Assignment 3: Actor-Critic Algorithm

1 Introduction

This Assignment is based on implementation of Actor Critic algorithm on different environments. We have used Advantage Actor critic (A2C) method with multithreading. The main idea of actor-critic algorithm is that it combines the benefits of both policy based and value based methods by using two neural network heads.Our implementation was tested across both simple and complex environments to evaluate its generalization capabilities and performance under varying state and action spaces.

1.1 Roles of Actor and Critic

The Actor is responsible for learning the policy, which defines the probability distribution over actions given a particular state. The actor's role is to improve the policy by maximizing the expected cumulative reward. It does this using policy gradient which pushes the policy towards actions that gives us higher advantage values. Where as the Critic estimates the value function, which represents the expected return that is the cumulative future reward from state, following the current policy. The critic helps the actor by providing a good target to compare. The critics role is to evaluate how good the action is that was taken by the actor and reduce variance in the policy updates done by actor. By estimating how good a state is, the critic provides a more stable signal to guide the actor's learning flow.

In shot, the actor is responsible for selecting actions given a state, while the critic evaluates the chosen action by estimating the value function

1.2 The Advantage Function

As discussed above, the actor takes an action, the critic evaluates the quality of actions. The advantage function is the key thing that helps the agent to understand how much better or worse a particular action is compared to the average performance at a given state. This provides a direction for the actor to improve the policy. If the advantage function is greater than zero, that means the action which was taken led to a higher reward than expected. This is indication for the actor to increase the probability of that action. On the other hand, if the advantage value was less than zero then the action taken by the actor was worse than expected, so the actor decreases its probability of that action. By this we can clearly see that advantage function is useful in guiding the actor to improve its policy. The other use of advantage function is that it connects the value estimates from the critic with the action selection policy of the actor. By doing this, it ensures that the actor doesnot blindly follow actions that gives high rewards but rather it helps actor choose actions that gives better than expected rewards compared to the current state value. The formula for advantage function is:

$$A(s_t, a_t) = r_t + \gamma V(s_{t+1}) - V(s_t)$$

1.3 Loss Function

The loss function in actor critic method is combination of actor loss, critic loss and entropy bonus. The critic is trained by decreasing the critic loss. As we know that the critic learns to estimate the value of a state that is, the expected cumulative reward from that state. This critic network is trained using temporal difference error. The loss function for the critic is mean squared error between the estimated value and the target. The formula is:

$$\mathcal{L}_{\text{critic}} = \left(r_t + \gamma V(s_{t+1}) - V(s_t)\right)^2$$

As discussed above, the actor uses the policy gradient method. The A2C uses the advantage function to guide updates instead of the total return. The loss increases the probability of actions that had

positive advantage and decreases it for negative advantages. The formula for loss function of actor is:

$$\mathcal{L}_{\text{actor}} = -\log \pi(a_t|s_t) \cdot A(s_t, a_t)$$

The entropy bonus is used to prevent the policy from becoming too deterministic in early stages itself, hence encouraging the network to do more exploration. This helps the network to avoid the premature convergence. The formula is:

$$\mathcal{L}_{\text{entropy}} = -\beta \sum_a \pi(a|s_t) \log \pi(a|s_t)$$

Total loss is calculated by the below equation:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{actor}} + c_v \cdot \mathcal{L}_{\text{critic}} + \mathcal{L}_{\text{entropy}}$$

where:

- The A(s,a) is the advantage function
- $\pi(a_t|s_t)$: Probability of taking action a_t in state s_t
- $V(s_t)$: Value estimate of state s_t
- r_t: Immediate reward at time t
- γ: Discount factor
- β: Entropy coefficient
- c_v: Coefficient for value loss

2 Environments Description

2.1 Cartpole

For the part 2 implementation, we used the CartPole-v1 environment from the Gymnasium library. This classic control task involves a pole attached to a cart that moves along a one dimensional track. The goal of the agent is to apply forces to the cart in order to keep the pole balanced upright.

