Cart Pole Using Reinforcement Learning – Component-Based Design Tutorial

Ammar Zulqarnain Samir Gupta
Computer Science Department, Vanderbilt University, Nashville, USA

***Abstract* -The main objective of this paper is to teach a bot to control a Cart Pole system. The objective of the Cart Pole is to balance the Pole placed on top of a Cart; the Pole moves in random directions due to forces such as inertia and friction. Hence our bot learns how to balance the Pole by counteracting the forces applied by moving the Cart. Our main goal in this paper is to provide a tutorial for a component-based design of the Cart Pole System using Reinforcement Learning.**

***Keywords*- Cyber-Physical Systems, Reinforcement Learning, Cart Pole System, Component-Based Design.**

I. INTRODUCTION

In this project, we have used concepts from Reinforcement Learning to train a bot to control a cart and balance a pole on top of it. The problem may be described as holding a pole attached to a cart such that it should be straight-vertical pointing upwards as much as possible while the cart can speed left or right on a straight trajectory utilizing just one bearing with one degree of freedom. Further, we can expand the problem by adding extra restrictions or mechanical substitutes which help imitate complicated real-world physical systems. Because of the cart pole system's solid physical foundation and the simplicity of the problem, it is an ideal environment for the creation and testing of RL algorithms. The control of articulated mechanical systems is dependent on a high degree of system knowledge, which is mostly stated by multibody dynamics models. The main concern with this strategy is that it is limited to systems that have a high degree of complexity or unpredictability [1]. Thus, using model-based control algorithms can be challenging as it consists of differential algebraic equations of motion, they also depend on model accuracy causing issues in real platforms and leading to the creation of nonlinear systems to be extremely difficult. If we take a more traditional engineering approach to the problem, the process requires deriving systems governing equations and parameter adjusting in the control system.

An alternative approach is making use of Data-Driven methods of Machine Learning such as Reinforcement Learning. Reinforcement Learning (RL) is a machine learning paradigm where an agent learns the optimal action for a given task through its repeated interaction with a dynamic environment that either rewards or punishes the agent's action [2]. Reinforcement Learning is neither Supervised nor Unsupervised it is a mixture of both known as Semi-Supervised Learning. Recently Reinforcement Learning has been gaining a lot of success in the small discrete state-action space domain. From various experiments and applications, it has also been observed that RL is very useful for learning the dynamic behavior of an agent as compared to static mappings in between sets of input and output variables. RL has also seen a lot of advancement in applications such as Self Driving vehicles (Autonomous Vehicles), Stock Market Trend Analysis, and Energy Management with the help of function approximations which are applied to a large and continuous state space. In this regard, it has become common to apply Deep Learning along with Reinforcement Learning known as Deep Reinforcement Learning.

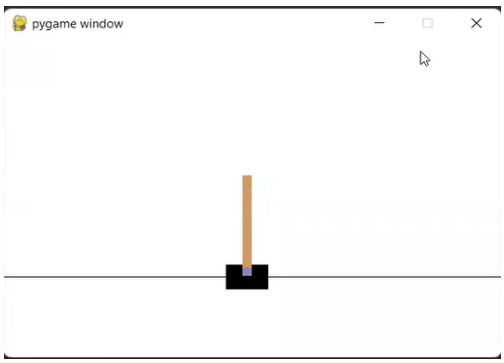

Figure 1

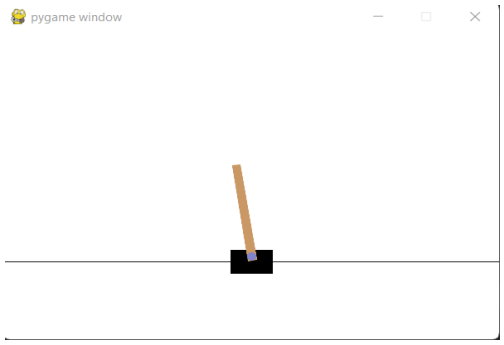

Figure 2

In this project we have made use of the Open AI Gym module available in Python to simulate the Cart Pole system as shown in Figure 1 and Figure 2. Figure 1 indicates the initial state and Figure 2 describes the Cart Pole after random perturbations are applied to it. It is made up of a cart (shown in black) and a vertical bar that is joined to the cart via a passive pivot joint. The cart can move left or right on the

horizontal axis. The safety condition is to prevent the vertical bar (Black Pole) from falling by moving the cart left or right. By running the code, we can observe the animation of system behavior under the random action policy.

