URNAI: A Multi-Game Toolkit for Experimenting Deep Reinforcement Learning Algorithms

Marco A. S. Araújo
Metropolis Digital Institute
Federal University of Rio Grande do Norte
Natal, Brazil
marcocspc@hotmail.com

Luiz P. C. Alves
Computing and Automation Department
Federal University of Rio Grande do Norte
Natal, Brazil
luiz_paulo.09@hotmail.com

Charles A. G. Madeira
Metropolis Digital Institute
Federal University of Rio Grande do Norte
Natal, Brazil
charles@imd.ufrn.br

Marcos M. Nóbrega
Metropolis Digital Institute
Federal University of Rio Grande do Norte
Natal, Brazil
marcos.mnobrega@gmail.com

***Abstract*—In the last decade, several game environments have been popularized as testbeds for experimenting reinforcement learning algorithms, an area of research that has shown great potential for artificial intelligence based solutions. These game environments range from the simplest ones like CartPole to the most complex ones such as StarCraft II. However, in order to experiment an algorithm in each of these environments, researchers need to prepare all the settings for each one, a task that is very time consuming since it entails integrating the game environment to their software and treating the game environment variables. So, this paper introduces URNAI, a new multi-game toolkit that enables researchers to easily experiment with deep reinforcement learning algorithms in several game environments. To do this, URNAI implements layers that integrate existing reinforcement learning libraries and existing game environments, simplifying the setup and management of several reinforcement learning components, such as algorithms, state spaces, action spaces, reward functions, and so on. Moreover, URNAI provides a framework prepared for GPU supercomputing, which allows much faster experiment cycles. The first toolkit results are very promising.**

Keywords—game environment, toolkit, deep reinforcement learning, experimentation setup

I. Introduction

Game environments have become very popular over the last decade as testbeds for experimenting Reinforcement Learning (RL) [12][15] and Deep Reinforcement Learning (DRL) [16][14] algorithms, since the amount of labeled training data available for training Artificial Intelligence (AI) models is nearly infinite, low-cost, replicable, and easily obtained at a much higher rate than in real-world experiments. In addition to that, the combinatorial explosion of the problems treated by several games leads to huge state and action spaces, being natural and reasonable candidates to be explored to design innovative AI solutions [7].

In order to contribute in this direction, some international conferences, such as the annual IEEE Conference on Games (CoG) and the annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), hold several Game AI competitions [21] that are a great way to share practical solutions for hard real-world problems simulated by game environments [17][18]. However, in order to experiment an algorithm in each of these game environments, researchers need to prepare all the settings for each one. This task is very time consuming since it entails integrating the game environment to the researcher's software and treating game environment variables in order to set up the system with a particular learning scenario and evaluate the system performance [19].

For this reason, researchers have tried to develop game toolkits in order to make the experimentation task easier [1][2]. As an alternative to these game toolkits, this paper introduces URNAI, a new multi-game toolkit developed to support researchers in the task of setting up their experiments with DRL algorithms. The originality of URNAI is that it supports multiple game environments and multiple DRL libraries by using a layered and modular architecture. This simplifies the setup and management of several DRL components, such as algorithms, state spaces, action spaces, reward functions, and so on.

This paper is structured as follows. First, section II outlines previous work. Next, section III introduces URNAI's layered architecture. Section IV presents the DRL libraries and game environments already integrated into URNAI. Subsequently, section V explains how to set up URNAI in order to do experiments. Next, section VI explains how to start a training session with URNAI. Section VII describes what needs to be done in order to expand URNAI by integrating new DRL libraries or new game environments. Subsequently, section VIII describes a URNAI example of use. Next, section IX shows URNAI performing experiments and presents the results obtained with it. Finally, section X concludes with the contribution of this paper and outlines a roadmap for future work.

II. Related Works

Open source DRL libraries and toolboxes have been recently launched as new options to make training and testing of DRL models easier for researchers, developers, and even non-experts. Giant companies, such as Facebook, Google, Uber and Tencent have invested in the development of these kinds of solutions since the area is growing very fast and it can be seen as a big bet for the coming years.

The main idea is to provide new possibilities for users, so that they can concentrate the efforts on making deep learning architectures rather than data wrangling, by interpreting the prototyping process and streamlining data processing. This comes with further advantages for non-programmers involving a set of command line utilities for training, experimenting models, and gaining predictions. These tools often also provide a programmatic API that enables users to train and deploy a model with only a few lines of code, treating different choices of modeling and algorithms that require different observations and actions.

To support this, OpenAI Gym [2] was one of the first solutions that introduced the concept of Wrapper, which can be stacked and nested to design environments with different observations and actions. This Gym environment is flexible and its wrappers are widely used by the RL community. It implements several simplified environments, in addition to some atari games simulated through the Arcade Learning Environment (ALE) [1]. Nonetheless, Gym only focuses on the generalization of game environments, and leaves DRL implementation, DRL model integration and DRL cycle in the user's hands.

Facebook ELF [10] implements a highly customizable and lightweight real-time strategy (RTS) platform with three game environments (Mini-RTS, Capture the Flag and Tower Defense). The platform allows both game parameter changes and new game additions. The training is deeply and flexibly integrated into the environment, with emphasis on concurrent simulations. One of its strengths is it has a very good performance that allows training in light computers. However, ELF was developed before the release of more modern RTS training environments, such as the StarCraft II Learning Environment (SC2LE) released by Blizzard Entertainment and DeepMind [7], and therefore falls behind when compared to the diversity of game environments available in more recent platforms, such as URNAI.

