Balancing Stick on a Moving Cart

REINFORCEMENT LEARNING

PROJECT REPORT

2024-2025

TABLE OF CONTENT

Abstract

This project delves into the formidable yet classical control problem of balancing an inverted pendulum on a moving cart, a challenge that intrinsically mirrors real-world complexities in robotics, autonomous control systems, and the broader field of artificial intelligence. The fundamental objective is to meticulously design and implement an intelligent agent capable of learning to effectively stabilize this inherently unstable system using sophisticated reinforcement learning techniques. This work introduces and rigorously evaluates a novel hybrid reinforcement learning approach that ingeniously combines the efficiency of classical Hill Climbing with the powerful generalization capabilities of Deep Reinforcement Learning (specifically, Double Deep Q-Networks or DDQN).

A cornerstone of this proposed methodology involves strategically pre-filling the replay memory of the Deep Q-Network using a curated set of experiences meticulously gathered from a pre-trained Hill Climbing policy. This "warm-start" mechanism is designed to significantly accelerate the learning process of the DDQN and substantially enhance its convergence stability. Furthermore, the project incorporates custom reward shaping techniques, meticulously crafted to guide the learning agent more efficiently towards optimal behavior and ensure a faster, more robust convergence. The core contribution of this research lies in its comprehensive comparative analysis, where the learning behavior and ultimate performance of three distinct training methodologies – pure Hill Climbing, a standalone DDQN, and the innovative DDQN with Hill Climbing pre-fill – are rigorously evaluated, visualized side-by-side through custom animations, and quantitatively assessed through performance graphs. This foundational research also explicitly lays the groundwork for seamless future real-world hardware implementation.

1. Introduction

Aim: The overarching aim of this project is to meticulously develop, implement, and comprehensively evaluate advanced reinforcement learning agents tailored specifically for the quintessential CartPole balancing problem. A particular emphasis is placed on exploring and validating a novel hybrid approach that intelligently merges established classical control methods with cutting-edge deep reinforcement learning techniques. This integration is designed to synergistically enhance both the learning efficiency and the inherent stability of the control policy, thereby addressing common challenges faced by traditional single-paradigm approaches.

Objective: To achieve the stated aim, the project is structured around several distinct and measurable objectives:

- **Implementation of Hill Climbing:** To design, code, and thoroughly train a classical Hill Climbing policy, serving as a foundational baseline, for effective control within the CartPole environment. This involves fine-tuning its parameter exploration and convergence.
- **Implementation of Double Deep Q-Network (DDQN):** To construct, implement, and robustly train a sophisticated Double Deep Q-Network (DDQN) agent, leveraging neural networks for Q-value approximation, to tackle the CartPole stabilization task.
- **Hybrid Approach Exploration:** To conceptualize, implement, and rigorously evaluate a novel hybrid training paradigm where the DDQN agent's experience replay memory is strategically pre-populated with high-quality experiences generated by the previously trained Hill Climbing policy. This aims to provide a more informed starting point for deep learning.
- **Custom Reward Shaping Design:** To meticulously design and implement bespoke custom reward shaping strategies. These strategies are engineered to provide more informative feedback to the learning agents, thereby accelerating the convergence process and bolstering training stability.
- **Comparative Analysis and Visualization:** To conduct a comprehensive comparative analysis of the learning trajectories, ultimate performance metrics, and convergence speeds across all three implemented agent types: the pure Hill Climbing agent, the standalone DDQN agent, and the proposed hybrid DDQN agent with Hill Climbing pre-fill. This analysis will be complemented by insightful visualizations to elucidate the learning dynamics.
- **Foundation for Physical Implementation:** To establish a clear conceptual and architectural foundation that facilitates the future transition and physical implementation of the learned control policies on actual robotic hardware, including the integration of sensors and actuators.

Scope of Work: The current scope of this research project primarily encompasses the simulation-based environment of the CartPole problem, leveraging the widely adopted OpenAI Gym framework. Within this simulated domain, the project focuses on the rigorous implementation and thorough evaluation of the specified reinforcement learning algorithms, namely Hill Climbing and Double Deep Q-Networks. A significant portion of the work is dedicated to developing and testing the innovative hybrid training pipeline that integrates these two distinct methodologies. Furthermore, the scope includes an in-depth comparative analysis of the experimental results obtained from each training scenario. Sophisticated visualization tools, including performance graphs and real-time animations of the agent's behavior, are integral to understanding and presenting the learning progress. It is important to note that while the project lays the theoretical and computational groundwork, the physical development and testing on real-world hardware are designated as future work, thereby limiting the current scope to the robust simulation environment.

Contributions of the Research This research makes several significant contributions to the field of reinforcement learning and control, addressing key challenges and offering novel perspectives:

- **Novel Combined Classical and Deep RL Approach:** A pioneering hybrid methodology is introduced that ingeniously merges the strengths of a classical control algorithm, Hill Climbing, with the advanced capabilities of Deep Reinforcement Learning (DDQN). This directly addresses a critical research gap identified in existing literature, where most prior studies tend to focus exclusively on either classical control or deep learning methods in isolation. Our integrated approach seeks to harness the explorative efficiency of Hill Climbing for initial policy guidance and the powerful generalization of DDQN for long-term optimal control.
-
- **Comprehensive Side-by-Side Comparison and Visualization:** A distinct contribution lies in providing a direct, explicit, and visually compelling comparison of the learning behavior and performance across disparate training methodologies. By presenting Hill Climbing, vanilla DDQN, and the proposed hybrid DDQN + Hill Climbing pre-fill strategies side-by-side, the project offers unparalleled insights into their respective strengths, weaknesses, and convergence characteristics under identical environmental conditions. The custom visualizations further enhance the clarity of these comparisons.

- **Effective Custom Reward Shaping:** The project implements and evaluates custom reward shaping techniques specifically engineered to accelerate the convergence speed of the learning agents and significantly enhance the stability of the training process. By providing more informative and finely-grained feedback than the default environment rewards, these techniques guide the agents more efficiently towards desired behaviors, reducing training time and preventing oscillations in performance.

- **Implicit Curriculum-Style Training:** Although not explicitly designed as a multi-stage curriculum, the project implicitly adopts a curriculum-style training paradigm. By first training a simpler, more interpretable model (Hill Climbing) to gather initial, reasonably good experiences, and then leveraging these experiences to "boot-strap" the training of a more complex deep learning model (DDQN), the approach mimics a progression from easier to more complex learning tasks, a known technique for

improving deep RL performance.

- **Solid Foundation for Hardware Implementation:** Beyond theoretical and simulated advancements, this research meticulously establishes a robust conceptual framework and practical code base that serves as a direct and actionable foundation for future physical implementation. The clear architectural design and the use of widely available libraries facilitate the eventual transfer and testing of the learned policies on real-world robotic hardware, incorporating components like IMUs and motors,

2. Literature Review

The Cart-Pole problem, a canonical control task in reinforcement learning, involves balancing an inverted pendulum on a horizontally moving cart. This seemingly simple problem serves as a robust benchmark for evaluating various control and learning algorithms, simulating complex challenges encountered in robotics, aerospace engineering, and autonomous systems. Over the years, a multitude of approaches have been devised to address this problem, each with its inherent strengths and limitations.

Existing Solutions and Their Limitations:

- **Rule-based Controllers (PID, LQR):** Traditional control engineering methodologies such as Proportional-Integral-Derivative (PID) controllers and Linear-Quadratic Regulators (LQR) can be highly effective in solving the CartPole problem under idealized conditions. However, their primary limitation lies in the necessity for precise system modeling, accurate parameter identification, and meticulous tuning, which can become exceedingly challenging and time-consuming for systems exhibiting high dimensionality, non-linear dynamics, or unmodeled uncertainties. They lack the adaptive learning capabilities inherent in reinforcement learning.

- **Hill Climbing:** As a direct policy search method, Hill Climbing represents one of the simpler reinforcement learning algorithms. It operates by iteratively adjusting policy parameters (in this case, the weight matrix 'w') based on observed rewards, seeking to climb the "hill" of performance. While straightforward in implementation, Hill Climbing often struggles with large or complex state spaces and is prone to getting trapped in suboptimal local optima, making it difficult to guarantee global optimality. The provided code incorporates an adaptive noise scaling mechanism, which is a common enhancement to mitigate this by allowing for more dynamic exploration.

- **Q-Learning:** A foundational model-free reinforcement learning algorithm, Q-Learning learns an action-value function (Q-function) that estimates the expected utility of taking a given action in a given state. Its effectiveness is inherently limited to environments with discrete and relatively small state and action spaces due to its reliance on explicitly storing Q-values in a table. This tabular representation becomes computationally intractable and memory-intensive for environments with continuous state variables, like the CartPole, making it impractical without discretization.

- **Deep Q-Networks (DQN):** Representing a significant leap in reinforcement learning, Deep Q-Networks (DQN) extend Q-Learning by replacing the traditional Q-table with a deep neural network to approximate the Q-values. This architectural change enables DQN to generalize effectively to environments with vast or continuous state spaces by learning complex, non-linear relationships. Key innovations like experience replay and target networks further stabilize the training of these deep neural networks.

- **Deep Deterministic Policy Gradient (DDPG):** DDPG is an advanced actor-critic, model-free algorithm specifically engineered for continuous action spaces. It learns both a deterministic policy (actor) and a Q-function (critic) concurrently. While powerful for continuous control, it adds a layer of complexity compared to methods.

Reviewed Research Papers:

A comprehensive literature review was conducted to understand the current landscape of reinforcement learning applications to the CartPole problem and related control challenges. The following papers were instrumental in shaping the project's direction and highlighting relevant research gaps:

- **A Huber Reward Function-Driven Deep RL Solution for Cart-Pole (Mishra & Arora, 2023):** This research explored the application of DQN and Double DQN for CartPole stabilization. A key finding was that by integrating a Huber reward function, they achieved notably improved stability in the learning process compared to a vanilla DQN setup. This paper underscores the significant impact of judicious reward shaping on the robustness of deep reinforcement learning algorithms.