- **State Space:** A 4-dimensional continuous vector representing:
  - Cart position
  - Cart velocity
  - Pole angle
  - Pole angular velocity
- **Action Space:** A discrete action space with 2 possible actions:
  - 0: Move the pole to the left
  - 1: Move the pole to the right
- **Agent:** The agent controls the horizontal movement of the cart and learns to balance the pole using the policy and updates this policy from experience.
- **Reward Function:** The agent receives a reward of +1 for every timestep that the pole remains upright. There is no penalty unless the pole falls and the episode ends.
- **Termination Criteria:**
  - The pole angle exceeds $\pm 12°$
  - The cart position exceeds ± 2.4 units from the center
  - The episode reaches 500 timesteps. This is considered as success.
- **Goal:** The goal is to maximize the duration for which the pole remains without falling by learning an optimal policy that selects actions accordingly based on the current state.

This environment is a widely used benchmark for testing reinforcement learning algorithms due to its simplicity and immediate feedback. It is especially suitable for A2C.

2.2 Lunar Lander

The Lunar Lander environment simulates a 2D rocket lander that must land upright on a designated pad using discrete engine commands. This environment presents a moderately complex task that requires stable control under gravity.

- **State Space:** An 8-dimensional continuous vector:
  - Horizontal and vertical position (x, y)
  - Linear velocities (v_x, v_y)
  - Angle and angular velocity
  - Boolean contact flags for left and right legs
- **Action Space:** Discrete with four actions:
  - 0: Do nothing
  - 1: Use left engine
  - 2: Use main engine
  - 3: Use right engine
- **Agent:** The agent controls the lander's thrusters to reduce speed and align for a safe landing.
- **Reward Function:**
  - $+100$ for successful landing
  - -0.3 per timestep (fuel usage penalty)
  - -100 for crashing
  - $+10$ for each leg making contact
- **Goal:** Safely land on the landing pad while minimizing fuel consumption and avoiding crashes.
- **Termination:** Episode ends upon crashing or successful landing.

2.3 Car Racing

The Car Racing environment presents a continuous control task where the agent must drive a car around a procedurally generated racetrack using pixel-based input.

- **State Space:** A $96 \times 96 \times 3$ RGB image representing the top down view of the track and car.
- **Action Space:** Continuous vector with three dimensions:
  - Steering angle: $[-1, 1]$
  - Acceleration : $[0, 1]$
  - Brake: $[0, 1]$
- **Agent:** The agent controls the car's steering, throttle, and brake using a learned policy network.
- **Reward Function:**
  - $+1$ per track tile visited
  - -0.1 per frame for slow or ineffective actions
- **Goal:** Maximize the number of unique tiles visited by successfully navigating the entire track.
- **Termination:** The episode terminates if the agent fails to move forward or the track is completed.

3 A2C Implementation and Thread Synchronization

We implemented a synchronous Advantage Actor-Critic (A2C) algorithm using Python's threading module to enable parallel training through multiple actor-learner processes. This design allows for efficient experience collection and more stable gradient updates.

- **Global Shared Model:** A central Actor-Critic neural network is maintained in shared memory. It has a common feature extraction base with two output heads: one for policy logits that is actor and one for value estimation which is critic.
- **Thread Architecture:** Two actor worker thread processes are used, each with its own independent copy of the environment. These processes interact with their environments in parallel, collecting the experience that is states, actions, and rewards.
- **Interaction with Global Model:**
  - Each thread uses the current global model to compute actions and collect experiences.
  - After a fixed number of environment steps n_steps, each thread computes local gradients based on its collected data.
  - These gradients are then sent to the main process, which averages them across all threads and applies the update to the shared global model.
- **Synchronization Strategy:**
  - All threads are paused after completing their n_steps.
  - Gradients are aggregated and applied synchronously, ensuring consistent updates to the model.
  - Once the update is complete, the threads are resumed for the next cycle of interaction and gradient computation.
- **Gradient Clipping:** To stabilize learning and avoid exploding gradients, we have used gradient clipping to the model's parameters before the optimizer step using the following code:

This synchronous update mechanism ensures that all the worker threads contribute equally to the training process, enabling more stable and coordinated learning. Additionally, since all gradients are computed based on different episodes, this approach is good as it uses different experiences from each actor threads.

