VisualRLComposer - NeuroWeaver Integration Documentation

Özgür Kara, Mehul Sinha

October 2022

Contents

1 Project Description

The NeuroWeaver Project is framework to utilize multiple domain-specific accelerators for end-to-end cross-domain applications. Neuroweaver provides a simple and lightweight programming interface, called Component and Flow programming model, which allows application programmers to specify various components of their program to be targeted for acceleration. Finally, Neuroweaver is equipped with a runtime system which creates a Component and Flow Graph, schedules the components and handles the data transfers between components.

While, the Components which are to be targeted for acceleration have to be specified by application programmers, The integration of VisualRLComposer with Neuroweaver aims to eliminate the necessity of writing code to make a Component and Flow Graph and instead use the GUI to do the same graphically.

1.1 Features

- PyQt5 based open source Graphical User Interface for visually creating Component and Flow programs
- 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
- Flows between Components can be given a label by right-clicking on the flow to cycle through list of same outputs, inputs and states between the 2 components.
- The GUI has a Runtime Settings to change runtime iterations, max message size and to handle sample data transfers between components
- The GUI allows user to keep track of model training via Tensorboard by using tf events files.
- 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
$ cd VisualRLComposer
$ git checkout neuroweaver
$ python -m venv env
$ source env/bin/activate
$ pip install -r /path/to/requirements.txt
```

After the installation is done, run the following commands to copy rlcomposer to the neuroweaver project

```
$ cp /path_to_VisualRLComposer/main.py /path_to_neuroweaver/ppo/
$ cp -r /path_to_VisualRLComposer/rlcomposer/ /path_to_neuroweaver/
```

1.3 Usage

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

```
$ cd /path_to_neuroweaver/
$ source /path_to_VisualRLComposer/env/bin/activate
$ python ppo/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 treeview_widget import FunctionTree
from .rl.instance import Instance
from .runtime_settings import RuntimeSettingsWindow

    def initUI(self):
        self.window_widget = RLComposerWindow(self)
        self.tensorboard = Tensorboard()
        self.plot_tab = QTabWidget(self)
        self.plot_tab.addTab(self.tensorboard, 'Tensorboard')
        self.tree = FunctionTree(self.window_widget)
        self.runtime_param = {"Iterations": 2, "Use Futures": True, "Max MSG Size": 3000000, "Samples": []
        self.settings_window = RuntimeSettingsWindow(self)
        self.createButton = QPushButton("Create Instance", self)
        self.runtimeSettingsButton = QPushButton("Runtime Settings", self)
        self.startRuntimeButton = QPushButton("Start Runtime", self)
        self.closeButton = QPushButton("Close Instance", self)
```

Most of the UI 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, 8)
        layout.setColumnStretch(2, 8)
        layout.setColumnStretch(3, 8)
        layout.setColumnStretch(4, 8)

        layout.addWidget(self.window_widget, 0, 0)
        layout.addWidget(self.plot_tab, 1, 0, 2, 1)
        layout.addWidget(self.tree, 0, 1, 2, 4)
        layout.addWidget(self.createButton, 2, 1)
        layout.addWidget(self.runtimeSettingsButton, 2, 2)
        layout.addWidget(self.startRuntimeButton, 2,3)
        layout.addWidget(self.closeButton, 2, 4)
```

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.

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 black grid background 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)

```python
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 and test multiple Component and Flow graphs by creating different 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.

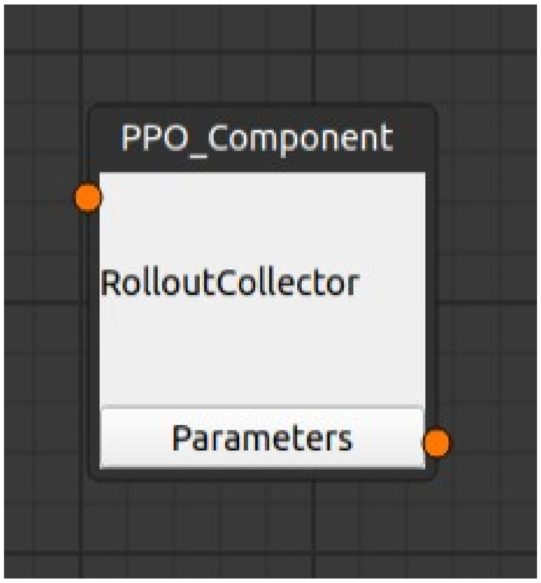
Code 6: RLComposerWindow class (window_widget.py)