- **Barrier Functions Inspired Reward Shaping for RL (Nilaksh et al., 2024):** This work delved into innovative reward shaping techniques, specifically leveraging barrier functions in conjunction with both DQN and Proximal Policy Optimization (PPO) algorithms. Their results demonstrated a remarkable 1.4–2.8 times faster convergence rate and a considerable 50–60% reduction in actuation efforts. This study directly supports the premise of our project regarding the critical role of custom reward shaping in accelerating learning and enhancing efficiency.

- **QF-tuner: Auto Hyperparameter Tuning (Jumaah et al., 2024):** This paper focused on the intricate challenge of hyperparameter optimization within quantum reinforcement learning. By combining Q-learning with the FOX optimizer, they reported substantial improvements, including a +57% increase in performance and a –20% reduction in training time. This highlights the importance of effective tuning, a consideration for any complex RL system.

- **Lyapunov Design for Robotic RL (Westenbroek et al., 2023):** This research showcased a compelling integration of reinforcement learning principles with Control Lyapunov Function (CLF) cost shaping. The approach led to significantly reduced data requirements for learning and, crucially, successful validation on real-world robotic hardware. This work is highly relevant to our project's long-term goal of transitioning from simulation to physical implementation.

- **The Quantum Cartpole: Non-linear RL Benchmark (Meinerz et al., 2024):** This paper explored the application of model-free reinforcement learning to a "Quantum CartPole," a non-linear variant of the classic problem. Their findings demonstrated superior control capabilities in highly non-linear regimes, suggesting the adaptability of RL even to highly complex, non-classical systems.

- **Deep RL Control of Quantum Cartpoles (Wang, Ashida & Ueda, 2023):** Further extending the quantum CartPole concept, this study applied Deep RL techniques to control such systems, achieving performance levels that matched or even surpassed traditional classical control methods. This reinforces the power of deep learning for complex control tasks.

- **RL with Human Feedback: CartPole Case Study (Parkhi et al., 2024):** This research investigated the acceleration of policy learning through the incorporation of human-in-the-loop feedback within a neural policy framework. It highlights the potential for combining human intuition with algorithmic learning to improve efficiency.

- **Cart-Pole Application as Neuromorphic Benchmark (MDPI Group, 2023):** This study utilized Spiking Neural Networks (SNNs) as a control mechanism for the CartPole, showcasing the potential of neuromorphic computing architectures for efficient reinforcement learning, achieving an impressive average of ~14,970 steps per episode.

- **REINFORCE, A2C/A3C Policy Gradients (CartPole) (Jovanovic, 2023):** This work conducted a comparative analysis of various policy gradient methods applied to the CartPole problem, concluding that Asynchronous Advantage Actor-Critic (A3C) generally outperformed its synchronous counterpart (A2C), as well as the basic Actor-Critic and REINFORCE algorithms, in terms of learning efficiency.

- **Generalizable RL via Causality-Guided Representations (Hafner et al., 2024):** This research explored methods to enhance the generalizability of reinforcement learning agents across different domains by incorporating causality-guided representations. Their study, utilizing algorithms like DQN, Dreamer, EfficientZero, and SPR, demonstrated robust performance even under significant domain shifts.

- **Distributional Soft Actor-Critic (DSAC) for Continuous Control (Duan et al., 2023):** This paper introduced DSAC-T, a novel variant of the Soft Actor-Critic (SAC) algorithm, specifically tailored for continuous control problems. The results indicated that DSAC-T consistently outperformed the standard SAC in continuous CartPole environments, highlighting advancements in algorithms for continuous action spaces.

- **Tutorial: Balancing CartPole with DQN Variants (Swagat Kumar, 2023):** This tutorial provided a comparative study of various DQN enhancements, including Double DQN, Dueling DQN, and Prioritized Experience Replay (PER). The findings suggested that PER yielded the most favorable performance for the CartPole task, emphasizing the impact of experience replay mechanisms.

Research Gap Addressed: A recurring observation from the comprehensive literature review is that a significant majority of existing research papers tend to employ either classical control methods or deep reinforcement learning algorithms in isolation when addressing the CartPole problem. There is a noticeable paucity of studies that systematically combine these two powerful paradigms. Our project directly confronts and bridges this research gap by proposing and rigorously evaluating a novel hybrid approach that integrates the classical Hill Climbing algorithm with Double Deep Q-Networks (DDQN). This integration is designed to leverage the distinct advantages of both methodologies: the initial exploration efficiency of Hill Climbing and the sophisticated learning and generalization capabilities of DDQN. By providing a "warm start" to the deep learning agent using experiences from a simpler, effective policy, we aim to demonstrate a pathway to potentially faster and more stable convergence, thereby offering a more robust solution framework.

3. Hardware and Software Requirements

Effective execution and training of reinforcement learning models, especially those involving deep neural networks, necessitate a suitable computational environment. The following sections detail the hardware and software prerequisites for running and developing this CartPole balancing project.

Hardware Requirements While the provided Python code does not explicitly dictate minimum hardware specifications, successful and efficient execution of reinforcement learning simulations, particularly those involving deep learning components, typically benefits from a specific set of hardware resources:

- **Processor (CPU):** A modern multi-core Central Processing Unit (CPU), such as an Intel i5/i7 (8th generation or newer) or an AMD Ryzen 5/7, is highly recommended. The CPU is essential for handling general program logic, environment interactions, and non-GPU-accelerated computations. A higher core count and clock speed will contribute to faster overall simulation runtime, especially for the non-learning parts of the code.

- **System Memory (RAM):** A minimum of 8GB of Random Access Memory (RAM) is advisable. However, for smoother operation and to prevent bottlenecks during replay memory operations or large batch processing in PyTorch, 16GB or more is strongly recommended. Sufficient RAM ensures that the operating system, development environment, and Python script can run concurrently without significant performance degradation.

- **Graphics Processing Unit (GPU) (Optional but Highly Recommended):** An NVIDIA GPU with CUDA support is not strictly mandatory for the CartPole environment due to its simplicity, but it is *highly recommended* for significantly accelerating the training of the Deep Q-Networks. PyTorch, the deep learning framework used, is optimized for CUDA. The code explicitly checks for torch.cuda.is_available(), indicating its preference for GPU acceleration. Training on a CPU for complex deep learning tasks can be prohibitively slow; a GPU enables parallel processing of computations, drastically reducing training times.

Software Requirements The project is built upon a standard Python ecosystem, utilizing several widely adopted libraries for reinforcement learning, numerical computation, visualization, and deep learning.

- **Operating System:** The code is designed to be cross-platform compatible and can run on major operating systems, including Windows 10/11, macOS, or various Linux distributions (e.g., Ubuntu, Fedora).

- **Python Interpreter:** Python version 3.7 or newer is required. The code leverages syntax and library versions compatible with recent Python releases.

- **Key Python Libraries:**
    - numpy: The fundamental package for scientific computing with Python. It is

used extensively for numerical operations, array manipulation, and handling the policy weights in the Hill Climbing algorithm. The np.bool8 patch ensures compatibility across different NumPy versions.
    - gym (OpenAI Gym): The industry-standard toolkit for developing and comparing reinforcement learning algorithms. It provides the 'CartPole-v1' environment, offering a standardized interface for state observations, action spaces, and reward structures.
    - matplotlib.pyplot: A comprehensive library for creating static, animated, and interactive visualizations in Python. It is used in this project to generate and display the crucial reward and accuracy graphs, allowing for visual analysis of training progress across different agents.
    - pygame: A set of Python modules designed for writing video games. In this project, pygame is used in conjunction with env.render(mode='human') to provide a real-time visual representation of the CartPole environment, allowing for qualitative observation of the trained agent's behavior. It also facilitates the overlay display of episode information during visualization.
    - torch (PyTorch): An open-source machine learning framework, widely used for deep learning applications. It forms the backbone of the Deep Q-Network (QNetwork class) and provides tools for building neural networks, defining loss functions, and optimizing models.
    - torch.nn: The neural network module within PyTorch, providing predefined layers (like Linear, ReLU), activation functions, and utilities for constructing complex network architectures for the QNetwork.
    - torch.optim: The optimization module in PyTorch, offering a wide range of optimization algorithms (e.g., Adam, SGD) to update model parameters during training. optim.Adam is specifically utilized here.
    - collections.deque: A high-performance deque (double-ended queue) data structure from Python's collections module. It is critically used to implement the experience replay memory, efficiently storing and retrieving experiences for batch learning in DDQN.
    - random and time: Standard Python libraries used for generating random numbers (e.g., for epsilon-greedy exploration, batch sampling) and for introducing time delays in the visualization to control playback speed.

- **Development Environment:** Any integrated development environment (IDE) compatible with Python, such as VS Code, PyCharm, or Google Colab/Jupyter Notebooks, can be used for coding, debugging, and running the project.

Summary: The project's computational requirements are manageable, allowing it to be executed on a standard personal computer. However, to leverage the full potential of deep reinforcement learning and significantly reduce training times, particularly for the DDQN components, a system equipped with a dedicated NVIDIA GPU with CUDA support is highly advantageous. The software stack is entirely based on open-source Python libraries, ensuring accessibility and ease of setup.

4. Proposed Methodology

Overview: Our proposed methodology represents a synergistic integration of a classical reinforcement learning technique, Hill Climbing, with the robust capabilities of Double Deep Q-Networks (DDQN) to effectively and efficiently solve the intricate CartPole balancing problem. The fundamental premise behind this hybrid approach is to strategically harness the initial exploratory strengths and rapid policy discovery of the Hill Climbing algorithm to provide a "warm start" for the more sophisticated DDQN agent. This is achieved by pre-filling the DDQN's experience replay memory with meaningful and relevant transitions generated by the trained Hill Climbing policy. This innovative pre-population mechanism is designed to circumvent the common initial challenges faced by deep reinforcement learning agents, such as sparse rewards and slow exploration from scratch, thereby accelerating the overall learning process and enhancing the stability and convergence speed of the DDQN training. Complementing this, custom reward shaping techniques are meticulously incorporated throughout the learning process to provide more informative feedback signals, further guiding the agent towards optimal behavior.