3.1 Architecture

We designed custom Actor-Critic neural network architectures specific to the input and action space complexities of the CartPole, LunarLander, and CarRacing environments. Each model consists of a shared encoder followed by two output heads: one for the actor policy and one for the critic value estimation.

The CartPole environment has a low-dimensional 4D continuous state space and a discrete action space with two actions. We used a simple feedforward neural network architecture with two fully connected layers and ReLU activations:

- **Shared layers:** Two fully connected layers with ReLU.
- **Actor Head:** A linear layer outputting logits over two actions that is left or right.
- **Critic Head:** A linear layer outputting a scalar value $V(s)$.

This lightweight architecture is effective for simple control problems, enabling fast convergence with minimal overfitting risk.

The Lunar Lander environment has an 8-dimensional state space and a discrete action space with four possible actions. We used more deep network structure with shared feature extraction and separate heads for the policy and value functions:

- **Shared Layers:** Two fully connected layers with ReLU activation.

- **Actor Head:** A two-layer network projecting to action logits.
- **Critic Head:** A two-layer network outputting a scalar value.

This structure allows the network to specialize the output branches for policy and value learning while retaining shared low-level features.

The Car Racing environment is a complex continuous environment with high-dimensional visual input ($3 \times 96 \times 96$ RGB images). We used a convolutional architecture to extract features:

- **CNN Block:** Three convolutional layers with ReLU activation for visual feature extraction.
- **Fully Connected Layer:** A dense layer processes flattened CNN features.
- **Actor Head:**
  - `muHead`: Outputs the mean of the continuous action distribution.
  - `logStdHead`: A trainable parameter representing log standard deviation.
- **Critic Head:** Outputs a scalar value estimate $V(s)$ from the shared features.

The actor outputs a Gaussian distribution used to sample continuous control actions, this is the common way to get actions in continuous action reinforcement learning.

4 Results

4.1 Cartpole

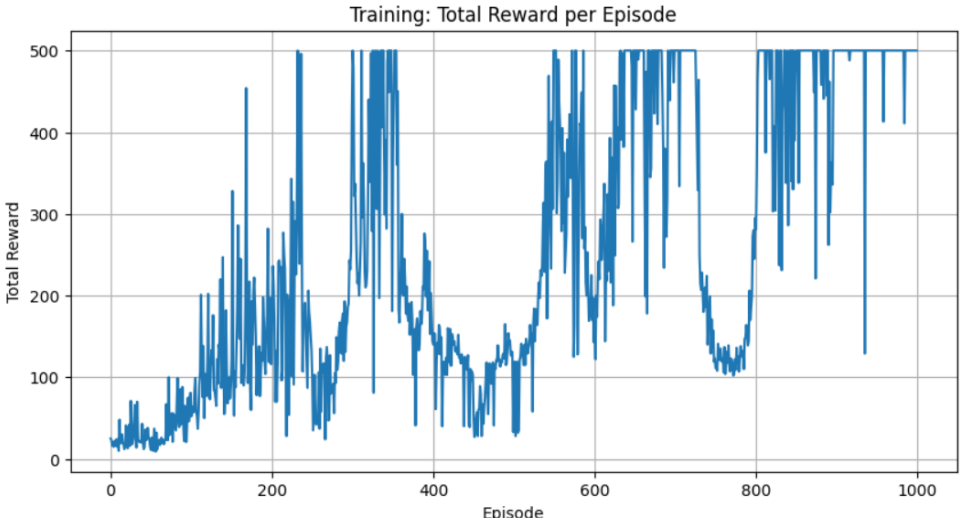


Figure 1: Training: Total Reward per Episode on CartPole

The plot shows a clear upward trend in episode rewards over time, indicating that the agent successfully learned to balance the pole for longer durations. Initially, the reward was low due to random exploration. As training progressed, the policy improved which lead to eventually converging to good outcome where the pole remains balanced for the full episode, that is when we get 500 reward.

