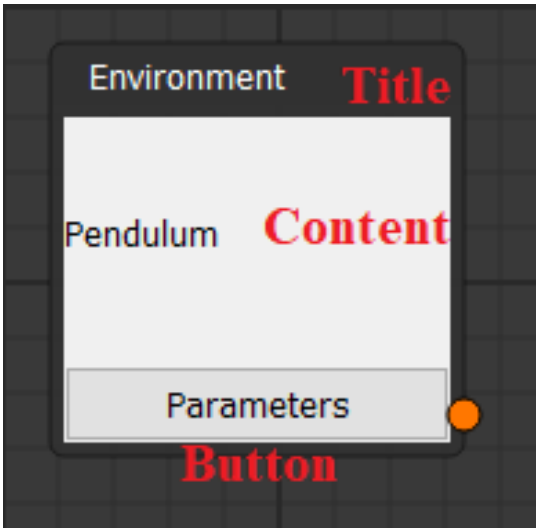
```
from scene import Scene
from graphics_view import QDMGraphicsView

class RLComposerWindow(QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.scene_list = []
        self.view_list = []
        self.view = None
        self.scene = None

    def add_page(self):
        self.scene_list.append(Scene())
        self.view_list.append(QDMGraphicsView(self.scene_list[-1].grScene, self))
        self.view_list[-1].setScene(self.scene_list[-1].grScene)

        self.scene_tab.addTab(self.view_list[-1], f"Scene {self.scene_tab.count()+1}")

    def onTabChange(self, i):
        self.scene = self.scene_list[i]
        self.view = self.view_list[i]
```

2.1.3 Nodes

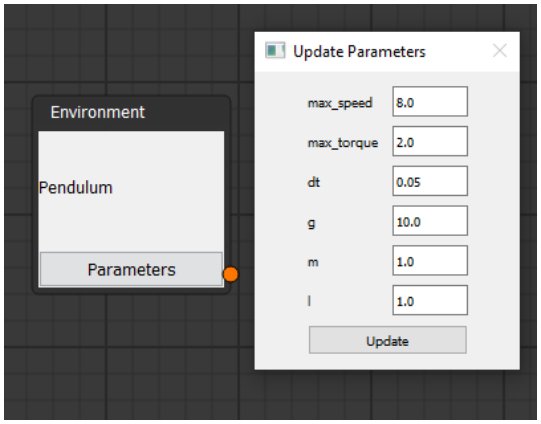
Nodes are the fundamental pieces that comprise of four units, namely: Title, content, updating button and sockets. Each node can be created from the tree widget component (positioned at right-upper most side) visually by assigning both the number of input and output sockets and the functionality of the node.

The node class can be found in `node.py`. Nodes are initialized and added to the scene after they are created. The class (QDMGraphicsNode) that is responsible for the graphical features (e.g. color, size, shape, etc.) of the node can be found in the `graphics/graphics_node` directory and be edited in order to design the node according to your preferences.

- Node title shows the type of the node.
- The content at the middle of the node shows the detailed information. The class that creates the content part of the node can be found in `node_content.py`. This module also involves the "ParameterWindow" class that is responsible for updating the parameters and the
- Once you click the button named "Parameters", it opens a new sub-window (see Figure 2-b) where you can update the relevant parameters of the node.
- The editable lines and text-boxes (see Figure 2-b, max_speed, max_torque, g, ...) are automatically created according to the type of the node. Like, for Pendulum environment, they are 6 different parameters and once you change the values and click the update button, it will change the default parameters and creates the environment with the new parameters.



(a) A node block



(b) A node and parameter update window

Figure 2: Node

2.1.4 Sockets

Sockets are the another important part of the workflow window where users can connect the sockets with edges by pressing the left mouse click on one of the output sockets and releasing it on one of the input sockets. Each socket has the "edge" attribute where it store the edge object that is connected through the current socket. The functions and graphical features (QGraphicsItem) can be found in `sockett.py` and `graphics/graphics_socket.py` respectively.

2.1.5 Edges

Edges are used to connect the sockets with each other visually. Once the user clicks an output socket and releases the mouse left button at another input socket, an edge will be created visually. Edge class can be found in `edge.py` and it's graphical representation is in `graphics/graphics_edge.py` module.

The assignments of the objects in scene after an edge is created are done in `graphics_view.py` module. See the mouseButton Press and Release functions if you want to change events that will happen after user clicks to the window.

2.2 Component Creation

In order to ease the control in the code, we have three wrapper classes for environments, rewards and RL models which can be found in `rl/...` directory. Each node has a "wrapper" and "param" attributes which are assigned at the initialization part of the Node class. "self.wrapper" attribute is assigned based on the node type.