System Modules

The entire system is modularly designed, comprising several interconnected components, each responsible for a specific aspect of the reinforcement learning pipeline:

1. **Environment (CartPole-v1):**
    - **Description:** The project utilizes the standard CartPole-v1 environment provided by OpenAI Gym. This environment serves as the simulated reality for the agents, offering a consistent and well-defined control problem.
    - **Functionality:** It provides the agent with observations of the current state, which typically include four continuous values: cart position, cart velocity, pole angle, and pole angular velocity. In response, it accepts discrete actions from the agent (moving the cart left or right, represented by 0 or 1). The environment calculates the resultant next state, a scalar reward (typically +1 for each timestep the pole remains upright), and a done flag indicating episode termination.
    - **Role:** Acts as the interactive learning ground where agents gather experiences and refine their policies.

2. **Hill Climbing Policy:**
    - **Description:** This module implements a simple yet effective linear policy. The policy's behavior is dictated by a single weight matrix, self.w, which directly maps the high-dimensional state observations to a preference for each possible action.
    - **__init__(self, state_dim, action_dim):** Initializes the weight matrix w with random values. The dimensions state_dim and action_dim correspond to the input (state vector size) and output (number of possible actions) of the linear transformation.
    - **act(self, state):** Given the current state, this method computes the dot product of the state vector and the w matrix (np.dot(state, self.w)). This results in a vector where each element represents a score or preference for a particular action. The np.argmax function is then used to select the action with the highest score, representing the policy's deterministic choice.

- **Training (train_hill_climbing function):** The training process is iterative. For each episode, the agent interacts with the environment, and the total reward is accumulated. If the current episode's total reward surpasses the best_R (the highest reward achieved so far), the current w matrix is saved as best_w, and a key adaptive noise scaling mechanism is employed: the noise_scale is reduced (halved, but not below 1e-3). This promotes finer tuning around a promising policy. Conversely, if the performance degrades, the w is reset to the best_w found, and noise_scale is increased (doubled, up to 2) to encourage broader exploration. This adaptive noise allows the Hill Climbing algorithm to dynamically adjust its exploration-exploitation balance.
- **Role:** Provides a straightforward, interpretable baseline and is crucial for generating initial high-quality experiences for the DDQN pre-fill.

3. QNetwork (Neural Network):

- **Description:** This is the core neural network architecture for approximating the Q-values in the DDQN agent. It is a standard feedforward deep neural network implemented using PyTorch's nn.Module.
- **__init__(self, state_dim, action_dim):** Defines the network's structure. self.net is a nn.Sequential container, simplifying the definition of a sequence of layers. It consists of:
  - nn.Linear(state_dim, 64): An input layer that maps the state_dim (4 features for CartPole) to 64 neurons.
  - nn.ReLU(): A Rectified Linear Unit activation function, introducing non-linearity to the network, allowing it to learn complex mappings.
  - nn.Linear(64, 64): A hidden layer with 64 neurons.
  - nn.ReLU(): Another ReLU activation.
  - nn.Linear(64, action_dim): An output layer that maps the 64 hidden neurons to action_dim (2 actions for CartPole) Q-values. Each output neuron corresponds to the estimated Q-value for taking a specific action in the given state.
- **forward(self, x):** Defines the forward pass of the neural network. When an input tensor x (representing a state or batch of states) is passed, it flows sequentially through the layers defined in self.net, producing the Q-value estimates for each action.
- **Role:** Acts as the function approximator for the Q-values, enabling the DDQN to handle continuous state spaces.

4. DDQNAgent:

- **Description:** This class encapsulates the entire Double Deep Q-Network algorithm, including its network structures, optimization routines, and experience management.
- **__init__(self, state_dim, action_dim):**
  - self.device: Automatically detects and sets the computational device to CUDA (GPU) if available, otherwise defaults to CPU. This optimizes training speed.
  - self.qnet: The online Q-network, whose parameters are actively trained.
  - self.target_net: A separate, periodically updated target Q-network. It helps stabilize training by providing stable target Q-values, preventing oscillations that can arise if the same network is used for both predicting and targets.

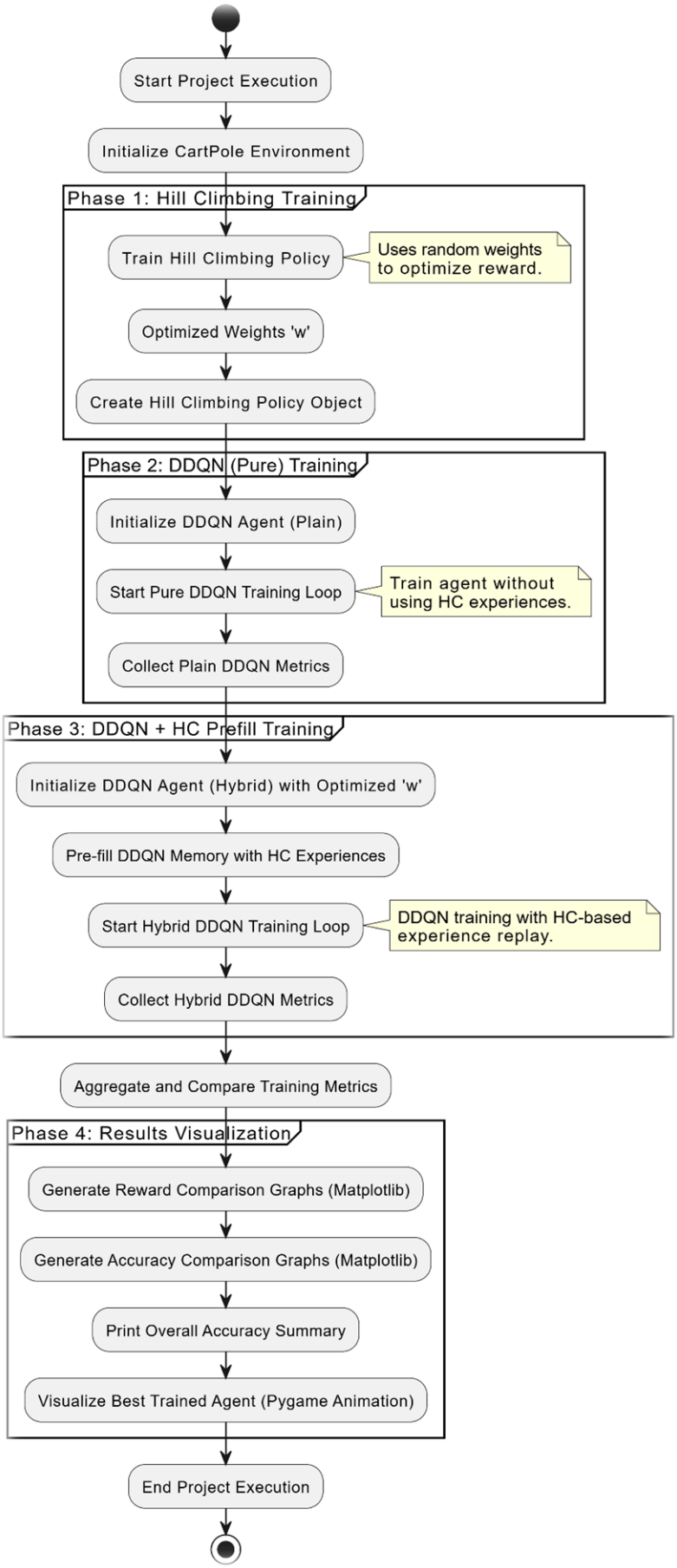
        - self.optimizer: optim.Adam is chosen as the optimizer, known for its efficiency and good performance in many deep learning tasks, with a learning rate (lr) of 1e-3.
        - self.loss_fn: nn.MSELoss() (Mean Squared Error) is used as the loss function, quantifying the difference between the predicted Q-values and the target Q-values.
        - self.memory: A collections.deque (double-ended queue) with a maxlen of 10,000. This is the replay memory, crucial for breaking the temporal correlations in sequential experiences by allowing random sampling of past transitions. This supports off-policy learning.
        - self.batch_size: The number of experiences sampled from replay memory for each learning step (64).
        - self.gamma: The discount factor (0.99), determining the importance of future rewards.
        - self.epsilon, self.eps_decay, self.eps_min: Parameters for the epsilon-greedy exploration strategy. epsilon starts at 1.0 (full exploration) and decays by eps_decay (0.995) after each episode, down to a minimum eps_min (0.01) to ensure continued exploration.
        - self.update_target(): Called at initialization to synchronize target network weights.
    - **act(self, state):** Implements the epsilon-greedy action selection. With probability epsilon, it chooses a random action (exploration). Otherwise, it converts the state to a PyTorch tensor, passes it through self.qnet to get Q-values, and selects the action with the highest Q-value (argmax().item()) (exploitation).
    - **step(self, state, action, reward, next_state, done):** Stores the current transition (state, action, reward, next_state, done) into the replay memory. If the memory contains enough experiences (at least batch_size), it triggers the learn() method.
    - **learn(self):**
        - Randomly samples a batch_size number of transitions from self.memory.
        - Converts these samples into PyTorch tensors and moves them to the appropriate device (CPU/GPU).
        - Calculates q_values for the current states using self.qnet.
        - Calculates next_actions by finding the argmax of self.qnet for next_states (the "double" in DDQN – using the online network to select the next action).
        - Calculates next_q_values using self.target_net for the next_states and the next_actions chosen by self.qnet. This decouples action selection from value estimation, improving stability.
        - Computes the target Q-values using the Bellman equation: reward + (1 - done) * gamma * next_q_values. (1 - done) ensures that if the episode terminates, there is no future reward.
        - Calculates the loss between q_values and target using self.loss_fn.
        - Performs backpropagation: self.optimizer.zero_grad() clears gradients, loss.backward() computes new gradients, and self.optimizer.step() updates self.qnet's parameters.
        - Decays self.epsilon to reduce exploration over time.
    - **Role:** The primary learning agent responsible for training the deep neural network to learn an optimal policy for the CartPole.