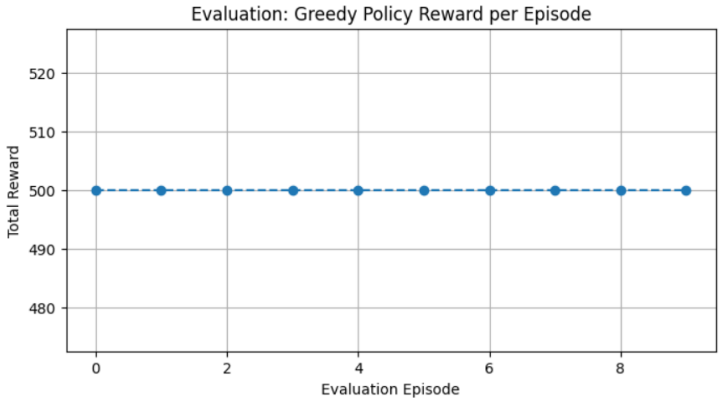


Figure 2: Evaluation: Greedy Policy Reward on CartPole-v1

To test the quality of the learned policy, we ran the agent for ten episodes using only greedy action selection strategy. As shown above, the agent consistently achieved the maximum reward of 500 in all 10 evaluation episodes. This confirms that the policy has generalized well.

Result Interpretation:
The results clearly shows the effectiveness of the A2C algorithm in learning an optimal policy in the CartPole environment. Despite fluctuations during training, the agent steadily improved through gradient updates from multi-threaded experience collection. The final evaluation performance shows the stability and reliability of the learned policy.

4.2 Lunar Lander

Figure 3: Training: Total Reward per Episode on LunarLander

As shown above, the training curve initially exhibits highly negative rewards due to the agent's exploratory and unstable behavior. Over the course of 1000 episodes, the agent progressively improves its landing capability. The average reward gradually shifts toward positive side, with several high-reward peaks appearing between episodes 600 and 900.

However, the learning remains somewhat noisy, which is expected in this environment due to sparse rewards, delayed feedback, and the presence of complex contact dynamics like landing legs touching the ground, angular velocity. The plot suggests that the agent learned to perform soft landings in many episodes.

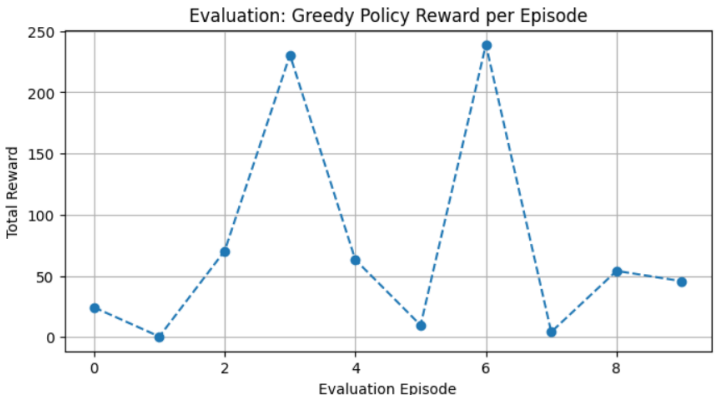


Figure 4: Evaluation: Greedy Policy Reward per Episode on LunarLander

The agent was evaluated over 10 episodes using a greedy policy which was learned from network. The evaluation rewards vary significantly between episodes, ranging from near-zero to over 240.

This variability is indicative of a partially trained policy that succeeds in landing. While it does not achieve consistent optimal performance, the learned policy is capable of executing successful landings.

Result Interpretation:
The A2C algorithm, when applied to Lunar Lander, demonstrates the capacity to learn stable behavior in a challenging environment. The agent improved significantly over time, with rewards rising from below -200 to above +200 in some episodes. Evaluation reveals a stochastic policy that occasionally results in successful landings. Some variance in both training and evaluation indicates the need for further tuning of entropy regularization or improved reward formulation.