Code 7: Node type initialization (node.py)

```
from rl.env_wrapper import EnvWrapper
from rl.reward_wrapper import RewardWrapper
from rl.model_wrapper import ModelWrapper

class Node():
    def __init__(self, scene, title="Undefined Node", inputs=[], outputs=[], nodeType=None, model_name=Non

        if self.title == "Environment":
            self.wrapper = EnvWrapper(self.nodeType)
        elif self.title == "Reward":
            self.wrapper = RewardWrapper(self.nodeType)
        elif self.title == "Models":
            self.wrapper = ModelWrapper(self.nodeType)
            if model_name is not None:
                self.wrapper.loadModel(model_name)
        self.param = self.wrapper.param
```

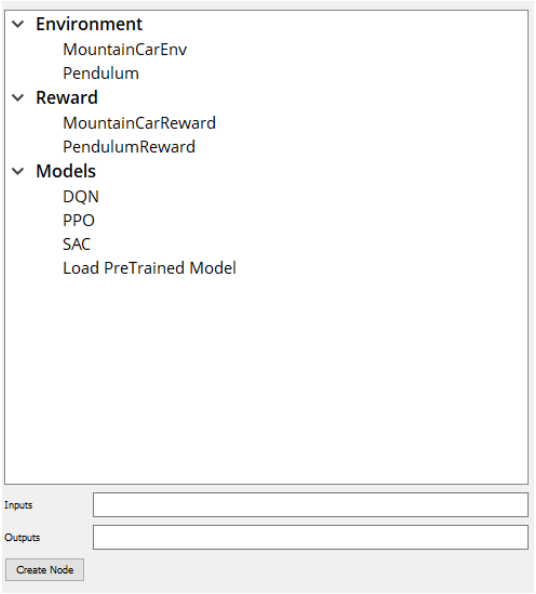


Figure 3: Component Creation

Nodes can be created using the treewidget component as shown in Figure 3. Once a node is to be created, user must select the type of the node by left clicking on one of the lines, then determine and type the number of sockets at both input and output sides before clicking the "Create Node" button. "onButtonClick()" function in `treeview_widget.py` will generate a node on the scene object by calling "generateNode()" function immediately after the button is clicked.

The codes can be found in `treeview_widget.py` module. Here, we initialize the name of the components automatically in "FunctionTree" class which is a type of QWidget.

Code 8: Treeview widget initialization(treeview_widget.py)

```
import rl.components.environments as envs
import rl.components.rewards as rewards
import rl.components.models as models

class FunctionTree(QWidget):
    def __init__(self, window_widget):
        super().__init__()
        self.layout = QGridLayout()
        self.window_widget = window_widget
        self.treeView = QTreeView()
        self.treeView.setHeaderHidden(True)

        self.env_names = envs.return_classes()
        self.reward_names = rewards.return_classes()
        self.model_names = models.return_classes()
        self.initTreeModel()

    def initTreeModel(self):

        self.treeModel = QStandardItemModel()
        self.rootNode = self.treeModel.invisibleRootItem()

        self.envs = StandardItem('Environment', 12, set_bold=True)
        for env_name in self.env_names:
            self.envs.appendRow(self.createItem(env_name))

        self.rewards = StandardItem('Reward', 12, set_bold=True)
        for rew_name in self.reward_names:
            self.rewards.appendRow(self.createItem(rew_name))

        self.models = StandardItem('Models', 12, set_bold=True)
        for model in self.model_names:
            self.models.appendRow(self.createItem(model))
        self.models.appendRow(self.createItem("Load PreTrained Model"))
```

Each environment, reward function and RL model is written in their corresponding modules that can be found in `rl/components/...py`. In Code 7, see that there is a function called "return_classes" which returns all of the class names in a module as a list.

Code 9: return_classes() function in `rl/components/...py`

```
def return_classes():
    current_module = sys.modules[__name__]
    class_names = []
    for key in dir(current_module):
        if isinstance(getattr(current_module, key), type):
            class_names.append(key)
    return class_names
```

Hence, if one wants to add it's own custom component, you only need to add your class to its corresponding module according to the type of your component. After you add it, it will be automatically seen by the treeview widget and visually added to the GUI.

2.2.1 Environments

In this program, we are utilizing OpenAI gym environments where your custom environments need to satisfy the same functions as in OpenAI gym. The following differences must be added to your custom environment.