Arena [11] is a toolkit for multiagent RL solutions that requires customizing observations, rewards and actions for each agent, changing cooperative and competitive interaction [13]. It provides a novel modular design in which different interfaces can be concatenated and combined, extending the OpenAI Gym Wrappers concept to multiagent scenarios such as StarCraft II, Pommerman, ViZDoom, and so on. Arena is one of the existing solutions that comes closest to the solution proposed in this paper. A drawback to Arena, however, is that it has not yet been published. Therefore, the scientific community is not currently able to experiment with this toolkit.

III. URNAI Architecture

URNAI's main goal is to provide a toolkit solution that is able to make implementation and testing of DRL agents easier, as well as out-of-the-box wrappers and tools that fit many different game environments. The general idea of the toolkit is to propose a modular architecture composed of interconnected components that can be easily replaced. This architecture features a layered design connecting high-level external components, such as game environments, to low-level external components, such as DRL libraries.

A. Layers

URNAI's architecture consists of three main layers: Libraries, Core and Environments (see Fig. 1). The Libraries layer and the Environment layer are interfaces for external components, while the Core layer is composed of all the structures needed to control a RL Agent..

URNAI was designed to be as modular as possible. That is why the Core layer has many different generalized structures that exchange data with each other. This gives greater flexibility for researchers who want to quickly iterate through different training setups.

- Layer 1: Libraries

DRL libraries are tools used to code AI models. A model is a memory structure that rules AI reasoning as well as its learning algorithm. This happens because, in most circumstances, a library is used to associate a learning algorithm to a memory structure. As an example, we can highlight Keras and PyTorch. If we build a memory structure, such as a Deep Neural Network (DNN) [20], using such Machine Learning libraries, they will make calls to functions provided by their own learning algorithm. So, we cannot, in general, separate the algorithm from the memory structure. That is why the model is a component inside URNAI's architecture that is designed to encapsulate memory structure and learning algorithm.

In the Model component, the memory can be anything, from relatively simple structures, such as tabular solutions controlled by Tabular Q-learning methods [8], to more complex ones, such as DNN solutions controlled by Deep Q-learning methods [9].

- Layer 2: Core

The Core of URNAI was derived from the workflow of a typical DRL scenario. In a typical use case, an Agent selects actions by using a DRL Model that performs in the Environment. The DRL cycle procedure is performed as follows: (1) from an initial State, the Agent uses the DRL Model to select an Action and carries it out in the Environment; (2) an interpreter receives the result of the action taken, leading the Environment to the next State and producing a positive or negative Reward; (3) the Agent sends this data to the DRL Model to learn from its experience. The DRL cycle procedure continues to repeat these steps until the goal is reached or a certain step limit is exceeded.

Following the DRL cycle and its structures, several components were designed as part of URNAI's architecture as abstract classes. All of them are tied together by different methods, defining a communication protocol among Agent, Model and Environment, as shown in Fig. 1.

In Fig. 1, we have all components that are responsible for the Agent's workflow. These are their definitions:

- **Model:** Memory structure that rules AI reasoning, controlled by a DRL Method (or Algorithm) that is responsible for updating it. It attempts to learn a policy in order to be able to properly select actions for an Agent according to its situation. Any deep

memory structures can be designed here, such as a DNN, as well as other simpler memory structures;

- **Reward:** Reward Function that determines the feedback (rewards and punishments) received by the Agent when it selects actions using a Model;
- **State:** State wrapper to the environment observation. Depending on the problem at hand, it might be advantageous to represent the environment under different perspectives, filtering out irrelevant information that the Agent does not need in order to select appropriate actions;
- **Action:** Action wrapper to tell the Agent which actions it can select to perform in the environment at any given moment. For example, there are scenarios in which the Agent is unable to perform some actions, so the Action Wrapper is responsible for controlling this in order to make these actions unavailable, simplifying the learning process and making it more likely to be successful. Also, the Action Wrapper may be used to build high-level abstractions to simplify large action spaces (such as the ones present on RTS games, like Starcraft);
- **Agent:** Component that joins the Model, State Wrapper, Action Wrapper and Reward Function. The Agent is one of the most central pieces of URNAI's DRL cycle. It exchanges data with the Model and the Trainer, requesting for actions from the Model and sending them to the Trainer, always applying the abstractions set up within the Wrappers. The Agent also calls the Model's learning method whenever a learning cycle is completed;
- **Trainer:** Component responsible for making the interaction between an Agent and the Environment. Here is where the whole training loop occurs, since it manages the exchange of data between the Agent and the Environment, as well as handling the amount of game matches that the Agent will be trained for. Another important role of the Trainer is to transfer the training data to the Statistics component. So that graphs can be automatically generated by tracking important training data;
- **Statistics:** Component that can be used to store relevant training data. It generates text-based reports and uses that to create graphs showcasing the evolution of important data, such as the average agent reward, the average win rate, the amount of each action being used in every game, etc., all of that throughout training;
- **Persistence:** Component used to store training data on the disk. It allows the persistence of the Agent to the media, allowing it to be transferred between computational nodes. It is specially useful when using the toolkit in the cloud or in supercomputing environments, since they are headless environments and the agent model is usually downloaded to be tested on systems with graphical interfaces;
- **Environment Wrapper:** Wrapper for an instance of a supported game environment. It is important to note that URNAI leans heavily on the environment structure standardized by OpenAI's Gym. Therefore, the Environment Wrappers are mostly used to standardize function calls to all foreign environments and bridge the gap between any game environment that may differ from Gym's structure.

All components exposed above are designed as template classes that have all the required abstract methods for the architecture to function. Beyond the base template classes, that should serve as guidelines for the development, URNAI comes with a serie of out-of-the-box action, reward and observation wrappers, as well as many algorithms, such as Deep Q-Network (DQN), Double Deep Q-Network (DDQN), Policy Gradient (PG) and Tabular Q-Learning.

