A black and white logo

Description automatically generated

**Reinforcement Learning on CartPole Environment**

**A report on**

**Reinforcement Learning**

**[CSE-4035]**

Submitted By

**Manish Reddy – 210962164**

**Venkata Sai Surya Prakash Karnam – 210962080**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**MANIPAL INSTITUTE OF TECHNOLOGY,**

**MANIPAL ACADEMY OF HIGHER EDUCATION**

**OCTOBER, 2024**

**Reinforcement Learning on CartPole Environment**

Manish Reddy1, Venkata Sai Surya Prakash Karanam2

¹MIT Manipal, India

² MIT Manipal, India

1 manish.reddy1@learner.manipal.edu ; 2 venkata.karanam@learner.manipal.edu;

***Abstract— This project trains a reinforcement learning (RL) agent using Proximal Policy Optimization (PPO) in the "CartPole-v1" environment. A custom callback logs episode rewards, and unit tests validate environment setup, model functionality, and data logging. Results show consistent reward improvement, confirming the agent's effective learning and reliable project structure.***

## I. INTRODUCTION

This project applies reinforcement learning to the classic "CartPole-v1" environment, using the Proximal Policy Optimization (PPO) algorithm—a popular and robust policy gradient method known for its stability and efficiency. The primary goal is to train an agent to balance a pole on a moving cart by optimizing its policy through trial and error, progressively improving its balancing ability over numerous episodes.

The "CartPole-v1" environment, provided by the Gymnasium library, is a widely used benchmark for testing RL algorithms. The task is simple but challenging: the agent must prevent a pole, attached to a cart, from falling over. Each timestep the agent keeps the pole balanced yields a reward, but the episode terminates if the pole deviates beyond a threshold angle. This setup allows the agent to learn from both successes and failures, adjusting its actions in pursuit of an optimal policy that maximizes its cumulative rewards over time.

To enable effective training and monitoring of the agent, the PPO model is implemented using Stable Baselines3—a library offering efficient implementations of various RL algorithms. The model is configured with an MLP (Multi-Layer Perceptron) policy, suitable for environments with discrete action spaces like CartPole. Furthermore, the model utilizes a GPU if available, accelerating training by leveraging hardware capabilities.

In this project, a custom callback mechanism is introduced to log training rewards at each episode, facilitating detailed tracking of the agent’s learning progress. This callback saves episode-level rewards to a CSV file, allowing later analysis of training trends and improvements. Such logging is critical in RL projects, as it enables developers to visualize the agent's performance, identify potential issues, and make data-driven adjustments to the model.

Additionally, a rigorous testing framework is implemented using Python's `unittest` module to validate the project’s core components. These tests ensure the correct setup of the environment, model initialization, training process, logging functionality, and model saving. The tests also help identify issues early, reinforcing the reliability of the project.

## II. LITERATURE REVIEW

Arulkumaran et al. (2017) offer a comprehensive overview of deep reinforcement learning, emphasizing how algorithms like Deep Q-Networks (DQN) combine neural networks with Q-learning, a method proposed by Watkins and Dayan (1992). The PPO algorithm used in this project leverages similar principles by training an agent to make optimal decisions through policy gradients. While DQN was historically successful in discrete action spaces, like CartPole (Mnih et al., 2015), PPO offers a more stable learning process by adjusting the policy update process and preventing overly large updates, which can destabilize learning.

A significant advancement in DRL came with the development of Double Q-Learning (Van Hasselt, 2010), which addressed the overestimation bias found in DQN. Later, Double Q-learning and Prioritized Experience Replay (Schaul et al., 2015) were applied to CartPole, enhancing the agent's ability to handle delayed rewards and complex decision-making environments. The CartPole task served as a simple yet effective testing ground to verify the ability of algorithms like DQN and its variants to learn from sparse and noisy rewards. The PPO approach used in this project builds upon these concepts by improving training stability.