```python
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]
```

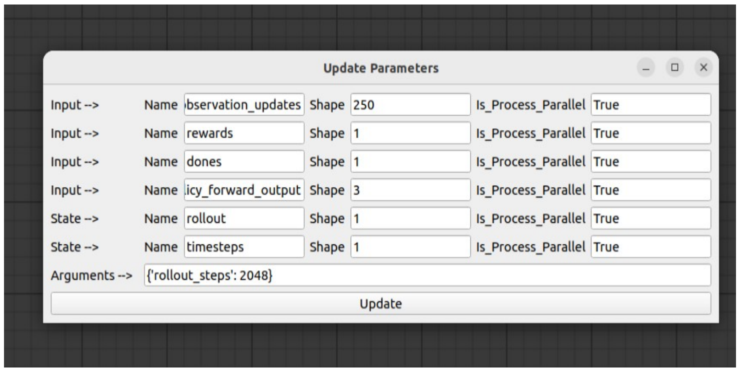
2.1.3 Nodes

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.



(a) A node block



Input -->	Name	›bservation_updates	Shape	250	Is_Process_Parallel	True
Input -->	Name	rewards	Shape	1	Is_Process_Parallel	True
Input -->	Name	dones	Shape	1	Is_Process_Parallel	True
Input -->	Name	icy_forward_output	Shape	3	Is_Process_Parallel	True
State -->	Name	rollout	Shape	1	Is_Process_Parallel	True
State -->	Name	timesteps	Shape	1	Is_Process_Parallel	True
Arguments -->	{'rollout_steps': 2048}					

(b) A parameter update window

Figure 1: Node

- 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 1-b) where you can update the relevant parameters of the node.
- The editable lines and text-boxes (see Figure 1-b) are automatically created according to the type of the node. Like, for RolloutCollector, there are 4 input parameters, 2 state parameters and 1 argument parameter and once you change the values and click the update button, it will change the default parameters and creates the component with the new parameters.

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. Every Edge between 2 Components can be assigned a label by right-clicking on that edge to cycle through all the possible labels for that particular edge between 2 Components. Edge class can be found in `edge.py` and it's graphical representation is in `graphics/graphics_edge.py` module.

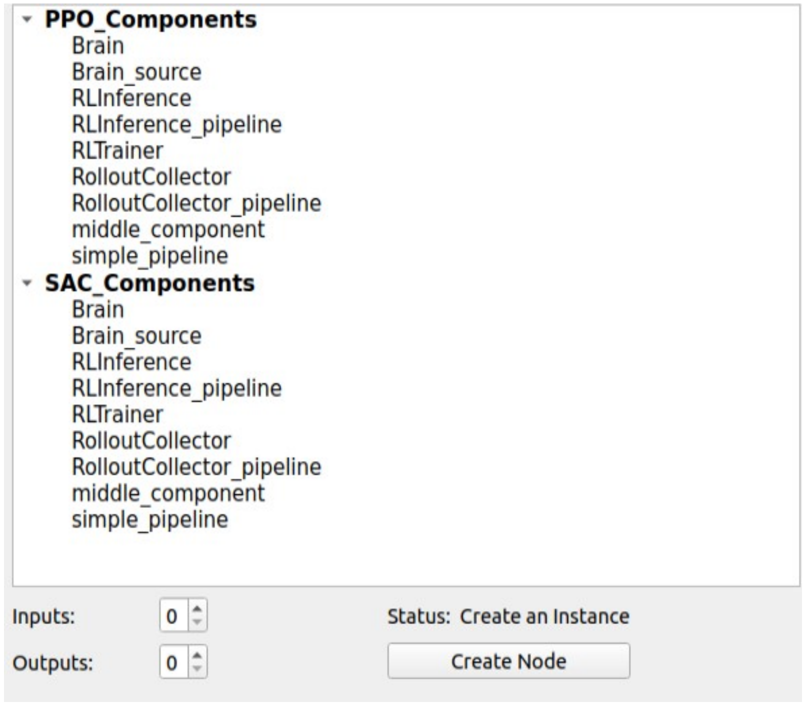
The assignments of the objects in scene after an edge is created are done in `graphics_view.py` module. See the mouseButton Press and Release functions if you want to change events that will happen after user clicks to the window.

2.2 Component Creation

In order to make components in Component and Flow programming model, we have a Component wrapper class which can be found in the `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)