We explain how we have incorporated a component-based design of the entire Cart Pole System in Section II. We also explain the results, safety/liveness requirments and analysis in the Section III.

II. Methodology

Some definition that requires clarification regarding a component-based design are mentioned below:

1. *Input Variables*

   An input to a reactive component C is a valuation over the set 'I' of its input variables, and the set of all possible inputs is 'Q_I'.

2. *State Variables*

   The state of component C is a valuation over the set 'S' of its state variables, and the set of its states is 'Q_S'.

3. *Output Variables*

   An output of component C is a valuation over the set 'O' of its output variables, and the set of all possible outputs is 'Q_O'.

4. *Safety Properties*

   A safety requirement for a system classifies its states into safe and unsafe and asserts that an unsafe state is never encountered during the execution of the system. It asserts that nothing bad ever happens.

5. *Liveness Properties*

   The Liveness Properties assert that something good eventually happens.

A. Description of Cart Pole environment

Our program utilizes the OpenAIGym Cartpole environment[3] as our simulation for the actual inverted pendulum problem.

The assumptions of this environment are listed below:

1. The track is frictionless. (Track relates to the surface on which the cart is moving)
2. Gravity is 9.8 m/s
3. The mass of the cart is 1 KG.
4. The mass of the pole is 0.1 KG.
5. The length of the pole is 1 meter.
6. The force on the cart is 10 Newton in each direction.
7. The angle of the pole at which the environment fails is above +12 ^{o} or below –12^{o} (The angle of the pole can still go to ± 24 ^{o})
8. The position of the cart at which the environment fails is when it exceeds 2.4 m from the center on either side.

The state space of this environment consists of four variables:

Num	Variable	Lowest Val	High Val
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

Table 1: State Space of CartPole Environment

The action space of this environment is Discrete with two actions:

1) Push Cart to the Left (0)
2) Push Cart to the Right (1)

Reward of +1 is given for every step of an episode the pole stays on Cart (Environment is not failed).

This environment was the closest simulation we can get to the actual inverted pendulum problem in python.

B. CartPole System Interface

The CartPole System Interface is composed of three individual components that interact with each other in a synchronous clock cycle. These components are:

1) Trainer Component: The trainer takes in the initial conditions (Cart Position, Cart Velocity, Pole Angle, and Pole Angular Velocity) and outputs optimal policy network weights. This whole training of the weights is done in the first clock cycle.
2) Tester Component: The tester is connected to the Trainer and it takes in initial conditions of the environment, policy network weights from the Trainer, and the current action to take as input variables. Note that, network weights and current actions are event variables which means that it is not always available throughout the clock cycles.
3) Delay Component: Delay is connected to the Tester and it takes in the next_action from the tester as an input variable and outputs the current_action. The function of this task is to use the next_action generated from the tester to be used as input for the Tester in the next clock cycle.

The interaction between these three components is shown in figure 3.

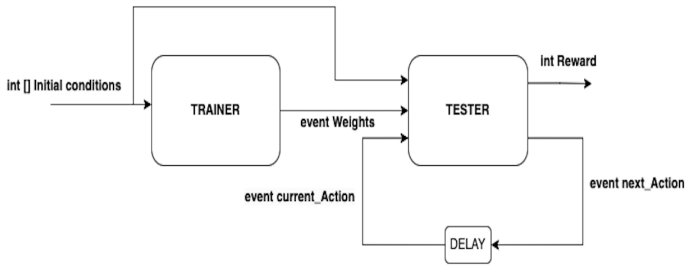

Fig 3: CartPole System Interface

C. Individual Component Design

In this section, we will discuss the internal reactions of the Trainer and Tester Components.