- Add a "self.reward_fn = reward" attribute at the initialization function of your environment, pass "reward=None" to your custom environment and use your reward function in the "step" function
- Add a self.parameter_box attribute, which contains a list of strings of environment attributes, at the initialization function of your environment. This attribute will show those parameters in parameter box generated by a node
- Make a set_render(self, n_envs) function in the environment class to change the render window size depending on the number of environments created by the user

An example of Pendulum environment is shown below:

Code 10: Pendulum environment example (`rl/components/environment.py`)

```
class Pendulum(gym.Env):

    def __init__(self, reward=None):
        #Other attributes are defined in a same way done in OpenAI Gym
        self.viewer_x, self.viewer_y = 500, 500
        self.reward_fn = reward
        self.parameter_box = ['max_speed', 'max_torque', 'dt', 'g', 'm', 'l']

    def set_render(self, n_envs):
        #function scales down the height and width of render window depending on n\_envs
        if n_envs > 1 and n_envs <= 4:
            self.viewer_x = int(self.viewer_x / 2)
            self.viewer_y = int(self.viewer_y / 2)
        elif n_envs > 4 and n_envs <= 9:
            self.viewer_x = int(self.viewer_x / 3)
            self.viewer_y = int(self.viewer_y / 3)

    def step(self, u):

        #Step calculations are done
        #Only difference is the usage of self.reward_fn
        costs = self.reward_fn.calculateReward(th, thdot, u)

        #Same codes
        newthdot = thdot + (-3 * g / (2 * l) * np.sin(th + np.pi) + 3. / (m * l ** 2) * u) * dt
        newth = th + newthdot * dt
        newthdot = np.clip(newthdot, -self.max_speed, self.max_speed)
        self.state = np.array([newth, newthdot])
        return self._get_obs(), -costs, False, {}
```

where the corresponding reward function is assigned according to the workflow that is created visually on the GUI.

2.2.2 Reward Functions

Reward functions are also automatically shown in the treewidget by looking the class names in `rl/components/rewards.py` module. In order to add your custom reward function, you need to create a class and add a function named "calculateReward()". You can write your own reward definition and modelling inside of that class. An example for Pendulum Reward is shown below:

Code 11: Pendulum reward example (`rl/components/rewards.py`)

```
class PendulumReward():
    def __init__(self):
        pass

    def calculateReward(self, th, thdot, u):
        def angle_normalize(x):
            return (((x + np.pi) % (2 * np.pi)) - np.pi)

        return angle_normalize(th) ** 2 + .1 * thdot ** 2 + .001 * (u ** 2)
```

2.2.3 Models

Models are imported from "stable_baselines3" library and the wrapper function can be found in `rl/model_wrapper.py` module which is responsible for updating the parameters of the selected model as well as loading any model from given directory. Note that if you save a model after training, it's name must start with the model's class name and must be splitted with "_" letter. For example, if you save a "SAC" model, please use the following form: "SAC_....". Because the program loads the model using "stable baseline3" modelClass.load(directory) function.

Code 12: Loading Model (`rl/model_wrapper.py`)

```
    def loadModel(self, dir):
        model_str = dir.split('/')[-1].split('_')[0]
        self.model = getattr(sys.modules[__name__], model_str).load(dir)
```

Apart from these, GUI enables the usage of the models that are available in stable-baselines3 library.

2.3 Multi-Tab Section

2.3.1 Tensorboard Integration

If you want to investigate the training analysis, GUI has a Tensorboard option that is embedded to the multi-tab section. During training, a tensorboard instance is spun up, and all relevant features are shown in the Tensorboard section on the left-bottom part of the GUI. You can add and change the Tensorboard callbacks according to your preferences by editing the code in `rl/components/tensorboard_callbacks.py`

2.3.2 Real Time Plotting

In order to plot the reward, action and state data during both training and testing, we utilize the components that are available in matplotlib.pyplot library. A "FigureCanvas" is created and updated while training or testing the RL agent and it is then embedded to a QTabWidget tab. Each environment has their data seperated on different tabs. On interacting with buttons, program opens a new window to display the data in plots. The source codes are in `plot_widget.py` where one can update and add new functions to plot new stuff according to his/her preferences. Each window displaying a different type of data will run on another thread provided by QThreadPool.

2.3.3 Custom Policy Designer

To change the policy of an agent, the GUI provides a network designer where the user can design custom networks for policies. When the current environment has an observation space that is not an image, the user can configure the networks by adding Dense layers(Fully connected layers) and changing the activation function of the network. When the current environment has an observation space that is an image, the user can configure the networks by adding Convolutional layers, Pooling layers, Dense layers(Fully connected layers) and changing the activation function of the network. If an activation function is not selected, ReLU is added as a default Non-Linear Activation function for CNN and Tanh is added as a default Non-Linear Activation function for MLP. The user can also choose whether to use a custom policy or use default policy provided by Stable-Baselines 3.