4.3 Car Racing

To evaluate our A2C implementation in a complex and continuous action space, we applied the algorithm to the CarRacing environment. This environment presents significant challenges due to high-dimensional image inputs and the need for precise control.

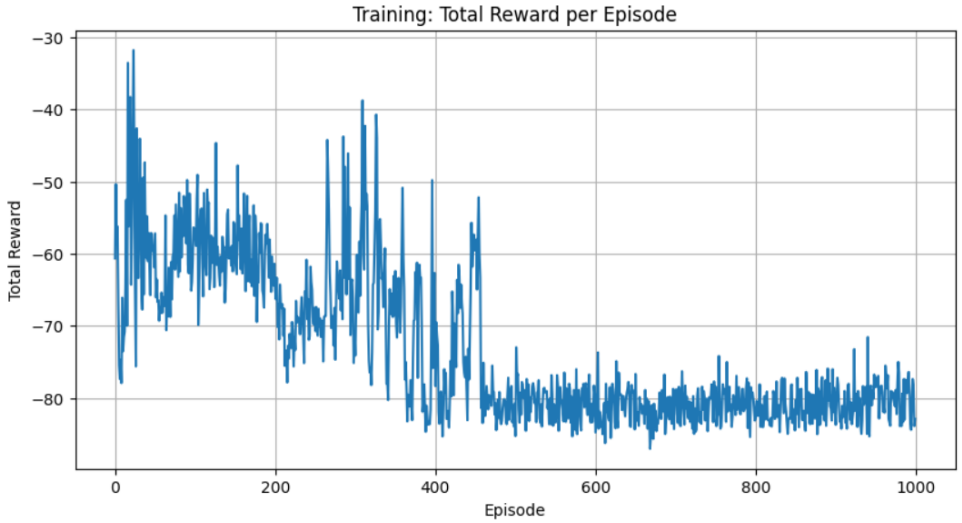


Figure 5: Training: Total Reward per Episode on CarRacing

The training curve shows little unstable behavior in the early phase, but eventually stabilizes after 600 episodes. Initially, the agent explores the environment with suboptimal driving behaviors so that resulted in low rewards like below -70. Over time, the agent gradually improves in navigating the track, that was seen in later episodes which gave high mean reward. The reward distribution suggests that the current policy sometimes struggles to make consistent forward progress or efficiently complete laps.

This performance is expected due to the difficulty of learning from high-dimensional visual input and the complexity of continuous control. more training epochs, improved reward shaping could help overcome this and learn in a more efficient way.

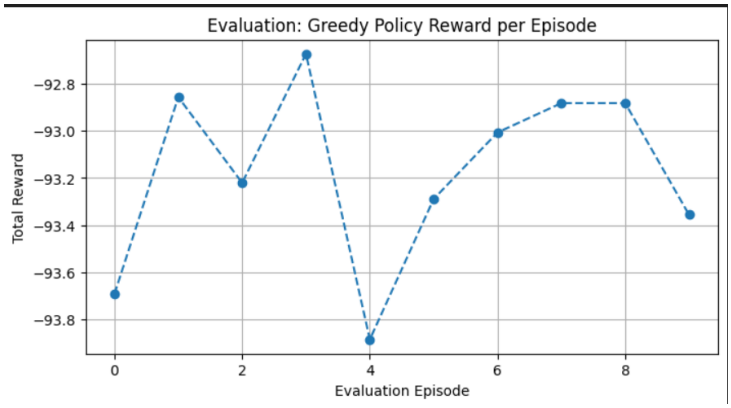


Figure 6: Evaluation: Greedy Policy Reward per Episode on CarRacing

Conclusion

In this assignment, we implemented the Advantage Actor-Critic (A2C) algorithm from scratch and applied to three different environments varying in complexity, including CartPole, LunarLander, and CarRacing. Through this process, we gained a good understanding of the actor-critic architecture, policy gradient optimization, learned how advantage is uised to make the actors learn and entropy based exploration control.