As mentioned before, Trainer takes in the initial conditions of the environment and outputs optimal weights for the policy network. It has three state variables: trained_state, policy_weights, and statevalue_weights.

The state variable trained_state is initialized to False and it stores information about whether the trainer has completed the training. If the trainer has completed training and generated network weights, it is assigned True so that it will always skip the training part in the following clock cycles. Other state Variables i.e StateValue_weights and policy_weights keep track of the weights of networks we are using to train the policy.

The trainer uses RL algorithm called Monte Carlo Policy Gradient Reinforce[4]. Figure 4 shows the setup of the Monte Carlo Policy Gradient Reinforce that is run inside the trainer component.

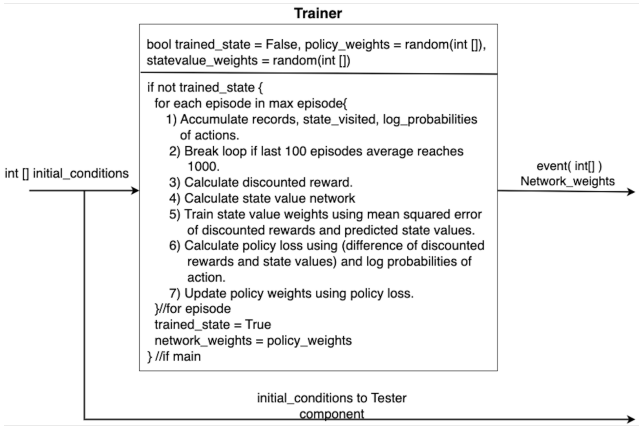

Fig 4: Component Design of the Trainer

This algorithm works in the following sequence:

1) Initialize State Value Network and Policy Network
2) Generate a trajectory of environment and accumulate states, rewards and log probabilities of actions for each trajectory
3) Estimated Returns
4) Predict state values of states visited using StateValue Network
5) Update weights of StateValue Network by using MSE of Returns and predicted state values as the loss function
6) Calculate the policy loss function by using (Returns-predicted state values) and log probabilities of action at each state
7) Update policy networks weights using policy loss
8) Repeat the process until either max trajectories are done or average score of last 100 trajectories is above 1000

One thing to note in the trainer component is that this whole training is done in the first clock cycle. Hence, in following clock cycles, the guard condition of trained_state prevents further training. Hence, it reduced the time to execute.

As far as the design of the Tester is concerned, it takes the network_weights, initial conditions of environment , and current_action as inputs and generates rewards and next_action as output. The state variables of tester are weights(of the policy network), state(current condition of the system that included cart position, cart velocity, pole angle and) and done (Whether the environment failed).

Since the inputs like current_action and network weights are events, we have used guard conditions to specify how the tester will run. Since we have initialized done as False, it will start with the notion that the environment has not failed in the first clock cycles. If there is no current_action or network weights, it will not do anything. If network weights are present and action is not, this suggests that the environment is in the initial conditions and hence it will generate the next_action by calling the function select_action() of the Tester class. Note that next_action goes as input for the delay component as we want to use this action as an input for the Tester in the next clock cycle. When current_action is present, the environment uses that action to assign new values to the state variables done and state (current conditions). It does that by calling the function step_action(). This function also generates both outputs next_action and reward.

State Variable done acts as the monitor for this system that checks whether the environment is in a stable state or not. If the environment has failed (Cart position or pole angle has exceeded), it will generate an error message which means that the system needs to be reset.

Fig 5 summarizes the whole component design of Tester interacting with Delay component.