The designer also provides a live updating visual representation of the policy network. When changes are made to the policy, the program creates a svg representation of the network and saves it to `assets\net.svg`. You can configure the functions to make the svg in `draw_nn.py`

Parameters for layers provided by Custom Policy Designer are:

- Convolutional Layer: User configurable parameters are Filters, Kernal, Stride
- Pooling Layer: User Configurable parametes are Type[MaxPooling or AveragePooling], Kernal, Stride
- Activation Function: User can choose between ReLU, Tanh, Sigmoid, ELU, GLU, Softmin, Softmax
- Fully Connected layer: User can configure number of nodes in the FC layer

2.4 Environment Rendering

Environment rendering is done using the "rendering" functionalities of OpenAI gym library. The only thing that one needs to keep in mind is that this rendering function opens a new "Pyglet" window, that's why we use multi-threading to open two windows simultaneously and hide the "Pyglet" window. In order to embed the RGB picture of the environment, we use the 'mode="rgb_array"' option in order to obtain the frame of the current situation. Then we embed this onto the window using Pyqt5 widgets. Rendering and embedding the picture to the window is done in `interface.py` module. Please see the comments.

3 Experimentation

Users are able to conduct experiments using environments, reward functions and RL models. In order to do an experiment, users must create an environment, reward and model nodes and connect these with each other with edges. A complete flow is shown in the following figure.

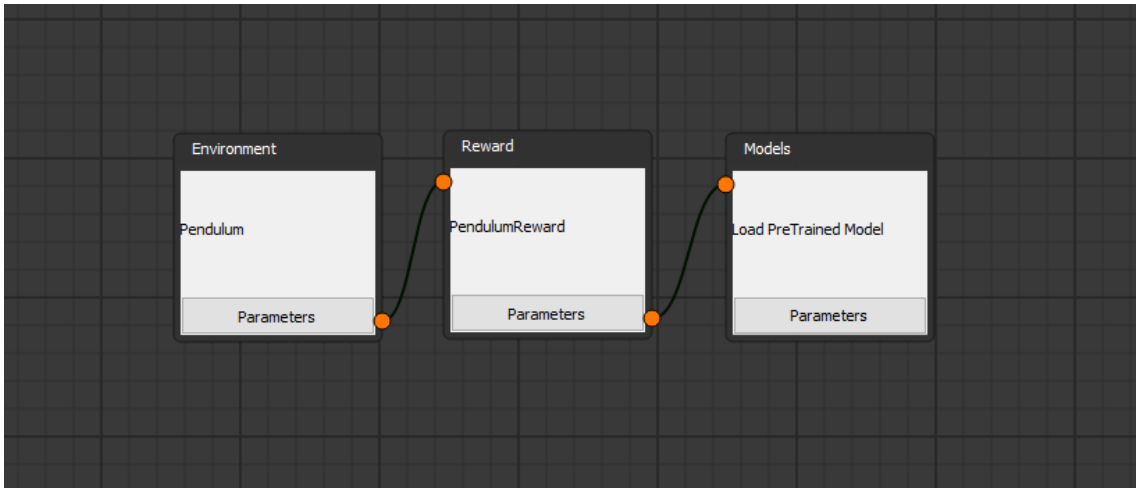


Figure 4: Example Flow

After you complete a flow, you should click Create Instance button to build the flow at the back-end. Then you are able to both train and test your RL model with different type of environments and reward functions. After you are done with the experimentation, you should click "Close Instance" button to delete the current model and you can restart experimentation again. Also, users have the option to save, save as and load the workflows using the toolbar options. Graphs are serialized using "json" formatting.

You can find the button functions in `interface.py` file by following which button is connected to which function.

3.1 Creating Instance

After a flow is completed, you should click the create instance button in order to build the instance object. You can find the codes for Instance class in `rl/components/instance.py` where there exist functions namely stepping, resetting, training, saving and building instance. An instance is a complete flow that is constituted by a RL model, an environment and a reward function.

After instance is created, you can either train your RL agent by clicking the "Train" button or test your model both visually and by looking the relevant plots. In order to do so, first create your instance, then click the test button and enjoy with your testing.

3.2 Loading and Saving Trained Model

You are able to load models that are pretrained by either on GUI or in the same type of "stable-baselines3" pretrained models. You can select the "Load PreTrained Model" from the node creation part and after choose your pretrained model by clicking it's directory. Then it will create a "Model" node. After you complete the flow with pretrained model, you can test the model visually by clicking Test Instance.