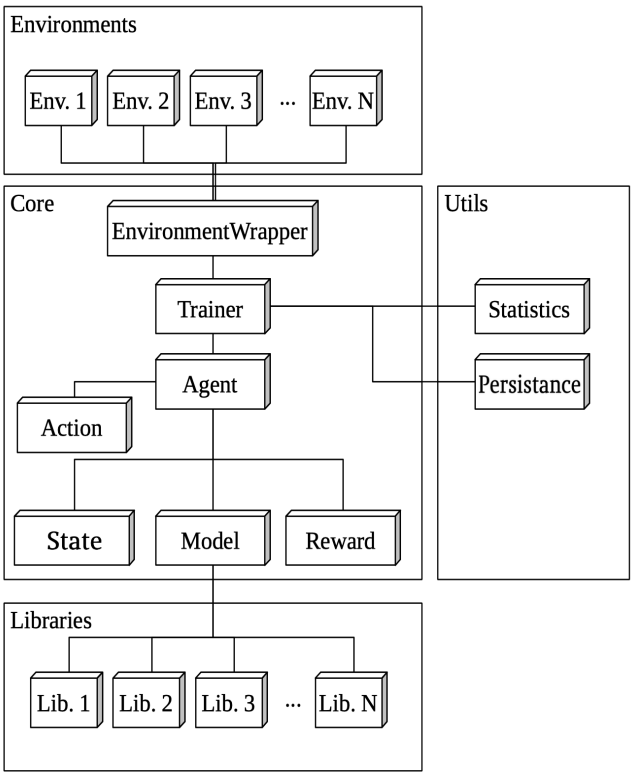

Fig. 1. URNAI architecture is composed of three main layers. From the lower level to the higher level, we have the following: Libraries layer, Core layer and Environments layer.

The out-of-the-box nature of these wrappers allows users to assemble different configurations for reward functions, action spaces and state observations in order to more quickly iterate through different possibilities while testing.

Currently, URNAI supports one out-of-the-box Environment Wrapper for each one of the supported game environments. As said before, these wrappers bridge the connection between external game environments and URNAI, generally just conveying all raw information that the game sends back.

- Layer 3: Game Environments

The high level layer of the architecture is composed of Game Environments. A Game Environment is any platform that has an interactive interface, allowing users to change the environment, and also has some sort of feedback on those changes. Game Environments for DRL algorithms are generally prepared to be used by AI Agents, and often have Application Programming Interfaces (API) that integrate them to programming languages, making the process of interpreting the information to and from the game easier.

A general requirement of Game Environments for DRL is that agents need to be able to affect the environment, and that generally comes as a set of possible actions in the game, such as moving left or right. Another common feature of Game Environments is that they return some sort of observation after an action is taken, representing the state of the game at a given moment and serving as a feedback to the agent. This observation can consist strictly of game scenario data or other kinds of data such as end of the game status, agent score and so on.

As seen previously, the connection between URNAI and Game Environments is bridged by Environment Wrappers. To do that, the Environment Wrappers pass to the environment the actions taken by the agent, and then receive an observation back. URNAI's structure requires that the Environment Wrapper returns to the agent three pieces of data: an environment observation, a reward representing whether the agent has won the match, and a flag showing if the game has ended. If the game environment does not send back all these pieces of data by default, it is in charge of the Environment Wrapper to interpret the observation and create them.

B. Documentation

The toolkit documentation is available at URNAI's *github* repository [5]. It is composed of four main parts: *README.md, generated docs, diagrams* and *results.*

README.md is a file in which one can find an overview of URNAI toolkit. How to install it, how to use command line tools, dependencies, supported Game Environments and so on. In *generated docs* there is a documentation for every class implemented in the toolkit, following the main directory structure. In *diagrams*, one can see a graphical view of URNAI's structure. Finally, in *results,* one can find statistics for some experiments done by using the toolkit.

IV. Libraries and game environments linked to URNAI

In order to make a multi-game toolkit available for DRL experimentation, URNAI needs to support the most recent and robust technologies used by the Machine Learning community. This is achieved in two different ways. First of all, by integrating state-of-the-art Machine Learning libraries to URNAI's repertory. Secondly, by adding support to modern and generalistic game environment toolkits, as well as more complex and detailed game environments, as detailed below.

A. Libraries

Currently, URNAI has three Machine Learning libraries already integrated to it:

- **TensorFlow**: TensorFlow [3] is an open software library that serves as a broad platform for machine learning. It has a broad and flexible ecosystem of resources that allows researchers to push the state-of-the-art in machine learning;
- **Keras**: Keras is a library that builds upon the capabilities of TensorFlow and simplifies much of its workflow, allowing for easier implementation and iteration of machine learning algorithms. Even though Keras has high-level abstractions to simplify machine learning solutions, it also allows for low-level development to experiment research ideas;
- **PyTorch:** PyTorch [23] is a Python package that allows to create and train DNN using different methods such as the ones present in Tensorflow. Those data structures are built using a Tape-Based Autograd System, which allows the DNN to be more flexible and dynamic when compared to other methods.