```python
from .rl.component_wrapper import ComponentWrapper

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

        if self.title == "PPO_Components":
            self.wrapper = ComponentWrapper(self.nodeType, self.title)
            self.param = self.wrapper.param
        elif self.title == "SAC_Components":
            self.wrapper = ComponentWrapper(self.nodeType, self.title)
            self.param = self.wrapper.param
```



Inputs: 0

Outputs: 0

Status: Create an Instance

Figure 2: Component Creation

Nodes can be created using the treewidget component as shown in Figure 2. Once a Component 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 ppo.ppo_components as ppo_components
import ppo.sac_components as sac_components

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.ppo_component_names = get_classes(ppo_components)
        self.sac_component_names = get_classes(ppo_components)
        self.initTreeModel()

    def initTreeModel(self):

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

        self.ppo_components = StandardItem('PPO_Components', 12, set_bold=True)
        for component_name in self.ppo_component_names:
            self.ppo_components.appendRow(self.createItem(component_name))

        self.sac_components = StandardItem('SAC_Components', 12, set_bold=True)
        for component_name in self.sac_component_names:
            self.sac_components.appendRow(self.createItem(component_name))
```

In Code 8, see that there is a function called ”get_classes” which returns all of the class names in a module as a list. This is used to get the Components pre-defined by the application programmer into the GUI.

Code 9: get_classes() function in `treeview_widget.py`

```
def get_classes(current_module):
    class_names = []
    to_be_dropped = ['Component', 'PPO', 'uidarray']
    for key in dir(current_module):
        if isinstance(getattr(current_module, key), type) and key not in to_be_dropped:
            class_names.append(key)
    return class_names
```

Hence, if one wants to add their 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.

2.3 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 runtime, a tensorboard instance is spun up, and all relevant features are shown in the Tensorboard section on the left-bottom part of the GUI. Any data during runtime can be logged to a TF events, which can then be read by the Tensorboard viewer. You can configure the Tensorboard viewer according to your preferences by editing the code in `tensorboard_widget.py`

3 Experimentation

Users are able to conduct experiments using the GUI to make CnF Graphs and following the CnF programming model. In order to do an experiment, users must create Component nodes and connect these with each other with edges. Then, the user must cycle through possible labels for the edge by right-clicking on that edge. A complete CnF model graph example is shown in the following figure.

