# Section 1: Cartpole Environment

The Cartpole environment here is modeled as a Markov Decision Process because it satisfies the Markov property: The next state and reward depend only on the current state and the chosen action, and not on the history of previous states and actions.

The Cartpole environment involves an agent that observes the four-dimensional state vector and selects a left or right push action based on the policy to keep the pole balanced.

State Space (S):

* The state space is four-dimensional.

It represents the following properties of the cartpole environment:

* Cart Position (x): This indicates the horizontal position of the cart.
* Cart Velocity (x\_dot): This represents the speed of the cart along the horizontal direction.
* Pole Angle (theta): This is the angle of the pole with respect to the vertical axis.
* Pole Angular velocity/Rate of Change of the pole angle (theta\_dot): This represents the rotational speed of the pole.

Action Space (A):

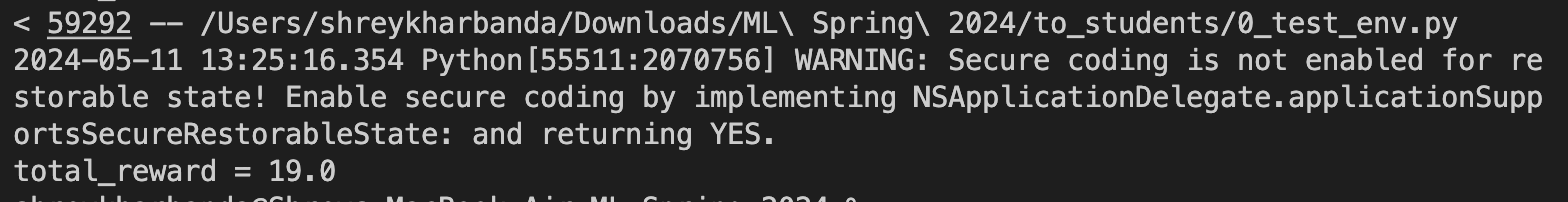
The action space is discrete and has two possible actions.

* An action value of 0: applying left force on the cart.
* An action value of 1: applying right force on the cart.

Reward:

* A positive reward of 1: for each timestep the agent takes an action that keeps the pole balanced.

On running script 0, I get a total reward of 19.0 for now.



# Section 2: Policy deep neural network

In actor\_v0.py,

* The state space is 4-dimensional. Hence, I put 4 for input\_size
* The action-space contains two discrete actions, left or right force and that’s why I put 2 for nb\_actions

In script 1,

For a strategy on how to select the best action, I considered pros and cons of 3 options:

1. Random Sampling:

* Pros: Simple to implement, encourages initial exploration
* Cons: Might be too explorative, leads to slow learning and potentially unstable behavior

1. Epsilon-Greedy:

* Pros: Provides a good balance between exploration and exploitation. We learnt about in class and I’m really familiar with its mathematical reasoning.
* Cons: Requires tuning the epsilon parameter, and it can affect performance if not chosen carefully.

1. Boltzmann Selection:

* Pros: Offers a smoother transition between exploration and exploitation compared to epsilon-greedy. It involves a temperature parameter that can control the exploration vs. exploitation trade-off.
* Cons: Requires tuning the temperature parameter, the selection process might be slightly more computationally expensive than other methods.

Overall, I ended up choosing Epsilon-Greedy because:

* I think for a logically simple and effective problem here, I would like to choose epsilon-greedy. Since the cartpole problem strategy is more deterministic than exploratory, I choose 0.1 for now. This means more exploitation than exploration.
* Hence, I stuck to manual testing for values: 0.1, 0.5, 0.6, 0.7, 0.75, 0.8, 0.85, 0.9, 1.0
* The best reward I got was for an epsilon value of 0.1 (it was high, compared to the rest)
* For an epsilon value of 0.1, I got a reward of 17.0.

Policy performance on cartpole,

* The performance is pretty bad compared to what we aim to achieve in this project, and that is because we haven’t even implemented a reinforcement learning algorithm yet, which we do in Section 3 below.

# Section 3: REINFORCE algorithm

I made some mistakes in this section initially:

* I forgot to add a stopping criteria that stops this algorithm early in case it’s not training well or converging appropriately.
* I was calculating the discounted rewards in a different order of elements
* My action\_path variable kept on giving a file not found error. So, I ended up using the full path instead of a relative path.
* I was using a negative log-likelihood without reversing the discounted rewards. This gave me a non converging algorithm, but as soon as I fixed this, my training started to converge very quickly.
* I added an ineffective stopping criteria. I decided that my training should stop when either episode\_total\_reward >= MAX\_REWARD. Here, the MAX\_REWARD = 500. According to the question, the training should give out a cumulative reward of atleast 500, so I chose 500 itself. However, my stopping criteria was based on a sparse reward signal which was leading to premature stopping, even if the policy was not getting well-trained. After emailing the professor and understanding his suggestion, I realized that a better stopping criterion would consider both the cumulative reward and the performance over multiple episodes to ensure the policy is consistently achieving good rewards.

I recognized my mistakes fast and solved them.

I tried different learning rates and discount factors.