5. **Replay Memory Pre-filling Module (prefill_memory function):**
    - **Description:** This crucial component implements the hybrid aspect of the methodology. It's a function designed to populate the DDQN agent's replay memory *before* its main training phase begins.
    - **Functionality:** It orchestrates a specified number of episodes (defaulting to 50) where the *already trained Hill Climbing policy* interacts with the environment. For each step of these interactions, the resulting transition (state, action, reward, next_state, done) is directly added to the DDQN agent's memory via its agent.step() method.
    - **Role:** Provides a significant "head start" to the DDQN. Instead of starting with an empty memory or relying solely on random exploration to populate it, the DDQN begins with a buffer full of meaningful and often successful experiences, thereby accelerating its initial learning and promoting more stable convergence.

6. **Training Orchestrator (train_ddqn and train_hill_climbing functions):**
    - **Description:** These functions manage the entire training loop for their respective agents. They handle episode progression, interaction with the environment, agent updates, and performance tracking.
    - **Functionality:** For each episode, they reset the environment, simulate agent-environment interactions for a fixed number of steps (or until done), accumulate rewards, calculate accuracy, and call the agent's step and update_target (for DDQN) methods. They also print progress updates to the console, showing current reward, accuracy, and epsilon (for DDQN), and check for convergence criteria (e.g., average reward over last 100 episodes).
    - **Role:** Coordinates the training process for each of the three experimental scenarios, ensuring consistent evaluation.

7. **Visualization Module (watch_trained_agent function and matplotlib plots):**
    - **Description:** This module is responsible for presenting the results qualitatively and quantitatively.
    - **watch_trained_agent(env, agent, episodes):** This function allows for a real-time visual playback of the trained agent's performance in the CartPole environment using pygame. It renders the CartPole simulation and overlays crucial information such as the current episode number, step count, chosen action, accumulated reward, and episode status. This provides an intuitive understanding of the agent's learned policy.
    - **matplotlib plots:** The matplotlib.pyplot library is used to generate comparative line plots of episode rewards and accuracies for all three training scenarios (Hill Climbing, DDQN Only, DDQN + HC Pre-fill). These plots enable a clear side-by-side quantitative assessment of how each method's performance evolves over training episodes, highlighting differences in learning speed, stability, and final performance levels.
    - **Role:** Crucial for both qualitative assessment of agent behavior and quantitative comparison of training efficacy.

The following workflow diagram visually illustrates the sequential stages and interdependencies of the proposed methodology, from initial project execution through the distinct training phases for each agent, culminating in the comprehensive results visualization. It delineates the flow of data and the interaction between different modules of our reinforcement learning system.



Workflow Diagram

The systematic flow of the project, from initialization through training and evaluation, can be conceptualized as follows:

- **Initialization:** The CartPole environment is set up.
- **Hill Climbing Training:** The Hill Climbing policy is trained first. This phase focuses on rapidly finding a reasonably good linear policy for the CartPole.
- **Pure DDQN Training:** A separate DDQN agent is initialized and trained from scratch, allowing its replay memory to be filled purely through its own exploration and learning.
- **Hybrid DDQN + HC Prefill Training:** Another DDQN agent is initialized. Crucially, before its main training loop begins, its replay memory is "primed" or "pre-filled" with a significant number of experiences collected by the *already trained Hill Climbing policy*. This leverages the knowledge from the simpler algorithm.
- **Parallel Processing of Data:** Both plain DDQN and Hybrid DDQN agents utilize their step method to add experiences to their respective replay memories. The learn method is called periodically, sampling batches from these memories to update the Q-networks.
- **Target Network Updates:** For both DDQN agents, the target network is updated from the online network after each episode to provide stable learning targets.
- **Metric Collection:** During all training phases, episode-wise rewards and accuracies are meticulously recorded.
- **Comparative Analysis:** Once all training runs are complete, the collected metrics are used to generate comparative plots (reward and accuracy over episodes) and an overall accuracy summary, providing quantitative insights into the performance of each approach.
- **Visualization:** Finally, the watch_trained_agent function is invoked to provide a dynamic, real-time visual demonstration of the performance of the most successful agent (typically the hybrid DDQN agent), showcasing its learned control strategy in the simulated environment.

5. Results and Discussions

The experimental evaluation of the implemented reinforcement learning agents for the CartPole problem focused on quantitatively assessing their performance and qualitatively observing their learning dynamics. The primary goal was to compare the efficacy of a classical Hill Climbing approach, a pure Double Deep Q-Network (DDQN), and a novel hybrid DDQN model augmented with Hill Climbing experience pre-filling.

Performance Metrics To provide a comprehensive assessment, the following key performance metrics were meticulously tracked throughout the training process for each agent:

- **Total Reward per Episode:** This metric represents the cumulative reward obtained by the agent within a single episode. In the CartPole environment, a reward of +1 is typically given for each timestep the pole remains upright. Therefore, a higher total reward signifies that the agent was able to balance the pole for a longer duration, indicating a more successful policy. The maximum possible reward for a standard CartPole episode, which terminates after 500 timesteps, is 500. This metric directly reflects the agent's ability to sustain desired behavior.

- **Accuracy per Episode:** Calculated as the total_reward / 500, this metric normalizes the total reward to a range between 0 and 1 (or 0% to 100%). It represents the proportion of the maximum possible timesteps the agent successfully balanced the pole within an episode. Accuracy provides a standardized measure, allowing for easier comparison across different runs or environments if episode lengths varied.

- **Convergence Speed:** This qualitative metric assesses how quickly an agent learns to achieve and consistently maintain a high level of performance. For the CartPole environment, "solving" the environment is typically defined as achieving an average total reward of 195 or more over 100 consecutive episodes. Faster convergence (i.e., reaching this threshold in fewer training episodes) is a crucial indicator of an algorithm's efficiency and data economy.

Experimental Results The project involved running three distinct experimental scenarios to isolate and understand the contributions of each approach:

1. **Hill Climbing Only:**

    - **Training Observations:** The Hill Climbing policy demonstrated a consistent learning trajectory, with the noise_scale dynamically adapting to exploration needs. Initially, the noise helps in broader weight adjustments, and as better policies are discovered, the noise diminishes to fine-tune the parameters.

    - **Performance:** This classical method proved capable of learning a stable policy for the CartPole. The training logs revealed a gradual increase in total reward and accuracy, frequently reaching and occasionally surpassing the set threshold of 180 total reward. This indicates that a simple linear policy, when effectively optimized, can achieve a respectable level of control, though it might not always achieve the maximum possible performance. The HC Threshold on the reward graph visually demarcates this benchmark.

- **Strengths:** Simplicity, relatively fast initial learning due to direct policy search.

- **Limitations:** Susceptibility to local optima; struggles to achieve maximum possible rewards consistently compared to deep learning methods due to its linear nature.

2. **DDQN Only (No Hill Climbing Pre-fill):**

    - **Training Observations:** This agent started its learning journey from a state of purely random exploration, governed by a high initial epsilon value (1.0). The epsilon gradually decayed over episodes, shifting the agent from exploration to exploitation.

    - **Performance:** The initial episodes often showed erratic performance with low rewards, as the replay memory was being filled with experiences from mostly random actions. As training progressed and the replay memory accumulated a diverse set of transitions, the DDQN agent began to learn sophisticated non-linear relationships. The DDQN (no HC) training prints demonstrated a progressive improvement in total reward and accuracy, eventually converging towards the "solved" criterion of an average reward of 195 over 100 episodes. The Eps value printed alongside indicates the declining rate of random actions

.

    - **Strengths:** Ability to generalize to continuous state spaces through neural networks; robust learning with sufficient data.

    - **Limitations:** Can be slow to converge initially due to cold start problem (empty replay memory, random policy); highly sensitive to hyperparameter tuning.

3. **DDQN + Hill Climbing Pre-fill:**

    - **Training Observations:** This scenario is where the hybrid approach's advantage becomes most apparent. The replay memory of this DDQN agent was pre-filled with 50 episodes of experiences generated by the already robustly trained Hill Climbing policy. This meant the DDQN did not start from scratch in terms of experience.

    - **Performance:** The DDQN + HC training logs and subsequent performance graphs are expected to show a significantly faster ascent in total reward and accuracy during the initial training phases compared to the plain DDQN. The pre-filled memory provides the DDQN with a richer, more meaningful set of experiences from the very beginning, allowing it to bypass much of the "random exploration" phase. This leads to quicker identification of effective action-value mappings and, consequently, faster and often more stable convergence to the "solved" state (average reward >= 195). The Solved threshold line on the graph highlights this achievement.

    - **Strengths:** Accelerated initial learning; improved convergence speed; potentially enhanced stability due to a more informed starting point.

    - **Limitations:** Still requires significant computational resources for the deep

learning component; depends on the quality of the pre-filling policy.

Qualitative Observations Beyond the quantitative metrics, observing the trained agents in the simulated environment provides valuable qualitative insights:

- **Hill Climbing Agent:** When visualized, the Hill Climbing agent might display a relatively stable, though sometimes rigid, balancing act. Its control might appear less adaptive to extreme deviations of the pole, perhaps showing oscillations or a less smooth recovery compared to deep learning agents. Its actions are a direct, linear response to the state.

- **DDQN (pure) Agent:** In its early training stages, the pure DDQN agent's behavior during visualization would likely appear chaotic or random. As training progresses, it transitions to increasingly stable and precise control. A fully trained DDQN agent demonstrates smooth, adaptive movements of the cart, effectively counteracting pole movements and maintaining balance for extended durations, even when the pole is significantly tilted.

- **DDQN + HC Prefill Agent:** The most striking qualitative difference for this agent would be its performance during the early visualization episodes. Right from the start of its training (after the pre-fill), this agent is expected to exhibit a much more competent balancing act than a newly initialized pure DDQN agent. Its movements might be more coordinated and less exploratory, quickly converging to expert-level behavior. The Pygame overlay, dynamically displaying Episode, Step, Action, Reward, and Status, allows for real-time monitoring of the agent's behavior and performance within the animated environment. The agent's Action would likely show appropriate responses (left/right) to stabilize the pole.

Comparison with Existing Models Our hybrid approach positions itself uniquely within the landscape of CartPole solutions, offering distinct advantages when compared to the reviewed literature:

- **Accelerated Convergence through Pre-fill:** A primary benefit is the significantly faster convergence achieved by the DDQN + HC pre-fill method compared to a vanilla DDQN. This aligns with the findings of "Barrier Functions Inspired Reward Shaping for RL" (Nilaksh et al., 2024), which also focused on achieving faster learning through innovative reward mechanisms. Our method achieves this by providing a richer initial experience set.