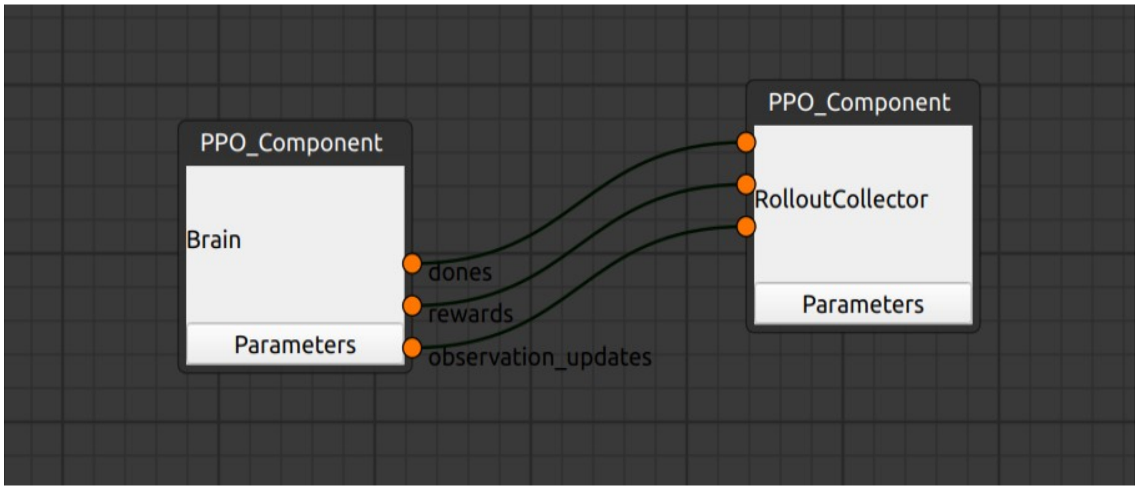
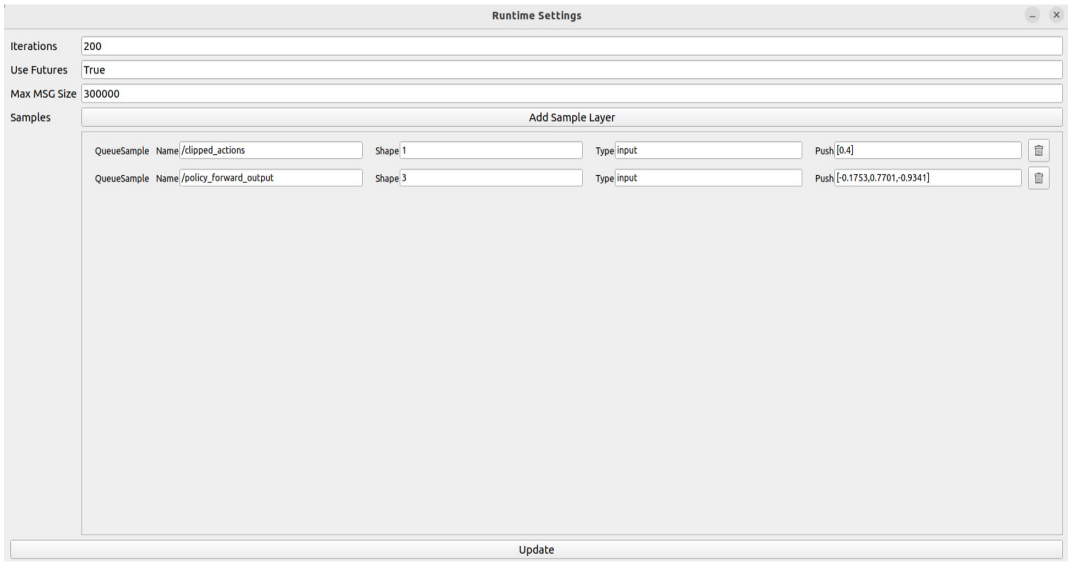


Figure 3: Example CnF Graph (test_brain_and_rollout)

After you complete a Graph, you should click Create Instance button to build the flow at the back-end. Then you are able to change the run-time settings as per your liking by changing iterations or by giving sample data to transfer between components. After you have configured the run-time settings, you should click ”Start Runtime” button to create a Runtime object and start the run-time for the current graph. After you are done with the experimentation, you should click ”Close Instance” button to delete the current instance 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 Runtime Settings



Iterations	200
Use Futures	True
Max MSG Size	300000
Samples	

QueueSample	Name /clipped_actions	Shape 1	Type input	Push [0.4]
QueueSample	Name /policy_forward_output	Shape 3	Type input	Push [-0.1753,0.7701,-0.9341]

Figure 4: Example Runtime Settings (test_brain_and_rollout)

After a CnF Graph 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/instance.py` where there exist functions to create graphs, make flows and start runtime. An instance is a complete flow that is constituted by the CnF programming model.

After instance is created, you can specify runtime settings by clicking "Runtime Settings" button to set runtime data like iterations, max message size. You can also add PosixMSGQueue by adding layers and giving information like Queue name, shape, type and the type of data to be pushed. You can push a list of integers or floats, or a boolean value or you can give `[oscillator-v0]` as push data to push inital observation of gym environment 'oscillator-v0'