B. Game Environments

URNAI currently supports several game environments (see Fig. 2):

- **Gym**: A toolkit developed by OpenAI that consists of a collection of several environments that share a common interface, enabling the use of generalized algorithms to solve them [2]. Gym's environments are generally very simple, consisting of basic mini-games or classic control problems, and run very efficiently, which makes it a very useful tool for testing Machine Learning algorithms;
- **StarCraft II**: StarCraft II is a commercial real-time strategy (RTS) game. Due to the very high complexity, unpredictability, strategic and real-time nature of StarCraft matches, it has emerged as a grand challenge for AI research [7]. Following the release of PySC2, a DeepMind's open-source Python library that gives access to the game's internal information, researchers are able to experiment algorithms, directly interacting with the game environment. So, StarCraft II (SC2) has become much more accessible to the wide research community. As mentioned before, it is a very complex environment that poses hard challenges for RL, which is a very interesting research goal. However, a downside to SC2 is it's computationally demanding nature, since it is a full-fledged game. It generally requires the employment of supercomputing with multiple GPUs to achieve reasonable training speeds;
- **VizDoom**: VizDoom is a platform developed to allow the training of AI agents within the iconic 1993 computer game DOOM [6]. VizDoom is primarily focused on computer vision based RL, since it represents the information from the game as series of images. Consequently, this environment is an interesting option to URNAI's repertory, since PySC2 does not represent the game as an image, and neither do most of Gym's environments;
- **DeepRTS**: DeepRTS is a game environment developed to mimic a RTS game, such as StarCraft II and so on [4]. Its focus is to provide an environment for training and experimenting DRL agents. DeepRTS reaches much better performance than a full-fledged game, achieving up to seven million frames per second. Therefore, DeepRTS is an interesting environment to have as an option in URNAI, since it can be viewed as a simplified and much more performant counterpart to StarCraft II, allowing researchers to experiment with both environments in several different ways.

Since these environments can be quite different from each other, URNAI provides preset Agents, State Representations, Reward Functions and Action Wrappers for each of them.

The Agent module has two preset classes: a generic agent that works on Gym, VizDoom and DeepRTS, and a second agent that is designed specifically for StarCraft II. Similarly to the Environment Wrappers, these agents are generic, and hardly need to be tinkered with, as they fulfill all basic needs of a RL Agent, without inferring any abstract view of the data.

Fig. 2. Game Environments supported by URNAI. Top Row (left to right): Gym, StarCraft II. Bottom Row (left to right): VizDoom, DeepRTS.

To affect the way the agents perceive the environment, a State Builder is used to represent a State (see Fig. 1). Each supported environment has at least one state builder related to it. StarCraft II, for instance, has three different out-of-the-box State Builders, each of them with a different approach to representing the environment.

Similarly to the State Builders, URNAI provides at least a Reward Builder for each game environment. StarCraft II and VizDoom have multiple builders. These Reward Builders define how an Agent can interpret the current state of the game, directly affecting the training.

Finally, the last type of wrapper provided by URNAI is the Action Wrapper, used to determine which actions must be available to the Agent at any given moment. Each game environment has at least an Action Wrapper implemented. StarCraft II has four Action Wrappers and VizDoom has two of them. StarCraft II also has a generalistic wrapper implemented for each playable race of the game, so that researchers can either use these generalistic wrappers as they are, or as a base to implement more specific wrappers that fit their research purposes.

V. Setting up URNAI to do experiments

Before experimenting with URNAI, there are some instructions to follow and some setting up to be done. A first and preliminary step towards training is deciding which Learning Model, Agent and Environment to use. As explained before, URNAI comes with many out-of-the-box wrappers. Therefore, it is recommended that users visit the documentation on *github* in order to see a more in-depth list of all available components, such as agents, environments, reward functions, algorithms, etc.

After components are selected, the next step is effectively setting up each component. Instructions on how to configure the Environment Wrappers are very specific to each game environment, and can be further verified in the documentation. Configuring Learning Models, however, is a much more generic task, in which URNAI users can specify most of the usual parameters in any DRL algorithms, such as number of layers, type of layers, number of neurons per layer, learning rate, exploration parameters etc. Lastly, configuring the Agent is a quite standard procedure, and only requires the user to select an action wrapper, a state builder and a reward builder.

The last step before proper training consists in selecting the training parameters, which are defined by the Trainer component. These parameters are related to the number of episodes the agent will train, the maximum number of steps the agent can act in each episode, the frequency of saving the Learning Model, the saving path, and so on.

In the following subsections, there is a brief explanation of the main features of each URNAI component, as well as any additional information that may aid users in understanding URNAI's structure and its use.

A. *Selecting algorithm*

Currently, URNAI has four Learning Models implemented: Q-Learning (tabular), Deep-Q Learning (fully-connected neural networks), Double Deep-Q Learning and Policy Gradients.

Those different kinds of models are present to allow the user to choose which is the most suited for its context. For example, Tabular Q-Learning is an excellent choice to be used in simple environments, such as the ones presented in Gym's collection. For those games, the tabular model is not only faster to train, but it is more stable to learn an usable policy. On the other hand, all the other environments require a more robust learning model. Deep-Q Learning (DQL) and Double Deep-Q Learning (DDQL) are examples of such. These algorithms use DNN as memory representation, allowing the training of the agent on more complex environments, with bigger state spaces.

Alongside that, a component called Model Builder is provided. This component helps the user dynamically build different structures of a Neural Network, by defining the amount of layers, type of layers, number of neurons and any other information necessary to the creation of a functioning Model. An interesting aspect of the Model Builder is that its main structure is defined as a Python dictionary, which allows users to build Learning Models without having much experience in programming. Another possibility, considered to be easier, is to instantiate a Model Builder object in Python and call its methods to add new layers. So, it dynamically builds the dictionary for the user.

B. *Selecting game environment*

As mentioned before, there are four main environments currently supported by URNAI: Gym, Starcraft II, VizDoom and DeepRTS. After an environment is selected, the user

needs to access the class of the Environment Wrapper selected, or any of the examples available in the repository, in order to instantiate their chosen environment.

C. Selecting Agent

There are two main agents in URNAI. For most training scenarios, a user can adopt a *GenericAgent*, except for Starcraft II that needs to use the *SCIIAgent*, designed to accommodate for some differences in the SC2 environment.