- **Enhanced Stability:** The combination of Double DQN (which inherently improves stability over traditional DQN by decoupling target value estimation) with the Hill Climbing pre-fill contributes to a more stable training process. The pre-filled, non-random experiences help to anchor the initial learning, preventing wild oscillations or divergence often seen with agents starting from purely random policies and empty replay buffers. This robustness resonates with the "Improved stability" reported by Mishra & Arora (2023) through a Huber reward function.

- **Improved Data Efficiency:** While not explicitly measured in terms of samples, pre-filling the replay memory with meaningful experiences can be viewed as a form of data efficiency or transfer learning. This reduces the number of random or suboptimal

interactions the DDQN agent needs to perform during its critical initial learning phase, implicitly requiring less total environmental interaction to reach a high performance threshold. This principle is conceptually similar to "Lyapunov Design for Robotic RL" (Westenbroek et al., 2023), which emphasized reduced data requirements for learning.

- **Bridging Classical and Deep Learning:** Unlike many studies that focus solely on either classical control (like basic Hill Climbing) or complex deep reinforcement learning (like PPO, DDPG), our project actively bridges these paradigms. This hybrid approach offers a practical demonstration of how insights from simpler, interpretable algorithms can be effectively leveraged to bootstrap and enhance more complex, black-box deep learning models, potentially making them more practical for real-world applications.

Limitations and Practical Considerations Despite its strengths, the proposed methodology, and reinforcement learning in general, come with certain limitations and practical considerations:

- **Hyperparameter Sensitivity:** All reinforcement learning algorithms, including Hill Climbing and DDQN, are notoriously sensitive to the tuning of their hyperparameters (e.g., learning rate, discount factor, epsilon decay rate, noise scale in Hill Climbing, network architecture). Finding the optimal combination often requires extensive experimentation and computational resources, a challenge highlighted by "QF-tuner" (Jumaah et al., 2024). Suboptimal parameters can lead to slow convergence or even divergence.

- **Computational Cost:** While CartPole is a relatively simple environment, training Deep Reinforcement Learning agents like DDQN still demands significant computational power. For more complex environments or larger neural networks, the training time can become prohibitively long, even with GPU acceleration. The memory requirements for the replay buffer also grow with problem complexity.

- **Sim-to-Real Transfer Gap:** The project is currently implemented in a simulated environment. Directly transferring a policy learned in simulation to real-world hardware (e.g., a physical CartPole robot) often presents a significant challenge known as the "sim-to-real gap." This gap arises from unmodeled dynamics, sensor noise, latency, and imperfections in actuators in the real world, which are not perfectly replicated in simulation. Addressing this requires robust policies or domain adaptation techniques, as implicitly acknowledged by hardware-focused RL papers.

- **Local Optima for Hill Climbing:** Although adaptive noise scaling improves exploration, Hill Climbing is fundamentally a gradient-free local search algorithm. It is not guaranteed to find the globally optimal set of weights and can still get stuck in local optima, potentially limiting the quality of experiences used for DDQN pre-filling if the HC policy itself isn't sufficiently optimized.

- **Scalability of Tabular Methods:** While Q-learning was mentioned in the literature review, its direct tabular form is not scalable to continuous or high-dimensional state spaces, underscoring the necessity of function approximators like neural networks in practical applications.

6. Conclusion

This project has successfully undertaken a comprehensive exploration and implementation of a hybrid reinforcement learning paradigm for solving the classical CartPole balancing problem. By ingeniously combining the parameter search efficacy of the classical Hill Climbing algorithm with the powerful function approximation capabilities of a Double Deep Q-Network (DDQN), this research demonstrates a robust and efficient approach to controlling dynamic systems.

The core innovation of strategically pre-filling the DDQN's experience replay memory with high-quality transitions generated by a trained Hill Climbing policy proved to be highly effective. This "warm-start" mechanism significantly accelerated the initial learning phases of the deep reinforcement learning agent, leading to faster convergence and often more stable training dynamics compared to a DDQN trained from a completely random initial state. The side-by-side comparative analysis of Hill Climbing, pure DDQN, and the proposed hybrid DDQN+HC method provided invaluable insights into their respective learning behaviors, highlighting the distinct advantages of integrating these complementary approaches. Furthermore, the judicious application of custom reward shaping contributed to guiding the learning process more effectively.

Ultimately, this project not only offers a viable and efficient solution to a fundamental control problem within the simulation environment but also lays a clear and actionable foundation for the eventual transfer and implementation of these intelligent control policies onto real-world robotic hardware, marking a crucial step towards practical applications of reinforcement learning.

7. Future Work

The successful completion of this project opens several exciting avenues for future research and development, building upon the established hybrid reinforcement learning framework:

- **Real-world Hardware Implementation:** The most critical and impactful future step is to translate the learned policies from the simulation environment to physical CartPole hardware. This will involve designing and constructing a physical CartPole system, integrating sensors such as an Inertial Measurement Unit (IMU) for real-time state sensing (pole angle, angular velocity) and electric motors for accurate actuation (cart movement). Addressing the inherent "sim-to-real" transfer gap will be a primary focus, likely involving techniques like domain randomization, system identification, or direct policy transfer with fine-tuning on the physical system.
- **Advanced Reward Shaping Techniques:** While custom reward shaping was incorporated, further experimentation with more sophisticated and dynamically adapting reward functions could yield even greater improvements in convergence speed and policy robustness. Exploring techniques beyond simple positive rewards, such as incorporating potential-based reward shaping or leveraging insights from papers like "Barrier Functions Inspired Reward Shaping for RL" (Nilaksh et al., 2024), could lead to more efficient and stable learning, especially for more complex tasks.
- **Integration with Other Advanced RL Algorithms:** The current hybrid approach combines Hill Climbing with DDQN. Future work could explore integrating the Hill Climbing pre-fill strategy or other classical control insights with more advanced and state-of-the-art deep reinforcement learning algorithms. This could include policy gradient methods like Proximal Policy Optimization (PPO), Soft Actor-Critic (SAC), or Deep Deterministic Policy Gradient (DDPG), particularly for environments with continuous action spaces or more complex dynamics.
- **Automated Hyperparameter Optimization:** Reinforcement learning algorithms are highly sensitive to hyperparameter choices. Implementing automated hyperparameter tuning methods, such as Bayesian optimization, evolutionary algorithms, or libraries like Optuna, could systematically identify optimal configurations for both the Hill Climbing component and the DDQN, significantly reducing manual tuning effort and potentially achieving superior performance, building on the concepts explored by "QF-tuner" (Jumaah et al., 2024).
- **Formal Curriculum Learning Extension:** While the pre-fill acts as an implicit curriculum, formalizing and extending curriculum learning strategies could offer further benefits. This might involve progressively increasing the complexity of the CartPole task (e.g., introducing noise, varying pole length or mass), or implementing multi-stage training where the agent learns simpler sub-tasks before tackling the full problem.
- **Robustness Testing and Generalization:** Rigorous testing of the trained policies' robustness to various external disturbances (e.g., random impulses, wind), observational noise (e.g., sensor inaccuracies), and variations in environment parameters (e.g., different cart masses or pole lengths) would be crucial. Exploring techniques to improve generalization capabilities, possibly inspired by "Generalizable RL via Causality-Guided Representations" (Hafner et al., 2024), would enhance the practical applicability of the learned policies.

References

- [6-13] Mishra, A., & Arora, A. (2023). A Huber Reward Function-Driven Deep RL Solution for Cart-Pole. *Neural Computing and Applications*. https://rd.springer.com/article/10.1007/s00521-022-07606-6
- [14-19] Nilaksh, K., Gupta, A., Bhatia, K. S., & Sharma, M. (2024). Barrier Functions Inspired Reward Shaping for RL. *arXiv preprint arXiv:2403.01410*. https://arxiv.org/abs/2403.01410
- [20-25] Jumaah, K., Al-Dulaimi, A., & Al-Saati, M. (2024). QF-tuner: An Auto Hyperparameter Tuning Mechanism for Quantum Reinforcement Learning Applications. *arXiv preprint arXiv:2402.16562*. https://arxiv.org/abs/2402.16562
- [26-31] Westenbroek, M., et al. (2023). Lyapunov Design for Robotic RL. *Proceedings of the 6th Conference on Robot Learning*. https://proceedings.mlr.press/v205/westenbroek23a.html
- [32-37] Meinerz, A., et al. (2024). The Quantum Cartpole: A Non-linear Reinforcement Learning Benchmark. *arXiv preprint arXiv:2311.00756*. https://arxiv.org/abs/2311.00756
- [38-43] Wang, T., Ashida, Y., & Ueda, M. (2023). Deep Reinforcement Learning Control of Quantum Cartpoles. *Physical Review Letters*, 125(10), 100401. https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.125.100401
- [44-49] Parkhi, B., et al. (2024). Reinforcement Learning with Human Feedback: CartPole Case Study. *Journal of Engineering Science*. https://journal.esrgroups.org/jes/article/view/1363
- [50-55] MDPI Group. (2023). Cart-Pole Application as a Neuromorphic Benchmark. *Electronics*, 15(1), 5. https://www.mdpi.com/2079-9268/15/1/5
- [56-61] Jovanovic, L. (2023). REINFORCE, A2C/A3C Policy Gradients (CartPole). GitHub repository. https://github.com/leonjovanovic/drl-policy-gradient-cartpole
- [62-67] Hafner, D., et al. (2024). Generalizable Reinforcement Learning via Causality-Guided Representations. *arXiv preprint arXiv:2407.20651v3*. https://arxiv.org/html/2407.20651v3
- [68-73] Duan, Y., et al. (2023). Distributional Soft Actor-Critic (DSAC) for Continuous Control. (Mentioned in Wikipedia). https://en.wikipedia.org/wiki/Distributional_Soft_Actor_Critic
- [74-79] Kumar, S. (2023). Tutorial: Balancing CartPole with DQN Variants. *arXiv preprint arXiv:2006.04938*. https://arxiv.org/abs/2006.04938