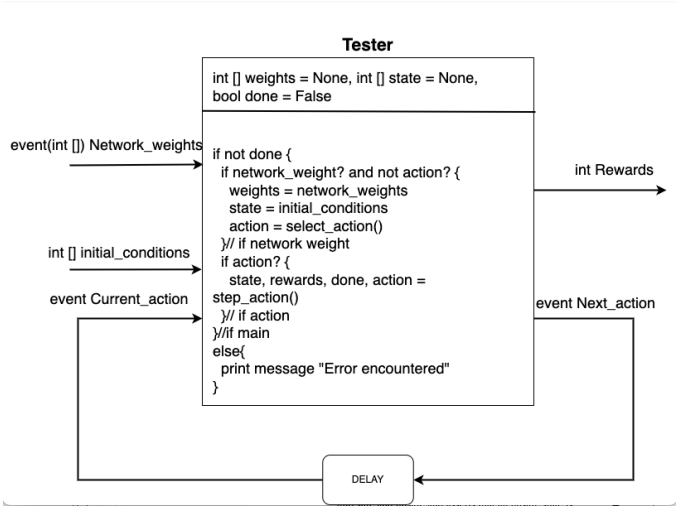

Fig 5: Component Design of the Tester and Delay

D. Networks Architecture

There are two neural network architectures, Policy Network and StateValueNetwork that are utilized in the CartPole environment. The trainer component uses both architectures to train the policy network weights and provide them as outputs. Whereas, the tester component uses only Policy Network to determine the suitable action

The architecture of the Policy Network is defined as follows:

1) An input layer of 128 neurons
2) A hidden layer of 64 neurons
3) An output layer of 2 neurons (action space)
4) ReLU activation functions are used for every internal passing
5) A softmax activation function is used on the output layer to generate probabilities of actions

The architecture of State Value Network is as follows:

1) An input layer of 128 neurons
2) A hidden layer of 64 neurons
3) An output layer of 1 neuron to output predicted state value
4) ReLU activation functions are used for every passing.

III. Results, Safety/Liveness Requirements and Analysis

A. Training Results

Fig 6 summarizes the training results of the simulation we have performed. The hyperparameters set during training were as follows:

- Discount Factor=0.99
- State Value Network Learning Rate= 1e-2
- Policy Network Learning Rate= 1e-2
- Solved Score= 1000
- Maximum number of trajectories during training: 10000
- Maximum number of steps in a trajectory=10000
- Number of last trajectories to take average of= 100
- Optimizer= Adam

The results of the training indicates big spikes in various stages with a general trend of improving performance. The trainer was able to determine the optimal weights in the first 180 training sessions.

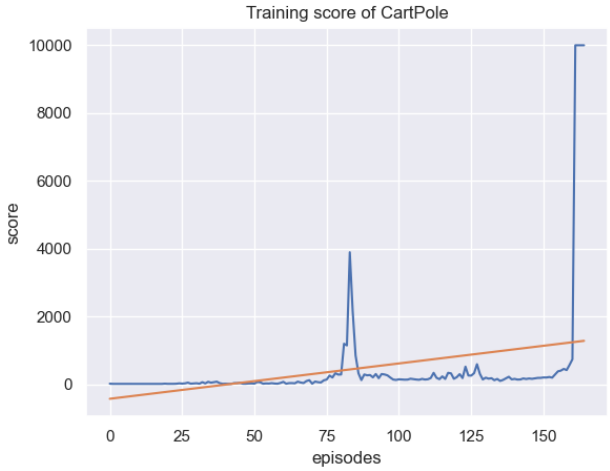

Fig 6: Training Results of Trainer

B. Safety Requirements

For this system, we have identified the following as the safety requirements:

- The pole angle in tester (state[2]) should never exceed is above +12 º or below –12º
- The cart position in the tester (state[0]) should never be above 2.4 and below -2.4
- State Variable done should never True.
- The tester should never generate next_action in the first clock cycle

These requirements ensure that CartPole System is not in a bad state that would lead to failure of environment.