We implemented multi-threaded training using PyTorch's threading module, where each actor thread collected experiences in its own environment and contributed gradients to a shared global model. Synchronization mechanisms ensured a correct and good coordination and gradient clipping was used for stability.

5 References

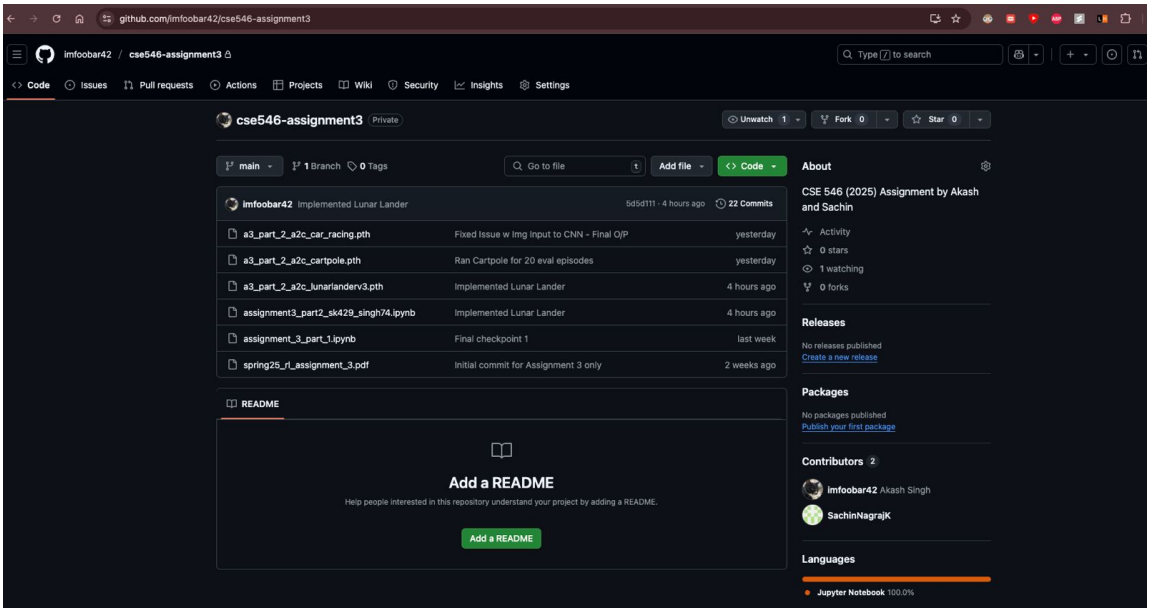
1. NeurIPS Styles (docx, tex).
2. Overleaf (LaTex based online document generator)- a free tool for creating professional reports
3. Lecture slides
4. Atari Environments
5. Gymnasium Environments
6. Sutton, R.S. & Barto, A.G. (2018). *Reinforcement Learning: An Introduction*. MIT Press.
7. Gymnasium Documentation. `https://gymnasium.farama.org/`