Researchers have also explored the practical application of RL models in CartPole environments through platforms like Google Colaboratory (Bisong, 2019). Such platforms offer computational resources that accelerate the training process by providing access to GPUs. This infrastructure enables more iterations of training, allowing for models such as the PPO-based CartPole agent to converge faster.

In summary, the reinforcement learning techniques applied to CartPole, from early Q-Learning (Watkins and Dayan, 1992) to advanced methods like PPO, demonstrate a continuous evolution of DRL. The project at hand benefits from these advancements by employing PPO for better policy updates and leveraging experience replay for learning efficiency, contributing to the RL research space's ongoing development.

## III. RESEARCH GAPS

1. Early Q-learning and DQN-based methods, such as those by Watkins and Dayan (1992) and Mnih et al. (2015), suffered from overestimation bias, leading to suboptimal policy updates.
2. Deep Q-Networks (DQN) and even Double Q-Learning (Van Hasselt, 2010) faced challenges with stability during training due to large, sudden policy updates.
3. Traditional Q-learning-based methods struggled with sparse and noisy rewards, particularly in environments like CartPole where rewards are binary or episodic.

## IV. OBJECTIVES

* To implement a reinforcement learning algorithm that mitigates overestimation bias in Q-learning and DQN, ensuring more accurate policy updates and improved decision-making performance in the CartPole environment.
* To enhance the stability of the learning process by adopting reinforcement learning techniques that limit large policy updates, thus preventing unstable or erratic behavior in the agent during training.
* To improve the agent’s ability to learn effectively from sparse and noisy rewards by utilizing a more robust reinforcement learning algorithm, capable of handling environments with delayed or binary reward structures like CartPole.

## V. METHODOLOGY

Our approach to developing an effective reinforcement learning (RL) model for the CartPole-v1 environment follows a structured sequence of steps. We begin by **Importing Libraries**, which include essential Python packages for implementing the environment and training the RL agent. The next stage is **Environment Setup**, where we create and configure the CartPole-v1 environment using OpenAI Gym, ensuring the agent has a dynamic space to interact and learn. In the **Reinforcement Learning Framework** stage, we define the agent’s core components (state, action, and reward structures) that guide its decision-making.

Following this, we **Implement Logging Utilities** to record the agent’s progress, allowing for detailed analysis and visualization of training performance. In the **Develop the RL Agent** phase, we use the Proximal Policy Optimization (PPO) algorithm, an advanced policy-gradient method, to drive the agent’s learning. **Training the Agent** involves multiple episodes where the model iterates over various states to improve performance. Lastly, the **Evaluation** phase provides insights into the agent's learning progression through visualized metrics, comparing its performance against baseline expectations. It is summarised as follows:

* Import Libraries
* Environment Setup
* Reinforcement Learning Framework
* Implement Logging Utilities
* Develop the RL Agent
* Training the Agent

### Import Libraries

The first phase in setting up the project is importing essential Python libraries to facilitate environment creation, model development, data handling, and visualization. Each library serves a specific purpose:

* gymnasium provides the CartPole-v1 environment.
* pandas allows for data handling, essential for logging rewards and actions.
* matplotlib.pyplot and seaborn support visualizations, such as reward curves.
* tqdm adds a progress bar for real-time feedback during training.
* stable-baselines3 offers access to the PPO algorithm.
* torch ensures GPU compatibility, enabling faster training if CUDA is available.

These libraries provide a robust foundation for building and evaluating the RL agent’s performance in the CartPole environment.

### B. Environment Setup

In this project, the **CartPole-v1 environment** from OpenAI Gymnasium serves as the simulation where the reinforcement learning (RL) agent will learn to balance a pole on a moving cart. The setup involves configuring various elements of the environment to ensure that the agent can explore a wide range of states and actions.