Learning Rate: I recognized that a high learning rate was ruining future rewards in my training and my model was taking very long to converge. So, I reduced it from 0.1 to 0.01 to 0.001. It worked really well for 0.001 and I ended up choosing that. For LR = 0.001, my model was converging in less than 900 iterations

Discount Factor: I tested different discount factors from 0.5 to 0.99 and recognized that by choosing a higher discount factor I’m giving the future rewards less weight compared to the immediate ones. My training iterations kept converging faster as I kept increasing my discount factor and it converged in less than 900 iterations for a DF = 0.99. I ended up choosing that!

Stopping Criteria: As a means of safety and preventing any infinite non-converging trainings, I stop early based on a certain number of MAX\_EPISODES. Since my best training converges in less than 1000 iterations, I ended up choosing MAX\_EPISODES = 2000 just to be safe. My approach here was to track the average cumulative reward over a window of episodes and stop training when this average surpasses a certain threshold for a predefined number of consecutive episodes. This way, I’m ensuring that the policy is consistently performing well instead of a sparse reward signal before I can consider that my training is complete. Instead of storing 10 consecutive episodes and later explicitly calculating their average reward, I’m keeping track of 10 consecutive successes and then stopping it once reached average target reward. This way it uses less memory space, because consecutive\_successes is constant memory access.

My training:

A screen shot of a computer

Description automatically generated

My testing:

A computer screen with white text

Description automatically generated

I used epsilon = 0.1 for an epsilon-greedy policy, for reasons as stated above in Section 2.

# Section 4: Markov property

My model did not converge at all when I ran the given scripts. On a non-converging model, which was stopped early by my code, achieved a total reward of only 83.0. I chose input\_size = 2 for this, nb\_actions still remains 2. For simplicity, I stopped at 1000 episodes here instead of 2000 max\_episodes because I knew it was not going to converge regardless.

A screen shot of a computer screen

Description automatically generated

A computer screen with white text

Description automatically generated

The removal of accelerations from the state space affected the Markov property of the environment. In a Markov decision process (MDP), the current state should contain all relevant information necessary to make decisions, and the next state transition probabilities should depend only on the current and previous state and current action, not on the history of previous states and actions.

By removing accelerations, we reduced the dimensionality of the state space and simplified the problem. However, the removed information was essential for the agent to learn the dynamics of the environment accurately, and thus, it hindered convergence.

In the context of the Cartpole environment, accelerations are important additional information about the dynamics of the system, such as how fast the cart and pole are moving and how these velocities change over time. Without this information, the agent struggled to learn an effective policy, especially since precise control over velocity is necessary for balancing the pole.

To address this issue, I considered storing the previous variable information so that I could do manual re-evaluations of the environment without directly using the given accelerations.

After updating the environment with alternatives to given accelerations, I ran all 3 scripts and got the following output:

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

This training converged way faster than Section 3’s training.

A computer screen with white text

Description automatically generated

I chose input\_size = 4 for this, nb\_actions remains the same. This algorithm converged because I calculated the relative velocities based on the current and previous state values. I stored the previous state (x and theta) to compute velocities in the next step. This approach provides the policy network with information about the current motion (speed and direction) of the cart and pole without explicit accelerations, allowing it to make informed decisions without relying on past observations. This approach should create a Markovian environment as the state representation captures the essential information about the system's current dynamics.

Mistakes I made in this section:

* I forgot to put the correct input size when testing convergence on the given code initially and I kept getting a Runtime error of incorrect dimension match in NN and then I realized my error and rectified it. I used the right dimensions for this problem (which became 2 since we removed accelerations on the code provided)

# Section 5.3: Actor-critic algorithm

I have chosen to do part 3 of section 5 since the concept of actor-critic algorithm seemed challenging and interesting at the same time.

The one-step actor-critic method is a RL algorithm that combines both policy-based (actor) and value-based (critic) methods’ elements.

* Like before, the actor is responsible for selecting actions based on the current policy. It takes the current state as input and outputs a probability distribution over different actions that are possible. We then sample an action from this distribution.
* The critic is useful for evaluating the goodness of the actions selected by the actor. It takes the current state as input and outputs an estimate of the value, or expected return, associated with that state-action pair.
* The difference between the value estimate from the critic and the actual return received is used to calculate an estimate of the advantage.
* The actor's parameters are updated to maximize the expected advantage.
* The critic's parameters are updated to minimize the difference between the estimated value and the actual return. I perform gradient descent on the mean squared error between the estimated value and the actual return.

Why this method works better according to me:

* The actor-critic method helped me achieve a low variance of policy gradient estimates as you can see in the screenshot below. Since the critic provides estimates of the value function, the variance of the policy gradient estimates is lower compared to pure policy gradient methods.
* By updating both the policy and value function simultaneously, the actor-critic method learns more efficiently compared to our previous method of updating the policy alone.

Unexpected Observation:

* While training the actor-critic methods, I sometimes noticed a more unstable training compared to pure policy gradient methods. It took very long to converge sometimes, other times it was very quick. I think due to the interaction between the actor and critic updates, there might be some instability that gets induced into the problem.

Training:

A screen shot of a computer

Description automatically generated

Testing:

A computer screen with white text

Description automatically generated

The training and testing both converged gradually and with low variance, this section was really fun and learning about the critic’s role was super useful.

Mistakes I made in this section:

* I forgot to update the critic’s parameters and was not seeing expected results until I realized that it’s missing. I fixed it immediately and got expected results then.
* I kept getting type errors because of Torch vs numpy differences in function inputs and had to convert to appropriate types and dimensions to achieve the right gradients