D. Selecting Action Wrapper, State Builder and Reward Function

Generally speaking, for each environment there are many different ways an agent could interact with it, and that is specially true for more complex environments such as StarCraft II or VizDoom. In order to give users some flexibility while training their agents, a few Action Wrappers are available for each environment. Some Action Wrappers can be used in any configuration of the game, while others are supposed to be used with some restrictions, such as in a special map, or in a specific game mode.

For example, there are classes made to be used with a specific Starcraft II playable race, or even in a specific scenario of VizDoom. More details about each one of these specifications can be found by referring to URNAI's documentation.

The same rules are applied to State Builders and Reward Functions. There are generic ones made to work with any context of a game. But there are also ones tailored to more specific scenarios or approaches to RL.

VI. Start Training in URNAI

After selecting all components of the training, the agent (with all its parts) and the environment, we will be able to start training. In this section we describe two ways to do this: coding directly in Python or using JSON files. At the end of the section, we have a special subsection that describes how to run URNAI on remote machines, such as GPU supercomputers.

A. Python

One way for using the toolkit is importing it in a Python script and joining all the components manually. In general, what it is needed to do is to instantiate each object with all of the chosen parameters and then ask the Python interpreter to run the script. Users can refer to the source code on *Github* to see examples of premade scripts called *Python Solve Files*, available in the *solves* directory. Generally, there is at least one solve file for each Game Environment.

B. JSON

JSON files are also supported by URNAI, being useful for users who are not familiar with Python or want a more direct way to select components and parameters. Some training examples using *JSON Solve* files are available in the *solves* directory. Users are encouraged to use these files as a base when creating their own JSON Solve files.

To start training using JSON files, the user can call URNAI from the command line, like so:

URNAI train --json-file file.json

C. Tuning learning parameters

When the training script is ready, in Python or JSON, the next step is tuning learning parameters, which is just a matter of swapping training components and changing parameters' values. For instance, if the user wants to change the reward function, he could swap the Reward Builder. Moreover, if he wants to change the DNN structure, he could update the model parameters and so on. This process is often done after an initial training, to accommodate the results, try and improve the model.

D. Supercomputing

In the current version, URNAI uses Keras as the main library for DRL. This means that Tensorflow is used to train DNN models, since it is a Keras dependency. These libraries make URNAI naturally able to run different setups, from a computational power perspective, such as in supercomputers on cloud services or in remote supercomputing systems.

One way that was found to be reliable in making URNAI run in supercomputers was to use container managers to simulate operating systems, consequently removing errors related to differences in system version, libraries and dependencies. A recipe to build a container that runs URNAI, using the Singularity container manager, is provided in the source code as an example.

VII. Expanding URNAI

A. Adding new libraries by implementing new models

In the previous section, we pointed out that the models of the current URNAI version are written using Keras. However, this does not mean that only this library could be used to code models.

The main goal of URNAI is to be flexible. So, adding new learning algorithms and libraries should be an easy task to do. That is why the Model has a set of abstract methods that can be implemented according to each required external library. These methods can be found in the *ABModel* class, which all models should inherit from.

In general, the main focus of a Model is on the *learn()* method. It receives a State, Action, Reward and State tuple. Thus, it should internally follow its own learning algorithm to improve the model's decision making process. So, in theory, any Machine Learning Library could be used here, the only limitation being that it must be Python compatible.

We encourage and invite any researchers interested in this topic to contribute to URNAI's repertory of learning models, by coding any new model they are interested in. Since URNAI's code is hosted on *github,* any developer can send a push request and collaborate.

B. Adding new game environments

To add new game environments, we can use the same idea described in the previous section. The *Env* class implements a series of abstract methods used by all Environment Wrappers. So, all that is needed to add a new game environment is to create a new class that inherits from *Env* and then implement its methods.

The main focus of an environment is on its *step* method. Here is where the connection with the game happens, since the environment wrapper receives an action as a parameter

of this function and then sends it to the real instance of the game. After delivering the action to the instance, the wrapper usually receives a reward and a state (this heavily depends on the game being adapted), which should be returned to the agent.

This is the main functionality of the *step()* method, but it may vary from an environment to another. For instance, before returning the state to the agent, the DeepRTS wrapper inserts game data in the object, such as player status, units, and so on, since the main state object of the game is just a map configuration.

It is important to note that the structure and functionality of URNAI's Environment Wrapper leans very heavily on OpenAI Gym's environment definition. So, if a new Environment Wrapper implementation is turning out to be a bit more complex than anticipated, it might be useful to check out the documentation and try to understand how the environment transfers data. So, URNAI's Environment Wrapper structure basically extends Gym's structure, opening the door to implementing other game environments.

VIII. Usage Example

In this section, a practical example of using URNAI will be discussed, going through the same steps detailed on sections V and VI. The focus here is to show a step-by-step guide on how to set up a training example from beginning to end, showcasing all of the components needed and presenting their real names.

A. Choosing Training Components

As mentioned before, the first step is to select the components required to prepare the training. In this practical example, we will be trying to make an agent learn how to defeat the Starcraft II default AI in the very-easy level. The map chosen to train on is the Simple64 map, provided by DeepMind's PySC2 library, and the race of our agent is going to be Terran.

So, URNAI has two main core parts: Agent and Environment. If we start on the agent, we can see on the documentation that we need to choose a State Builder, a Learning Model, a Reward Function and a Action Wrapper. For each one, there will be a set of classes that can be used. Below will be presented the classes selected:

- State Builder: since we are playing on the Simple64 map, the state builder will be *Simple64GridState;*
- Model: the model used here is a DNN controlled by a Double Deep-Q Learning algorithm, then the class is a DDQNKeras;
- Reward Function: there are several possibilities to reward the agent on this map, but the one that works best on this context is *KilledUnitsReward*;
- Action Wrapper: there is a generalized action wrapper for the Terran race, called *TerranWrapper*. However, since the Simple64 map is very simple and small, a simpler version of the Terran wrapper will be used: *SimpleTerranWrapper*;

In addition to that, there are the Agent and Environment classes. Starcraft II, for instance, needs its specific agent class called SC2Agent. For the Environment Wrapper, the proper class for StarCraft II is called SC2Env.