C. Liveness Requirements

For this System, we have identified the following as the liveness requirements:

- The average score of last 100 trajectories eventually goes above 1000 in the trainer component
- Eventually, tester will always generate next_action and rewards as output.

- Always eventually, next_action will be 1 (push cart to the right) when pole angle is positive

- Always eventually, next_action will be 0 (push cart to the left) when pole angle is negative

D. Analysis

Even we have mentioned the safety and liveness requirements of in this report, the design of the CartPole environment is not guaranteed to be safe because of various factors.

Firstly, the problem itself constitutes an environment that is not stable without any control (external force). This is because any disturbance to this system will not change the equilibrium point of the system if it is left alone. The reason of this is because its center of gravity is above the point of pivot.[5] Hence, the system cannot go back to its original state.

In addition to that, our design utilizes the reinforcement learning approach that learns the optimal policy from experiences. We can never guarantee in Reinforcement Learning how the algorithm would perform to a state which algorithm has not experienced before[6]. In addition to that, reinforcement learning is a hyperparameter-sensitive algorithm. Changing learning rates, or optimizers would drastically change the experiences of the algorithm. Hence, this dependency on hyperparameters also affects the safety of our design. Moreover, the random initialization of starting weights also affects how the trainer learns the optimal weights.

Overall, this system is not guaranteed to safe. What this system can achieve is the safety in the experiences the trainer comes across. What this means is that the system should be to decide optimally in the conditions it has experienced during the training phase.

IV. Conclusion

As we can see from our results obtained and literature work it is possible to design a component-based design for the Cart Pole System. This design can help us with further analysis and verification of the system.

The biggest hurdle that we encountered in this project was that we couldn't verify the entire system due to the complexity of the Reinforcement Learning Algorithm which was in P Space; making it nearly impossible to verify. However, using the component-based design certain parts apart from the RL algorithm can be easily verified and we can apply safety and liveness properties on them.

Another improvement that can be added to the System is the inclusion of a safety monitor that ensure that nothing bad will ever happen. This can help in countering the issue of RL algorithm not getting verified. Further improvements can be made by reducing the time taken by the algorithm to learn with the help of better Hardware such as GPU (like Nvidia GTX 4090) in the future. Also, we can deploy the model online as a Beta Program for users to apply random inputs and the results can be used to gain insights into the performance of our model. This is also known as online testing of Machine Learning Models.

Acknowledgment

We would like to thank our professor, Dr. Abhishek Dubey, for guiding us throughout this project.

References

1. Manrique Escobar, C.A.; Pappalardo, C.M.; Guida, D. A Parametric Study of a Deep Reinforcement Learning Control System Applied to the Swing-Up Problem of the Cart-Pole. Appl. Sci. 2020, 10, 9013. https://doi.org/10.3390/app10249013

2. Swagat Kumar Balancing a CartPole System with Reinforcement Learning https://arxiv.org/pdf/2006.04938.pdf

3. Openai. (2022, October 4). *Gym/cartpole.py at master · Openai/Gym*. GitHub. Retrieved December 17, 2022, from https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py

4. Weng, L. (2018, April 8). *Policy gradient algorithms*. Lil'Log (Alt + H). Retrieved December 17, 2022, from https://lilianweng.github.io/posts/2018-04-08-policy-gradient/

5. Morasso, P., Nomura, T., Suzuki, Y., & Zenzeri, J. (2019, March 11). *Stabilization of a cart inverted pendulum: Improving the intermittent feedback strategy to match the limits of human performance*. Frontiers. Retrieved December 17, 2022, from https://www.frontiersin.org/articles/10.3389/fncom.2019.00016/full

6. Gupta, A., & Hwang, I. (2020, October 21). *Safety verification of model based reinforcement learning controllers*. arXiv.org. Retrieved December 17, 2022, from https://arxiv.org/abs/2010.10740