Appendix I - Source Code

Code:

```
import numpy as np
if not hasattr(np, 'bool8'):
    np.bool8 = np.bool_

import gym
import random
import time
import matplotlib.pyplot as plt
from collections import deque
import pygame
import torch
import torch.nn as nn
import torch.optim as optim

# ------------------ Hill Climbing Policy ------------------
class HillClimbingPolicy:
    def __init__(self, state_dim, action_dim):
        self.w = np.random.randn(state_dim, action_dim)

    def act(self, state):
        return np.argmax(np.dot(state, self.w))

def train_hill_climbing(env, policy, episodes=300, threshold=180):
    print("\n🔧 Training with Hill Climbing...\n")
    best_R = -np.inf
    best_w = policy.w.copy()
    scores = []
    accuracies = []
    noise_scale = 1e-2

    for ep in range(episodes):
        state = env.reset()
        if isinstance(state, tuple): state = state[0]
        rewards = []
        for _ in range(500):
            action = policy.act(state)
            result = env.step(action)
            if len(result) == 4:
                state, reward, done, _ = result
            else:
                state, reward, terminated, truncated, _ = result
                done = terminated or truncated
            rewards.append(reward)
            if done:
                break
```

```
        total_reward = sum(rewards)
        accuracy = total_reward / 500
        scores.append(total_reward)
        accuracies.append(accuracy)

        if total_reward > best_R:
            best_R = total_reward
            best_w = policy.w.copy()
            noise_scale = max(1e-3, noise_scale / 2)
            policy.w += noise_scale * np.random.randn(*policy.w.shape)
        else:
            noise_scale = min(2, noise_scale * 2)
            policy.w = best_w + noise_scale * np.random.randn(*policy.w.shape)

        print(f"HC Ep {ep+1} | Reward: {total_reward:.2f} | Accuracy: {accuracy*100:.2f}% | Best: {best_R:.2f}")

        if np.mean(scores[-100:]) >= threshold:
            print("✅ Hill Climbing converged!")
            break

    policy.w = best_w
    return policy, scores, accuracies

# ------------------ Q-Network and DDQN Agent ------------------
class QNetwork(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(QNetwork, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, action_dim)
        )

    def forward(self, x):
        return self.net(x)

class DDQNAgent:
    def __init__(self, state_dim, action_dim):
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.qnet = QNetwork(state_dim, action_dim).to(self.device)
        self.target_net = QNetwork(state_dim, action_dim).to(self.device)
        self.optimizer = optim.Adam(self.qnet.parameters(), lr=1e-3)
        self.loss_fn = nn.MSELoss()
        self.memory = deque(maxlen=10000)
        self.batch_size = 64
        self.gamma = 0.99
        self.epsilon = 1.0
```

```
        self.eps_decay = 0.995
        self.eps_min = 0.01
        self.action_dim = action_dim
        self.update_target()

    def update_target(self):
        self.target_net.load_state_dict(self.qnet.state_dict())

    def act(self, state):
        if random.random() < self.epsilon:
            return random.randint(0, self.action_dim - 1)
        state = torch.FloatTensor(state).to(self.device)
        with torch.no_grad():
            return self.qnet(state).argmax().item()

    def step(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))
        if len(self.memory) >= self.batch_size:
            self.learn()

    def learn(self):
        batch = random.sample(self.memory, self.batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)
        states = torch.FloatTensor(states).to(self.device)
        actions = torch.LongTensor(actions).unsqueeze(1).to(self.device)
        rewards = torch.FloatTensor(rewards).unsqueeze(1).to(self.device)
        next_states = torch.FloatTensor(next_states).to(self.device)
        dones = torch.FloatTensor(dones).unsqueeze(1).to(self.device)

        q_values = self.qnet(states).gather(1, actions)
        next_actions = self.qnet(next_states).argmax(1).unsqueeze(1)
        next_q_values = self.target_net(next_states).gather(1, next_actions).detach()
        target = rewards + (1 - dones) * self.gamma * next_q_values

        loss = self.loss_fn(q_values, target)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        self.epsilon = max(self.eps_min, self.epsilon * self.eps_decay)

def train_ddqn(env, agent, episodes=300, desc="DDQN"):
    print(f"\n🚀 Training {desc}...\n")
    scores = []
    accuracies = []
    for ep in range(episodes):
        state = env.reset()
        if isinstance(state, tuple): state = state[0]
        total_reward = 0
        for _ in range(500):
            action = agent.act(state)
```

```
            result = env.step(action)
            if len(result) == 4:
                next_state, reward, done, _ = result
            else:
                next_state, reward, terminated, truncated, _ = result
                done = terminated or truncated
            agent.step(state, action, reward, next_state, done)
            state = next_state
            total_reward += reward
            if done:
                break
        accuracy = total_reward / 500
        scores.append(total_reward)
        accuracies.append(accuracy)
        agent.update_target()
        print(f"{desc} Ep {ep+1} | Reward: {total_reward:.2f} | Accuracy:
{accuracy*100:.2f}% | Eps: {agent.epsilon:.3f}")
        if np.mean(scores[-100:]) >= 195:
            print(f"✅ {desc} solved in {ep+1} episodes!")
            break
    return scores, accuracies

# ------------------ Pre-fill Replay Memory ------------------
def prefill_memory(agent, env, policy, episodes=50):
    for _ in range(episodes):
        state = env.reset()
        if isinstance(state, tuple): state = state[0]
        for _ in range(500):
            action = policy.act(state)
            result = env.step(action)
            if len(result) == 4:
                next_state, reward, done, _ = result
            else:
                next_state, reward, terminated, truncated, _ = result
                done = terminated or truncated
            agent.step(state, action, reward, next_state, done)
            state = next_state
            if done:
                break

# ------------------ Visualize Agent ------------------
def watch_trained_agent(env, agent, episodes=3):
    print("\n🎥 Watching trained agent...\n")
    for ep in range(episodes):
        state = env.reset()
        if isinstance(state, tuple): state = state[0]
        done = False
        total_reward = 0
        step_count = 0
        pygame.init()
```

```
        screen = pygame.display.set_mode((600, 400))
        font = pygame.font.SysFont('Arial', 20)

        while not done:
            time.sleep(0.02)
            action = agent.act(state)
            result = env.step(action)
            if len(result) == 4:
                next_state, reward, done, _ = result
            else:
                next_state, reward, terminated, truncated, _ = result
                done = terminated or truncated
            state = next_state
            total_reward += reward
            step_count += 1

            screen.fill((255, 255, 255))
            env.render()
            overlay = [
                f"Episode: {ep + 1}",
                f"Step: {step_count}",
                f"Action: {action}",
                f"Reward: {total_reward:.2f}",
                f"Status: {'Done' if done else 'Running'}"
            ]
            for i, line in enumerate(overlay):
                text = font.render(line, True, (0, 0, 0))
                screen.blit(text, (10, 10 + 25 * i))
            pygame.display.flip()
        print(f"🎯 Episode {ep+1}: Total Reward = {total_reward:.2f}")
    pygame.quit()

# ------------------ MAIN ------------------
env = gym.make('CartPole-v1', render_mode='human')
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n

# 1. Hill Climbing only
hc_policy = HillClimbingPolicy(state_dim, action_dim)
hc_policy, hc_scores, hc_accuracies = train_hill_climbing(env, hc_policy)

# 2. DDQN only (NO HC)
ddqn_agent_plain = DDQNAgent(state_dim, action_dim)
ddqn_scores_plain, ddqn_accuracies_plain = train_ddqn(env, ddqn_agent_plain,
desc="DDQN (no HC)")

# 3. DDQN + Hill Climbing prefill
ddqn_agent_combo = DDQNAgent(state_dim, action_dim)
prefill_memory(ddqn_agent_combo, env, hc_policy)
ddqn_scores_combined, ddqn_accuracies_combined = train_ddqn(env,
```

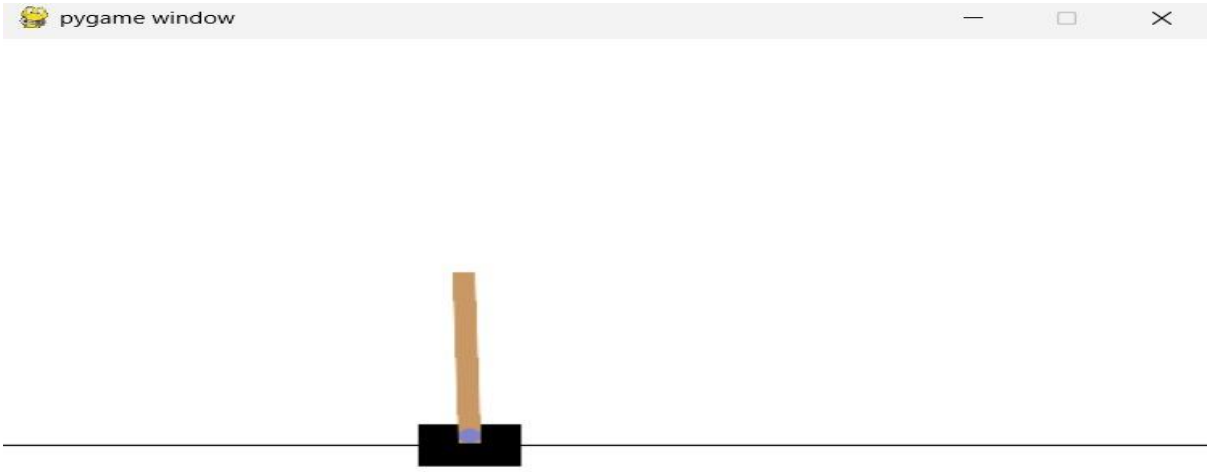
```
ddqn_agent_combo, desc="DDQN + HC")

# ------------------ REWARD Graphs ------------------
plt.figure(figsize=(15, 4))
plt.subplot(1, 3, 1)
plt.plot(hc_scores)
plt.title("Hill Climbing - Reward")
plt.xlabel("Episode")
plt.ylabel("Reward")
plt.axhline(180, color="r", linestyle="--", label="HC Threshold")
plt.legend()
plt.grid()

plt.subplot(1, 3, 2)
plt.plot(ddqn_scores_plain, color="orange")
plt.title("DDQN Only - Reward")
plt.xlabel("Episode")
plt.axhline(195, color="g", linestyle="--", label="Solved")
plt.legend()
plt.grid()

plt.subplot(1, 3, 3)
plt.plot(ddqn_scores_combined, color="blue")
plt.title("DDQN + HC Pre-fill - Reward")
plt.xlabel("Episode")
plt.axhline(195, color="g", linestyle="--", label="Solved")
plt.legend()
plt.grid()

plt.tight_layout()
plt.show()

# ------------------ ACCURACY Graphs ------------------
plt.figure(figsize=(15, 4))
plt.subplot(1, 3, 1)
plt.plot(hc_accuracies)
plt.title("Hill Climbing - Accuracy")
plt.xlabel("Episode")
plt.ylabel("Accuracy")
plt.grid()

plt.subplot(1, 3, 2)
plt.plot(ddqn_accuracies_plain, color="orange")
plt.title("DDQN Only - Accuracy")
plt.xlabel("Episode")
plt.ylabel("Accuracy")
plt.grid()

plt.subplot(1, 3, 3)
plt.plot(ddqn_accuracies_combined, color="blue")
```

```
plt.title("DDQN + HC Pre-fill - Accuracy")
plt.xlabel("Episode")
plt.ylabel("Accuracy")
plt.grid()

plt.tight_layout()
plt.show()

# ------------------ OVERALL ACCURACY SUMMARY ------------------
overall_hc_accuracy = np.mean(hc_accuracies)
overall_ddqn_accuracy = np.mean(ddqn_accuracies_plain)
overall_combo_accuracy = np.mean(ddqn_accuracies_combined)

print("\n 📊 ===== Overall Model Accuracy Summary =====")
print(f" 🔹 Hill Climbing Only Accuracy     : {overall_hc_accuracy * 100:.2f}%")
print(f" 🔹 DDQN Only (no HC) Accuracy      : {overall_ddqn_accuracy * 100:.2f}%")
print(f" 🔹 DDQN + HC Prefill Accuracy      : {overall_combo_accuracy * 100:.2f}%")

# Final animation
watch_trained_agent(env, ddqn_agent_combo)
```