B. Coding The Training File

Once all required components are selected, it is now time to assemble them together in either a Python file or a JSON file in order to start training. This section will present a brief explanation and code snippets of these files, but for a deep dive in them we recommend that users visit the source code on the *github* repository. So, in Python, we instantiate the environment like in Fig. 3.

```
env = SC2Env(map_name="Simple64", render=False,
step_mul=16, player_race="terran",
enemy_race="random", difficulty="very_easy")
```

Fig. 3. Instantiating Starcraft II environment wrapper.

Then we instantiate the Action Wrapper and the State Builder in Fig. 4.

```
action_wrapper = SimpleTerranWrapper()
state_builder = Simple64GridState(grid_size=4)
```

Fig. 4. Instantiating Starcraft II Simple Terran Action Wrapper and Simple 64 State Builder.

Following, we will use the ModelBuilder by quickly creating a DRL Model with two hidden fully connected layers, each one of them with 50 nodes. Beyond that, we will be using the state builder to get the dimension of our input layer, and the action wrapper to get the output dimension, as we can see on Fig. 5.

```
helper = ModelBuilder()
helper.add_input_layer(int(state_builder.get_state_d
im()), nodes=50)
helper.add_fullyconn_layer(nodes=50)
helper.add_output_layer(action_wrapper.get_action_sp
ace_dim())
```

Fig. 5. Using ModelBuilder to generate DNN.

We can then instantiate the Learning Model as we do in Fig. 6. In this particular case, the best training setup tuned the learning rate to 0.1%, the rate of decay of epsilon greedy strategy to 0.001% at every game step, and a maximum memory of 100,000 tuples, as well as a few other details that can be further understood by reading the source code and the documentation.

```
dq_network = DDQNKeras(
action_wrapper=action_wrapper,
state_builder=state_builder,
build_model=helper.get_model_layout(), gamma=0.99,
learning_rate=0.001, epsilon_decay=0.99999,
epsilon_min=0.005, memory_maxlen=100000,
min_memory_size=2000)
```

Fig. 6. Instantiating the Double Deep-Q Network Model.

After that, we can instantiate the Agent itself, like in Fig. 7, passing the Learning Model and the Reward Builder (*KilledUnitsReward*).

```
agent = SC2Agent(dq_network, KilledUnitsReward())
```

Fig. 7. Instantiating Starcraft II Agent.

Finally, we will instantiate the Trainer, as seen in Fig. 8, and prepare it for training. Some parameters required for that are the saving directory, the file name, the frequency at which the trainer will save the Agent to disk, the number of episodes for training and playing, and the maximum number of steps in each game episode.

The Python file is finally ready for starting training. A brief explanation on how to run this file is given in the next subsection. Some programming details were omitted from this guide for simplicity sake, but the full file can be found in the source code under the *solves* directory as *solve_simple64_veryeasy.py*.

```
trainer = Trainer(env, agent,
save_path='urnai/models/saved',
file_name="terran_ddqn_vs_random_v_easy",
save_every=100, enable_save=True,
relative_path=True)
trainer.train(num_episodes=3000, max_steps=1200)
trainer.play(num_matches=100, max_steps=1200)
```

Fig. 8. Instantiating Trainer, which is responsible for joining the Agent and the Environment Wrapper.

An alternative to Python files is to create a JSON file. The steps to code this type of file are very similar to the ones exposed before, and mainly consist in going from module to module and declaring each one with the proper parameters. Therefore, we will not be showing code snippets of this process since a JSON file can become quite extensive, and the parameters chosen are exactly the same as the ones shown on the Python snippets. So, if a user wishes to see the full file, they should refer to the source code under the *solves* directory, as *solve_simple64_veryeasy.json*.

In this sense, there is a small difference in JSON parameters that should be clarified. When declaring the parameters of the build model, we should explicitly write the dimensions of the input and output layer, as it can be seen in Fig. 9. An easy way to do that is to check our State Builder and Action Wrapper, respectively, to see the size of their output. This process can be seen below, in which the size of input layer is 54, due to the size of our State Builder's output, and the size of output layer is 34, due to the size of our Action Wrapper's output.

```
"build_model" : [
    {
        "type" : "input",
        "nodes" : 50,
        "shape" : [null, 54]
    },
    {
        "type": "fullyconn",
        "nodes": 50,
        "name": "fullyconn0"
    },
    {
        "type": "output",
        "length": 34
    }
]
```

Fig. 9. Setupping Double Deep-Q Network in JSON Solve File.

C. *Starting Training*

Running the scripts created in the last subsection is a relatively simple task. To do that with the Python file, all that is needed is to run it with a Python interpreter, like so:

python /your/file/directory/solve_simple64_veryeasy.py.

Running a JSON file is similar, the only difference being that we need to have URNAI installed as a Python package. So, we are able to use it in the command line:

urnai train --json-file solve_simple64_veryeasy.json

```
from jsontrainer import JSONTrainer

trainer =
JSONTrainer("your/file/directory/solve_simple64_very
easy.json")
trainer.start_training()
```

Fig. 10. Using JSON Trainer in a Python Script.

Furthemore, there is a workaround that allows users to run a JSON file without installing the toolkit to their Python environment, which could be useful in situations where users have limited access or permission. This involves creating a Python script importing a JSONTrainer class and instantiating a trainer based on a JSON file (see Fig. 10).