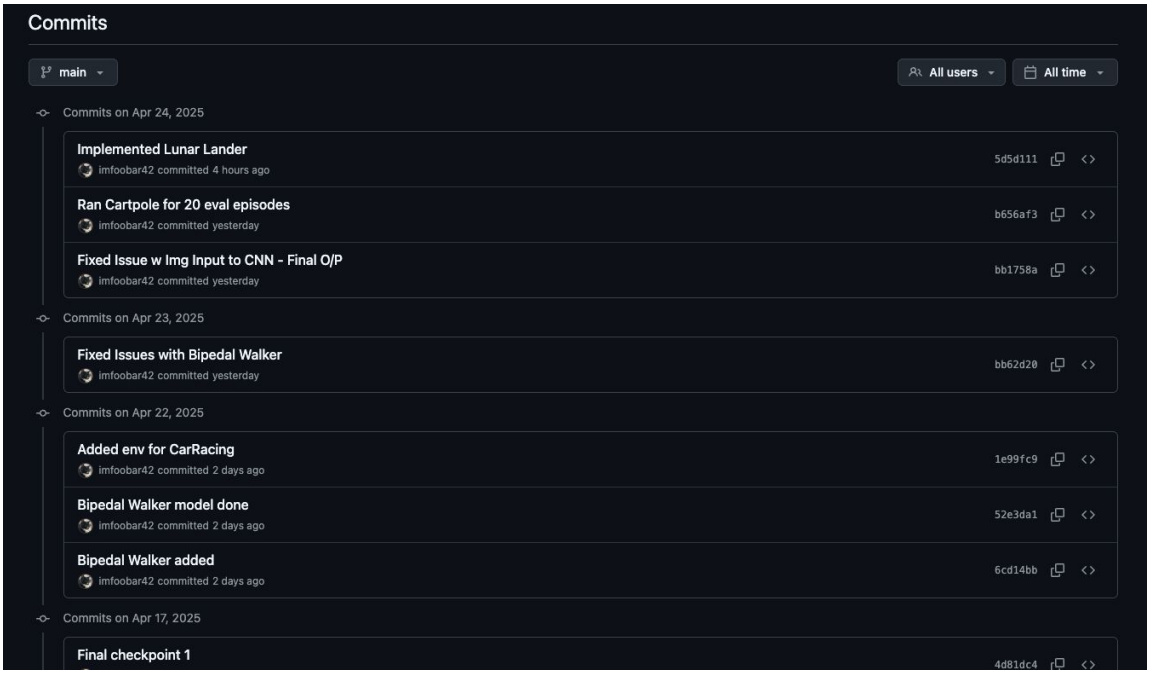
6 Contribution

- Akash Singh (singh74): 50 percent
- Sachin Kulkarni (sk429) : 50 percent

Github Repository link:

https://github.com/imfoobar42/cse546-assignment3


github.com/imfoobar42/cse546-assignment3
imfoobar42 / cse546-assignment3
Code Issues Pull requests Actions Projects Wiki Security Insights Settings
cse546-assignment3 Private
Unwatch 1 Fork 0 Star 0
main 1 Branch 0 Tags Go to file Add file Code
imfoobar42 Implemented Lunar Lander 5d5d111 · 4 hours ago 22 Commits
a3_part_2_a2c_car_racing.pth Fixed Issue w Img Input to CNN - Final O/P yesterday
a3_part_2_a2c_cartpole.pth Ran Cartpole for 20 eval episodes yesterday
a3_part_2_a2c_lunarlanderv3.pth Implemented Lunar Lander 4 hours ago
assignment3_part2_sk429_singh74.ipynb Implemented Lunar Lander 4 hours ago
assignment_3_part_1.ipynb Final checkpoint 1 last week
spring25_rl_assignment_3.pdf Initial commit for Assignment 3 only 2 weeks ago
README
Add a README
Help people interested in this repository understand your project by adding a README.
Add a README
About
CSE 546 (2025) Assignment by Akash and Sachin
Activity
0 stars
1 watching
0 forks
Releases
No releases published
Create a new release
Packages
No packages published
Publish your first package
Contributors 2
imfoobar42 Akash Singh
SachinNagrajK
Languages
Jupyter Notebook 100.0%


Commits
main
All users
All time
Commits on Apr 24, 2025
Implemented Lunar Lander
imfoobar42 committed 4 hours ago
5d5d111
Ran Cartpole for 20 eval episodes
imfoobar42 committed yesterday
b656af3
Fixed Issue w Img Input to CNN - Final O/P
imfoobar42 committed yesterday
bb1758a
Commits on Apr 23, 2025
Fixed Issues with Bipedal Walker
imfoobar42 committed yesterday
bb62d20
Commits on Apr 22, 2025
Added env for CarRacing
imfoobar42 committed 2 days ago
1e99fc9
Bipedal Walker model done
imfoobar42 committed 2 days ago
52e3da1
Bipedal Walker added
imfoobar42 committed 2 days ago
6cd14bb
Commits on Apr 17, 2025
Final checkpoint 1
4d81dc4