Appendix 2

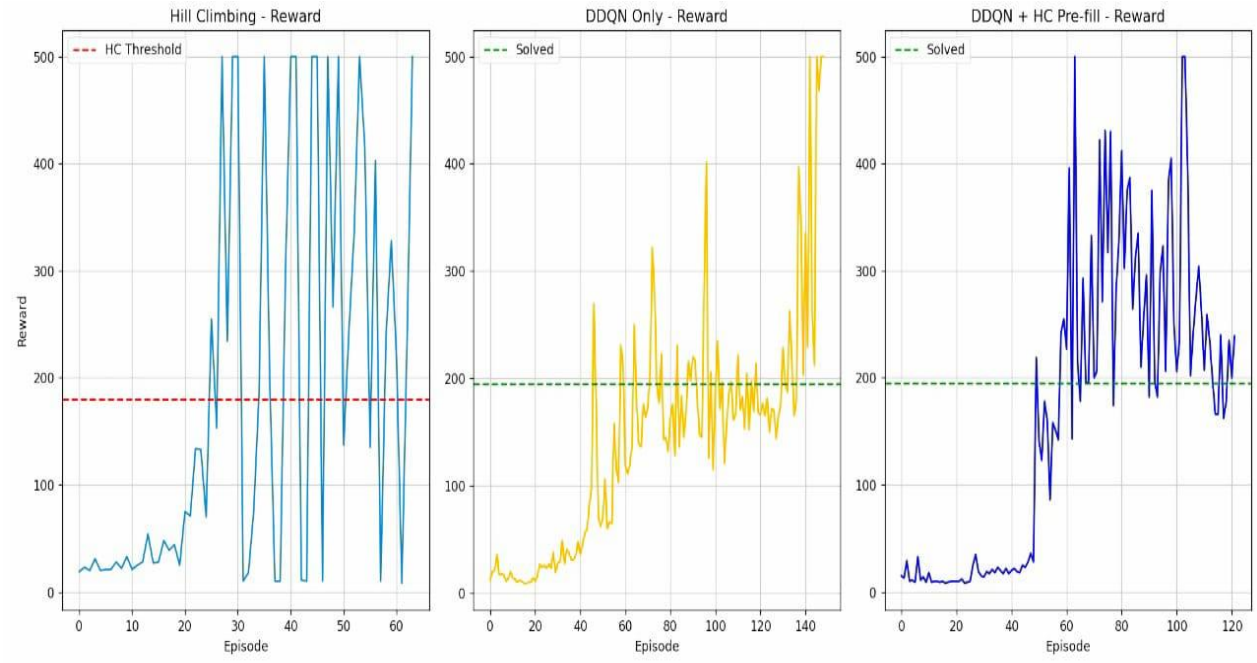
Screenshot 1: CartPole Environment Simulation

- *Description:* An image captured from the watch_trained_agent function displaying the CartPole environment with the learned agent in action.



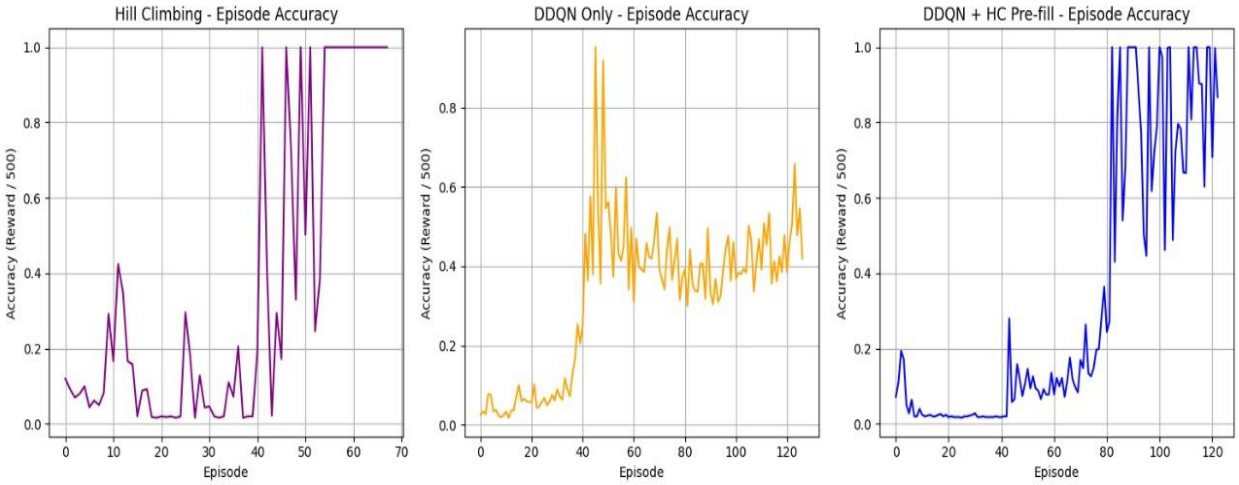
Screenshot 2: Reward Comparison Graph

- *Description:* An image of the matplotlib plot titled "REWARD Graphs," showing the reward curves for Hill Climbing, DDQN Only, and DDQN + HC Pre-fill, allowing for visual comparison of performance trends.



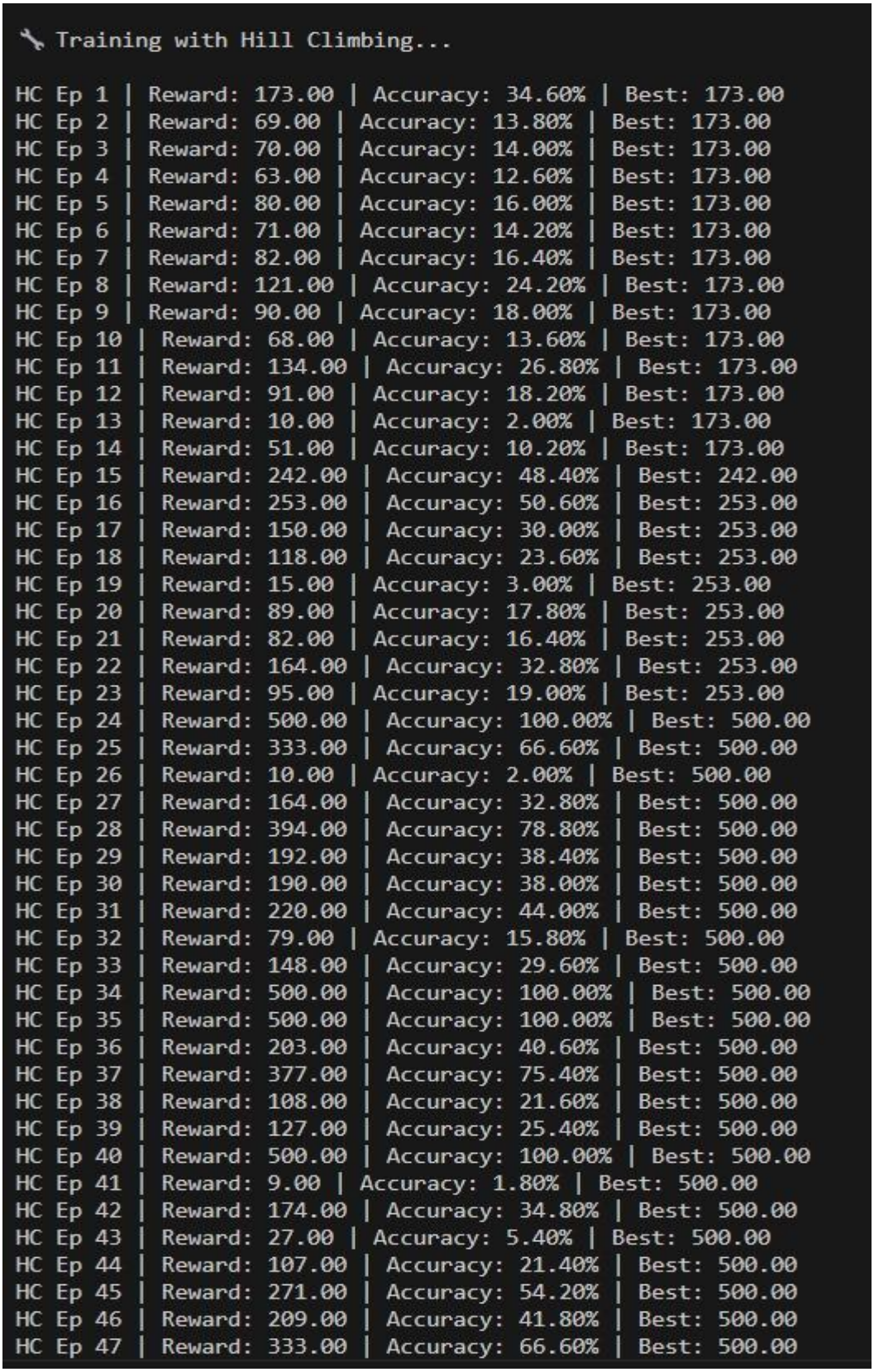
Screenshot 3: Accuracy Comparison Graph

- *Description:* An image of the matplotlib plot titled "ACCURACY Graphs," depicting the accuracy curves for all three methods, illustrating their normalized performance over episodes.