IX. EXPERIMENTS CARRIED OUT AND RESULTS OBTAINED

The goal of this section is to present the results obtained with an agent trained using URNAI. It is highlighted that our objective here is not to create a state-of-the-art agent that defeats pro-level players of Starcraft II. Instead of that, we will demonstrate that it is possible to run an agent with the toolkit and generate some statistical results to see how well the agent is doing during the training and playing steps.

A. *Training Step*

The agent trained for 3000 episodes during the learning step. It started by randomly trying all of the actions available in its Action Wrapper. This behavior can be verified at the first 200 episodes in the graph of Fig. 11. The agent has a peak in average reward at the beginning, when there are a lot of random variations, and then it starts to fall until around episode 600. After that, it starts to learn a good strategy and raises the average reward.

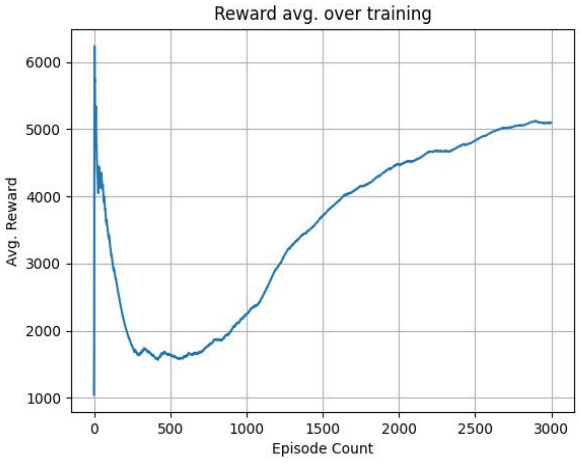

Fig. 11. Average reward of the agent by episode.

There are some other graphs that we can use to observe the agent's behavior. In Fig. 12, it is presented that the agent takes fewer and fewer steps to end the game as learning progresses. The number of steps is the amount of actions the agent takes to end an episode (game). If we only consider this figure, this would mean that the agent could be losing quickier as time went on, but Fig. 11 and Fig. 13 show the opposite.

Finally, in Fig. 13 we can see the average winning rate of the agent. The more he wins, the higher the average is. By the end of the training, the agent winning percentage is in the 50% range against his opponent.

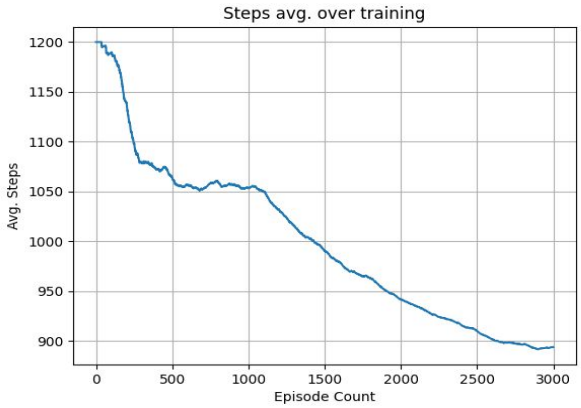

Fig. 12. Number of steps the agent has taken to end training episodes.

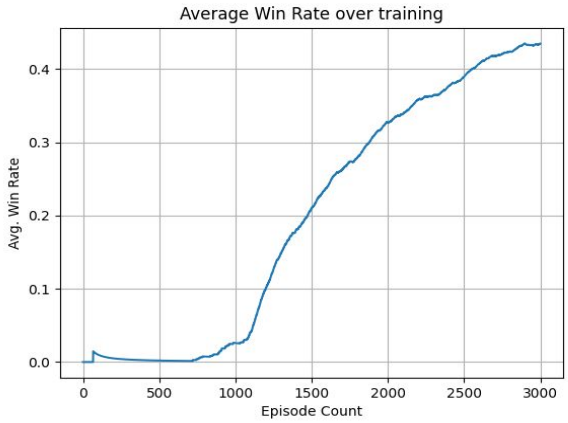

Fig. 13. Average winning rate of the agent.

By analyzing all these three figures, we can see that the agent learns how to win and even how to win quicker. This shows that it might be using some *rushing* strategy, which means that it tries to attack as soon as possible, allowing it to surprise the enemy before it can create an army or even grow an economy.

B. *Playing Step*

After the training ends, the agent's knowledge can be evaluated by leading it to play episodes without learning. In this subsection, we show the evaluation results for the agent playing 100 episodes.

Fig. 14 shows how reward tended to fall during this evaluation session. At first glance, the agent might seem to be losing, but if we compare the average with the one presented on Fig. 11, we can see that it is much higher. So the agent is still winning, but the average is falling since its winning rate is not 100%.

We can see that the winning rate has the same trend as in average reward. It falls because the enemy is not winning constantly, but the winning rate is still high, around 84%. This can be checked by looking at Fig. 15, in which we might verify that the winning percentage is much better than the rate obtained during training.

Finally, Fig. 16 shows the agent is still trying to win as fast as possible, even quicker than while it was training. The average number of steps rises during a hundred evaluation episodes, going up to 800 steps, but this value is still lower than the training average (900 steps).

Fig. 14. Agent's average reward when evaluating playing episodes.

Fig. 15. Agent's winning rate when evaluating playing episodes.

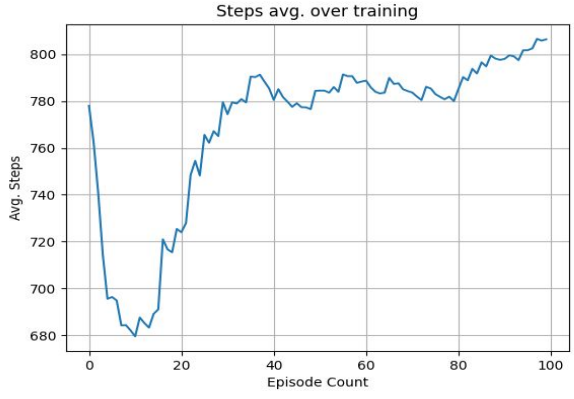

Fig. 16. Agent's average number of steps to end episodes.