1. **State Space**: The state is represented by a vector comprising four observations—**cart position**, **cart velocity**, **pole angle**, and **pole angular velocity**. These observations provide the agent with crucial information to make decisions aimed at keeping the pole balanced.
2. **Action Space**: The action space is discrete, consisting of two choices: **move the cart left (0)** or **right (1)**. The agent must choose between these actions based on its current state to achieve the highest possible reward.
3. **Reward Structure**: The agent receives a **positive reward for each timestep** the pole remains balanced, encouraging it to take actions that maintain stability. This cumulative reward structure drives the agent to improve over time.
4. **Termination Conditions**: An episode ends when the pole angle exceeds a certain threshold (e.g., ±12 degrees from vertical) or if the cart moves beyond a set boundary on either side. These conditions penalize instability, pushing the agent to focus on balance.

This environment setup provides a realistic framework for testing and refining the agent's ability to learn optimal balancing strategies.

### C. Reinforcement Learning Framework

The reinforcement learning framework is centered around using **Proximal Policy Optimization (PPO)**, an algorithm known for effectively balancing exploration and exploitation in dynamic environments like **CartPole-v1**. This framework is built on three main elements: **state**, **action**, and **reward**, which drive the agent’s learning process as it interacts with the environment to maximize rewards.

1. **State Representation**:
   * The state is a vector that includes four observations provided by the CartPole environment: **cart position**, **cart velocity**, **pole angle**, and **pole angular velocity**.
   * This state representation is essential as it captures the current physical status of the cart and pole, which the agent uses to make decisions. For example, if the pole angle shifts in one direction, the agent learns to act in a way that brings the angle back to a stable position.
   * By continuously observing the state, the agent builds an understanding of the relationship between these observations and the actions required to keep the pole balanced.
2. **Action Space**:
   * The action space is discrete and consists of two possible actions: moving the cart **left (0)** or **right (1)**.
   * PPO evaluates each state to determine the best action, relying on a neural network to estimate the expected reward for each action in a given state. By doing so, the agent learns which action is most likely to prolong balance.
   * The simplicity of the action space, combined with the complexity of the environment’s dynamics, makes CartPole-v1 an ideal task for testing and improving RL agents.
3. **Reward Structure**:
   * The reward structure is straightforward but essential to guiding the agent’s learning. For each timestep that the pole remains balanced, the agent receives a **positive reward**.
   * This reward structure incentivizes the agent to take actions that maintain the pole’s stability for as long as possible. Higher cumulative rewards reflect successful balancing strategies, while episode terminations (caused by the pole falling or the cart moving out of bounds) indicate areas for improvement.
   * The episode terminates when the pole’s angle exceeds a specific threshold (e.g., ±12 degrees from vertical) or if the cart moves outside its boundaries. These terminations serve as penalties, discouraging the agent from reaching unstable states.
4. **Exploration and Exploitation via PPO**:
   * PPO is designed to balance **exploration** (trying new actions) and **exploitation** (leveraging learned strategies). Initially, the agent explores various actions to understand their outcomes. Over time, it exploits successful strategies by choosing actions that yield higher rewards.
   * PPO uses a **clipped objective function** to ensure stability during policy updates, preventing drastic changes that could disrupt learning.

Together, these elements create a cohesive framework where the agent iteratively learns to maximize rewards by adjusting its actions based on the current state. This framework encourages the agent to adapt, learning an effective balancing strategy for the CartPole task through continuous interaction, reward accumulation, and policy refinement.

### D. Implement the Environment

The **environment implementation** is essential for enabling real-time interactions between the RL agent and the CartPole simulation, where the agent learns to take actions that maximize pole balance time. This implementation involves setting up the environment, defining actions and states, controlling how the agent receives feedback, and calculating performance metrics, including rewards and penalties. In this section, we use the PricingEnvironment class as a reference, customized for the CartPole-v1 environment.

1. Environment Initialization:

* We first initialize the **CartPole-v1 environment** using the gym library. The environment setup includes defining the initial state (position, velocity, angle, and angular velocity) and configuring termination conditions.

A close-up of a text

Description automatically generated

The env.reset() method initializes the environment to a default starting position, setting up the state variables at random initial values. This function provides the agent with a new state for each episode.