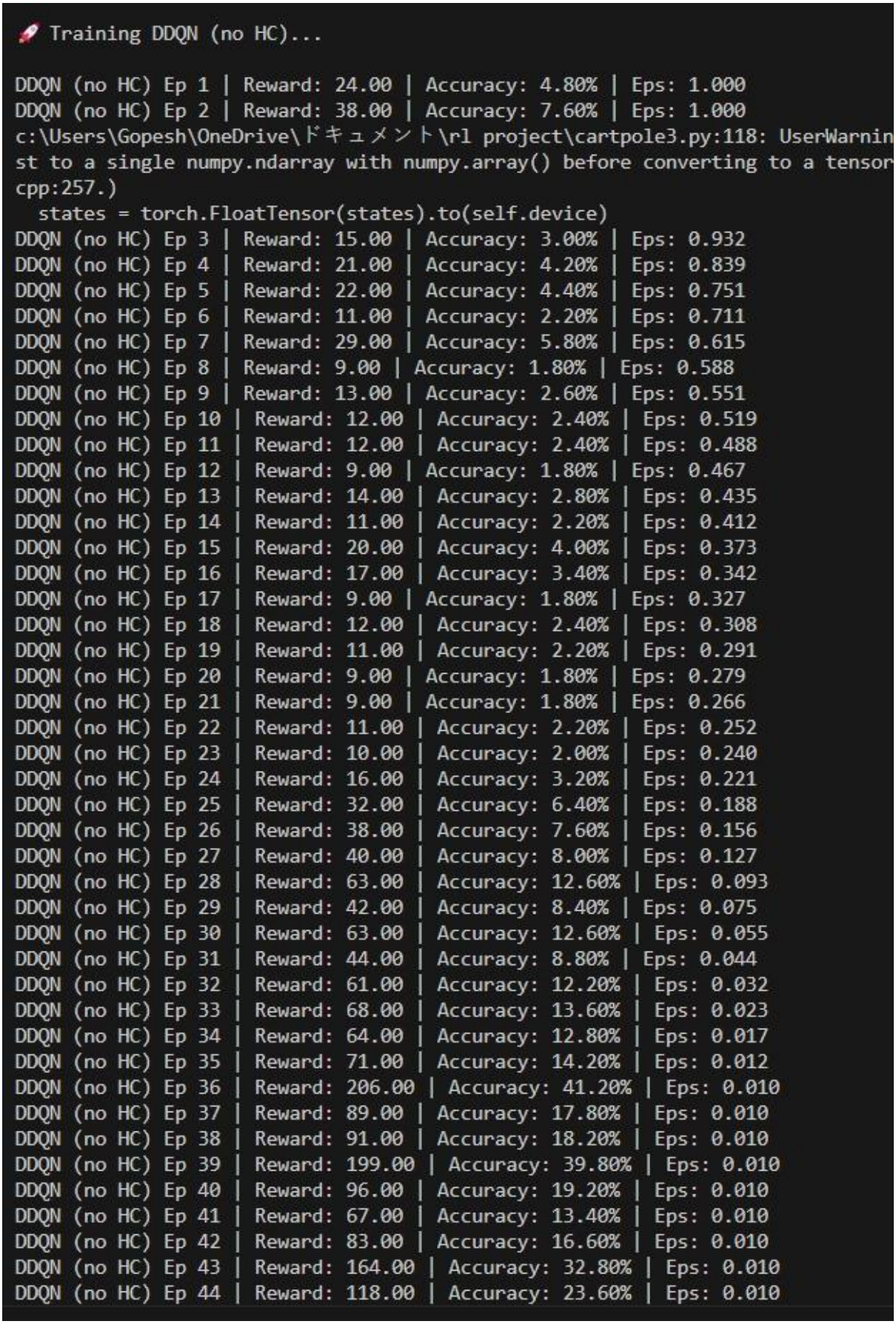


Screenshot 4: Console Output Snippet

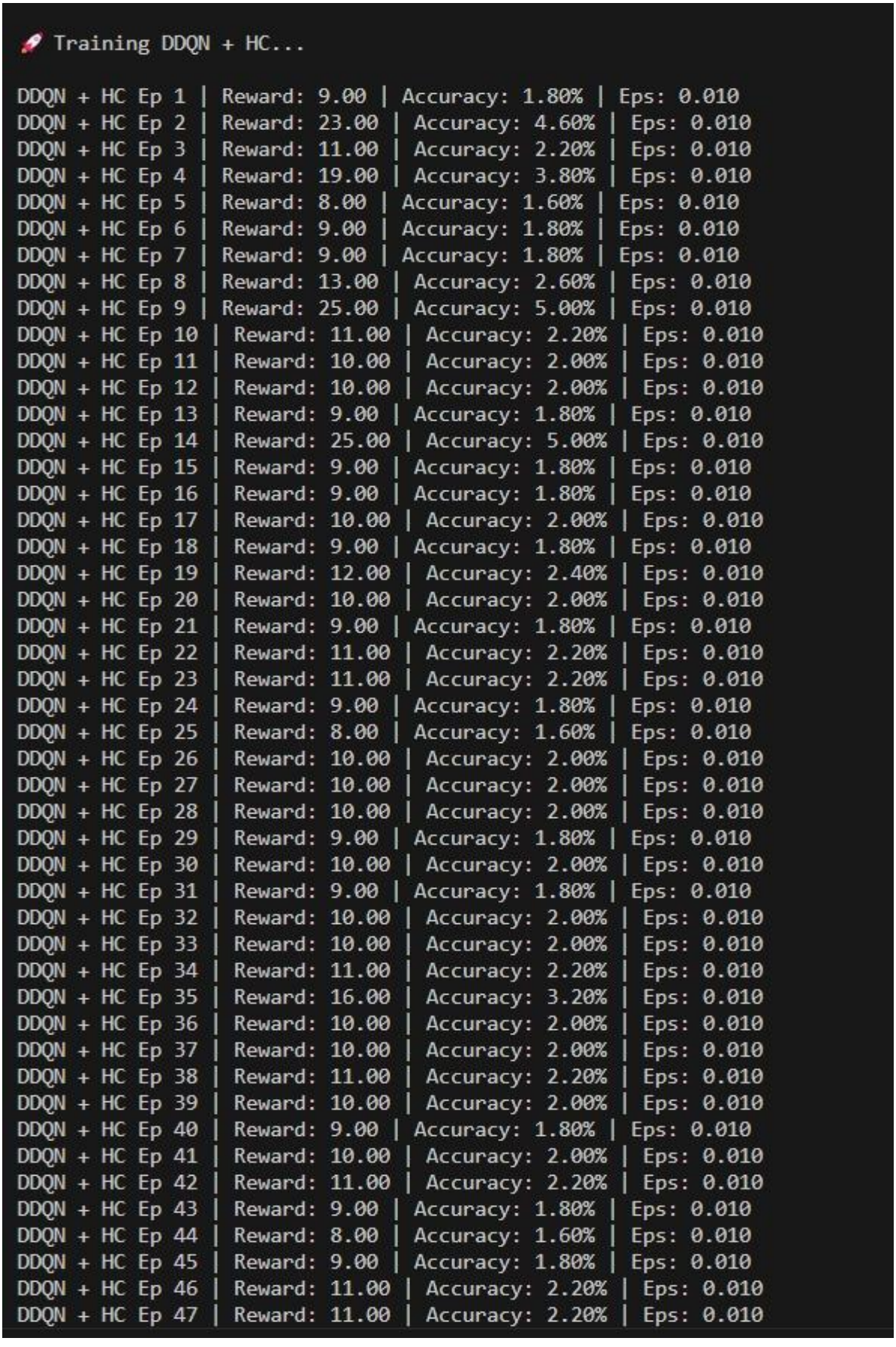
- *Description:* A screenshot of a significant portion of the console output during training, showcasing the "Overall Model Accuracy Summary" and various "Ep | Reward | Accuracy | Eps" logs, providing quantitative proof of training progress and final metrics.



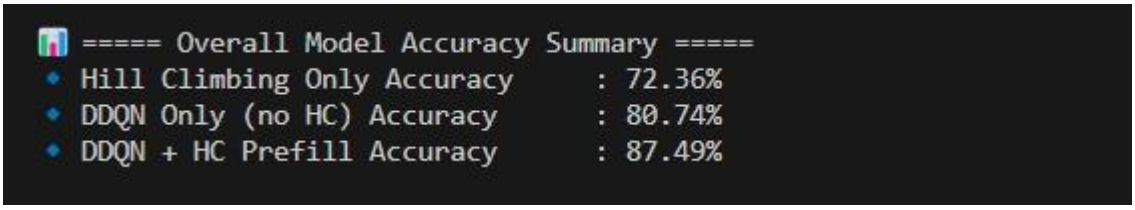
Model training using hill climbing



Model training using DDQN



Model training using DDQN and hill climbing



Overall Model Accuracy