We can conclude the agent was able to develop a good strategy in the super easy AI level of Starcraft II. It might be a rushing strategy, since the average number of steps tended to be low at training stage and is even lower in evaluation. But it is not the best strategy because we can see that the winning rate while training was around 50%, and it tended to fall while evaluating the agent.

X. Conclusions and Future Works

This paper introduced URNAI, a new toolkit that facilitates the training and evaluation of AI into multiple game environments. The main feature of the toolkit is its architecture, which allows the configuration of multiple learning scenarios and agents, as well as the integration of several games and new libraries for DRL algorithms.

We also showed how to set up and use the toolkit, as well as how to do an experiment that showcased some of its capabilities. In the experimentation section, an agent was able to learn a good strategy against the very-easy AI difficulty of StarCraft II. This demonstrates that URNAI is currently functional and can be used for various types of learning experimentation settings.

However, since this is URNAI's first version, it is clear that there are several improvements that could be done. So, the next step in URNAI's development is to provide other learning models, libraries and games. Examples of such could be the Asynchronous Advantage Actor Critic (A3C) algorithm as a new model, Theano as a new library, and the Arcade Learning Environment (ALE) [1] as a new game environment.

Also, some aspects of the architecture could be improved, such as modules to standardize environments' states and actions, and the possibility to inherit the toolkit to use some of its services, which could lead to it eventually being transformed into a framework. Additionally, learning scenarios could be included to make training easier for users who want to configure only the learning model parameters, without worrying about action wrappers, reward functions, etc.

Finally, we invite researchers and developers to contribute to URNAI's repository [5] in order to add new features, such as new models and environments, or just improving its architecture. Since it is hosted in *github*, anyone willing to collaborate can send a pull request and help improve this work.

Acknowledgment

We would like to thank the National Council for Scientific and Technological Development (CNPq) for providing scientific research scholarships that made this endeavor possible. We would also like to thank the High Performance Computing Center of the Federal University of Rio Grande do Norte (NPAD/UFRN) for providing the infrastructure and support necessary to run our experiments in a GPU supercomputing environment.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, M. Bowl-ing. The Arcade Learning Environment: An Evaluation Platform for General Agents. In International Joint Conference on Artificial Intelligence, 2015.

[2] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. Openai gym. arXiv preprint arXiv:1606.01540, 2016.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In12thUSENIX symposium on operating systems design and implementation (OSDI 16), pages 265–283, 2016.

[4] P. A. Andersen, M. Goodwin, O. C. Granmo. Deep RTS: a game environment for deep reinforcement learning in real-time strategy games. In 2018 IEEE conference on computational intelligence and games (CIG) (pp. 1-8). IEEE, 2018..

[5] URNAI Tools Repository. Federal University of Rio Grande do Norte, 2020. Available in: https://github.com/marcocspc/URNAI-Tools.

[6] M. Kempka, M. Wydmuch, G. Runc, J. Toczek & W. Jaśkowski, ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning, IEEE Conference on Computational Intelligence and Games, pp. 341-348, Santorini, Greece, 2016 (arXiv:1605.02097).

[7] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. Vezhnevets, M. Yeo, A. Makhzani, H. Kuttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, R. Tsing. StarCraft II: A New Challenge for Reinforcement Learning. arXiv preprint arXiv:1708.04782, 2017.

[8] Christopher JCH Watkins and Peter Dayan. Q-learning. Machine learning, 8(3-4):279–292,1992.

[9] Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G.Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostro-vski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King,D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.

[10] Y. Tian, Q. Gong, W. Shang, Y. Wu, C. Zitnick. "ELF: An Extensive, Lightweight and Flexible Research Platform for Real-time Strategy Games". arXiv:1707.01067, 2017.

[11] Q. Wang, J. Xiong, L. Han, M. Fang, X. Sun, Z. Zheng, P. Sun, Z. Zhang1. "Arena: a toolkit for Multi-Agent Reinforcement Learning". arXiv:1907.09467, 2020.

[12] R. S. Sutton and A. G. Barto, Introduction to Reinforcement Learning, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[13] P. B. S. Serafim, Y. L. B. Nogueira, C. A. Vidal, J. B. Cavalcante Neto. Evaluating competition in training of Deep Reinforcement Learning agents inFirst-Person Shooter games. Proceedings of SBGames 2018, 2018.

[14] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, D. Hassabis. Mastering the game of Go without human knowledge. Nature, v.550, p.354-359, 2015.

[15] G. Tesauro. Programming Backgammon Using Self-teaching Neural Nets. Artificial Intelligence, 134:181-199, 2002.

[16] M. Laoan. Deep Reinforcement Learning Hands-On. Packt PUblishing, 546p, 2018.

[17] S. Ontañon, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, M. Preuss. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. IEEE Transactions on Computational Intelligence and AI in Games, v.5, n.4, p.1-19, 2013.

[18] G. Robertson, I. Watson. A Review of Real-Time Strategy Game AI. AI Magazine, v.35, n.4, p.75-104, 2014.

[19] M. Buro. Real-Time Strategy Games: A new AI Research Challenge. In Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, 2003.

[20] I. Goodfellow, Y. Bengio, A. Courville. Deep Learning. MIT Press, 2016.

[21] D. Churchill. A history of starcraft ai competitions. AIIDE Starcraft AI Competitions, 2016.

[22] P. Molino, Y. Dudin, S. S. Miryala. "Ludwig: a type-based declarative deep learning toolbox". arXiv preprint arXiv:1909.07930, 2019.

[23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Advances in Neural Information Processing Systems 32 (NIPS 2019), p.8026-8037, 2019.