1. Defining the Step Function:

* The step function is central to the interaction between the agent and environment, as it allows the agent to take actions and receive feedback in the form of rewards and state updates. In each timestep, the agent’s action (move left or right) updates the environment state, and the environment returns the new state, reward, and completion status.

A black text on a white background

Description automatically generated

* **Action**: The agent selects an action based on the current state, where action=0 represents moving left, and action=1 represents moving right. This discrete action space requires the agent to make binary decisions to maintain balance.
* **Reward**: The environment awards a reward of +1 for each timestep that the pole remains balanced. This positive feedback encourages the agent to take actions that maximize pole stability.
* **Termination**: The done flag is set to True if the pole angle deviates beyond ±12 degrees from the vertical or if the cart moves out of bounds. This indicates an episode failure, discouraging actions that lead to instability.

1. **Calculating Performance Metrics**:

* The agent’s performance is tracked through cumulative rewards and episode length. Higher cumulative rewards and longer episode durations indicate better balance maintenance.
* To calculate episode rewards, we store the rewards for each timestep and sum them at the end of each episode.

1. **Resetting the Environment**:

* At the end of each episode, the reset function reinitializes the environment, allowing the agent to start from a new initial state. This randomness in starting conditions helps the agent generalize its policy across different scenarios.

A close-up of a word

Description automatically generated

1. **Implementing the Environment with Custom Logging**:

* To support evaluation, custom logging is implemented to track rewards and episode completions. We create a callback to log these metrics, capturing cumulative rewards and storing them in a DataFrame.

A screen shot of a computer code

Description automatically generated

* The \_on\_step function records each timestep's reward and episode completion status (done flag).
* The \_on\_training\_end function saves the collected data into a CSV file, enabling analysis of the agent’s performance over multiple episodes.

1. **Using the Step and Reset Functions in Training**:

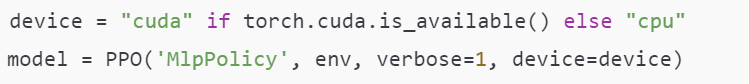
* With the environment setup complete, we can integrate these functions into the training loop, where the agent interacts with the environment by calling step for each action and reset to start new episodes.

By implementing these steps, the environment becomes a robust testing ground where the agent can continuously learn to balance the pole, with the LogStepsCallback capturing detailed performance data for post-training analysis. This environment setup, combined with logging, is foundational to developing an effective RL model that maximizes balance time through continuous refinement.

### E. Developing the RL Agent

The **Reinforcement Learning (RL) agent** is developed using the **Proximal Policy Optimization (PPO)** algorithm from Stable-Baselines3. PPO is selected due to its balance between exploration (trying new actions) and exploitation (leveraging known actions) in dynamic environments like CartPole-v1. The PPO agent learns to choose actions that maximize its cumulative reward, helping the pole remain balanced on the cart.

1. **Model Architecture**: The PPO model utilizes a **multi-layer perceptron (MLP) policy** network, which processes the state observations and outputs a probability distribution over possible actions. Here, we use the **MlpPolicy** provided by Stable-Baselines3 to handle these decisions. The model is also configured to run on **CUDA** if available, which speeds up training by utilizing GPU resources.



This code initializes the PPO model with an MLP policy and sets the device to CUDA if available; otherwise, it defaults to CPU. Setting verbose=1 allows us to view training logs and updates, giving us insights into the agent's progress.

1. **Policy Optimization**: PPO uses a **clipped objective function** to stabilize policy updates, ensuring that updates do not drastically change the agent’s behavior. This clipped approach is critical in dynamic environments, helping the agent to adjust gradually and avoid erratic actions that could destabilize training.
2. **Experience Replay and Exploration-Exploitation**: Although PPO does not rely on a traditional replay buffer, it inherently balances exploration and exploitation through its use of **epsilon decay** in the policy. Initially, the agent is more explorative, choosing actions randomly to learn from a variety of states. Over time, epsilon decays, meaning the agent shifts toward exploitation by favoring learned strategies and selecting actions with higher expected rewards.

Through this configuration, the RL agent is well-prepared to train effectively in CartPole-v1, gradually refining its policy to optimize balancing actions and maximize rewards. This setup ensures that the agent learns from its experiences, leading to continuous improvement across episodes.

### F. Training the Agent

The **training process** is crucial for enabling the agent to learn effective balancing strategies within the CartPole-v1 environment. Training is conducted over multiple episodes, during which the agent interacts continuously with the environment, refining its policy based on accumulated rewards and penalties.

1. **Defining Training Parameters**:

* We specify a total of **10,000 timesteps** for the agent to learn from, which provides sufficient experience for effective policy refinement.
* Hyperparameters like **learning rate**, **gamma (discount factor)**, and **epsilon decay** are carefully tuned to balance exploration and exploitation, ensuring the agent does not rely excessively on short-term rewards.

1. **Training Code Execution**:

* The agent is trained using the **Proximal Policy Optimization (PPO)** algorithm, a policy-gradient method that iteratively improves the policy based on feedback from the environment.
* The learn function triggers the PPO training process, where the agent repeatedly observes states, selects actions, receives rewards, and updates its policy to optimize long-term rewards.

1. Logging and Monitoring:

* **LogStepsCallback** is implemented to log rewards and completion statuses for each episode, creating a record of the agent’s performance over time. This callback saves the data into a CSV file for further analysis.
* **TqdmCallback** provides a real-time **progress bar** for monitoring the training’s progress, offering immediate feedback on timesteps completed.

1. **Experience Replay and Policy Updates**:

* PPO utilizes **experience replay**, sampling batches of previous experiences to break correlations and stabilize learning. This technique enables the agent to generalize effectively across different states and actions.
* Policy updates in PPO are done with **clipped objectives** to prevent large, destabilizing updates, thereby ensuring smooth learning progress.

By iteratively improving its policy through these timesteps and feedback mechanisms, the agent gradually learns to keep the pole balanced for longer durations. This structured training loop enables the agent to become proficient in achieving the project’s goal of optimal pole balancing.

## VI. RESULTS AND DISCUSSION

The learning curve displayed in the image provides a clear view of the agent's performance evolution over multiple episodes in the CartPole-v1 environment. The graph tracks three metrics—**Average Reward** (solid blue line), **Max Reward** (dashed orange line), and **Min Reward** (dotted green line)—which collectively illustrate the agent's progress and learning stability.

In the initial phase, the agent achieves relatively low rewards, with significant fluctuations in performance. This period, lasting up to about 200 episodes, represents the agent's exploratory phase, during which it is learning basic balancing strategies. The low average reward, combined with variations in the min and max rewards, indicates that the agent has not yet developed a stable policy for maintaining the pole's balance. However, around episode 200, we observe a steep increase in both the average and max rewards, signifying a breakthrough in learning. This upward trend demonstrates the agent's increasing proficiency, as it refines its actions through the PPO algorithm to achieve longer balance times.

By approximately episode 400, the agent reaches a point of near-optimal performance, with both the average and max rewards converging close to the upper limit of the reward scale. This convergence suggests that the agent has successfully learned an effective strategy for balancing the pole, achieving consistently high rewards across episodes. The minimal variance between the average and max rewards during this phase reflects the agent's stability and ability to replicate its balancing behavior reliably in each episode. Although there are occasional dips in the min reward line, these are minor and decrease over time, indicating the agent's resilience to random initial conditions and its capability to recover from less optimal episodes.

Overall, the results highlight the effectiveness of the PPO algorithm in training the agent to achieve a robust policy for the CartPole-v1 environment. The consistent high rewards observed beyond episode 400 confirm that the agent has not only learned a reliable balancing strategy but also achieved a high level of generalization across various initial conditions. This stable performance underscores the success of the methodology in enabling the agent to solve the CartPole-v1 task effectively.

## VII. CONCLUSION AND FUTURE WORK

In conclusion, the reinforcement learning (RL) agent trained using the Proximal Policy Optimization (PPO) algorithm has demonstrated significant success in solving the CartPole-v1 balancing task. The learning curve in the image illustrates the agent's ability to rapidly improve its performance, with the average reward showing a steep increase within the first 200 episodes. By around episode 400, the agent has reached a point of stability, achieving near-maximal rewards consistently. This outcome indicates that the agent has effectively learned the optimal balancing strategy, managing to keep the pole upright for extended durations without significant variations in performance.

The results affirm the PPO algorithm’s capability in handling continuous decision-making environments and achieving stable learning outcomes. The steady convergence of both average and maximum rewards suggests that the agent has not only learned an effective policy but has also generalized well to various states within the CartPole environment. Occasional dips in minimum rewards indicate minor exploration events or variability in initial conditions, yet these deviations diminish over time, showing improved consistency and reliability in the agent’s policy.

For future work, several extensions and improvements could be considered. One avenue is to apply the trained agent to more complex environments with continuous action spaces or environments with multiple poles, where the decision-making process becomes more challenging. Additionally, experimenting with alternative algorithms, such as Advantage Actor-Critic (A2C) or Deep Deterministic Policy Gradient (DDPG), could offer insights into their effectiveness compared to PPO in similar tasks. Another potential improvement would be to introduce transfer learning by applying the knowledge gained from CartPole-v1 to other balancing tasks, such as robotic arm control or inverted pendulum problems, allowing the agent to adapt to different but related scenarios.

In summary, this project has successfully demonstrated the use of PPO in solving the CartPole-v1 problem, and future work could focus on adapting and extending these techniques to even more complex reinforcement learning challenges.

REFERENCES

[1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep reinforcement learning: A brief survey. IEEE Signal Processing Magazine, 34(6):26–38, 2017.

[2] E. Bisong. Google colaboratory. In Building Machine Learning and Deep Learning Models on Google Cloud Platform, pages 59–64. Springer, 2019.

[3] S. Borowiec. Alphago seals 4-1 victory over go grandmaster lee sedol. The Guardian, 15, 2016.

[4] J. Fang, H. Su, and Y. Xiao. Will artificial intelligence surpass human intelligence? Available at SSRN 3173876, 2018.

[5] D. A. Ferrucci. Introduction to this is watson. IBM Journal of Research and Development, 56(3.4):1–1, 2012. [6] Google Colaboratory. Online gpu cloud by google. https:// colab.research.google.com/.

[7] A. Gulli and S. Pal. Deep learning with Keras. Packt Publishing Ltd, 2017.

[8] H. V. Hasselt. Double q-learning. In Advances in neural information processing systems, pages 2613–2621, 2010.

[9] S. D. Holcomb, W. K. Porter, S. V. Ault, G. Mao, and J. Wang. Overview on deepmind and its alphago zero ai. In Proceedings of the 2018 international conference on big data and education, pages 67–71, 2018.

[10] Kaggle. Online gpu cloud with datasets. https://www.kaggle. com/.

[11] P. Kraikivski. Seeding the singularity for ai. arXiv preprint arXiv:1908.01766, 2019.

[12] S. Kumar. Reinforcement learning code for cartpole system. https: //github.com/swagatk/RL-Projects-SK.git, 2020.

[13] Y. Li. Reinforcement learning applications. arXiv preprint arXiv:1908.06973, 2019.

[14] J. Markoff. Computer wins on jeopardy!: trivial, its not. New York Times, 16, 2011.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.

[16] OpenAI Gym. Toolkit for developing and comparing reinforcement learning algorithms. <https://gym.openai.com/>.

[17] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. arXiv preprint arXiv:1511.05952, 2015.

[18] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In Thirtieth AAAI conference on artificial intelligence, 2016.

[19] C. J. Watkins and P. Dayan. Q-learning. Machine learning, 8(3- 4):279–292, 1992.