SUBJECT NAME: REINFORCEMENT LEARNING

UNIT-3

TOPIC : POLICY FUNCTION USING NEURAL NETWORKS

The **policy function** implemented through neural networks in reinforcement learning is an approach where the network outputs actions or probabilities of actions directly, rather than predicting action values (as done in Q-learning). This shifts the learning paradigm from value-based methods (like DQN) to **policy-based methods** by approximating the policy directly.

Fig: A Q-network takes a state and returns Q values (action values) for each action. We can use those action values to decide which actions to take.

Policy Network

- A **policy network** accepts the current state as input and outputs a **probability distribution** over all possible actions.
- It defines a function π: State$\rightarrow$P(Action$\mid$State)
- The network encourages exploration through stochastic outputs, which allows it to learn optimal behavior by randomly sampling actions.

Example:
In a four-action environment (e.g., Gridworld), the policy network may predict:
P(A)=[0.25,0.25,0.10,0.40]

This indicates probabilities for actions [up, down, left, right] respectively. Higher probability actions are more likely to be selected but are still subject to randomness.

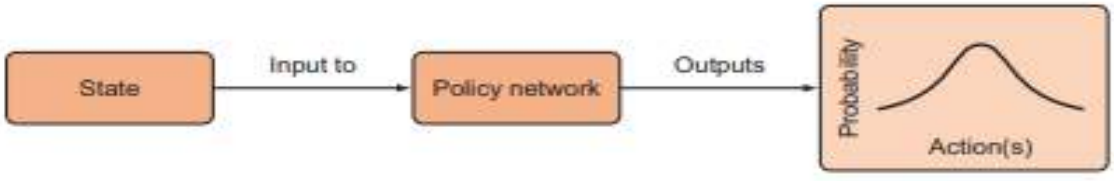


Fig: A policy network is a function that takes a state and returns a probability distribution over

the possible actions

Stochastic vs. Deterministic Policy

- **Stochastic Policy:** The output is a **probability distribution**, allowing for a non-deterministic selection of actions. This ensures that even suboptimal actions may be selected occasionally, which promotes exploration.

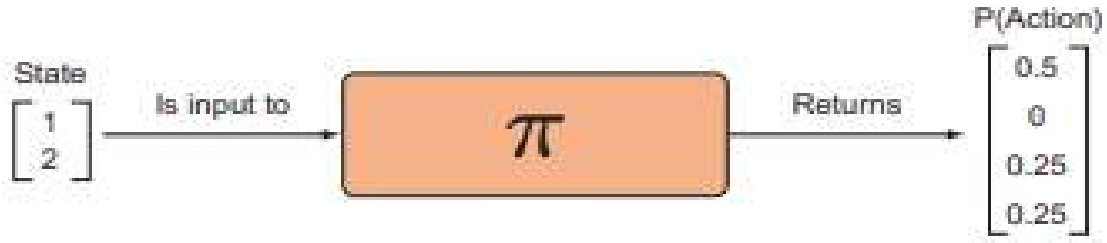


Fig: A stochastic policy function. A policy function accepts a state and returns a probability distribution over actions. It is stochastic because it returns a probability distribution over actions rather than returning a deterministic, single action.

- **Deterministic Policy:** The network always returns the same action for a given state. This can limit exploration and make training harder since it lacks variability.

Fig: A deterministic policy function, often represented by the Greek character pi, takes a state and returns a specific action to take, unlike a stochastic policy, which returns a probability distribution over actions.

Exploration and Convergence

- Early in training, the policy network's distribution is **uniform** (all actions have nearly equal probabilities). This enables **exploration** of different state-action pairs.
- As training progresses, the policy converges toward an **optimal distribution**. If one action is optimal, the probabilities will converge into a **degenerate distribution** (all mass on a single action).
- Stochastic policies enable sustained exploration even in dynamic environments.

Advantages of Policy Networks over Value-Based Methods

- **Simpler Action Selection:** The output is the action itself or a distribution over actions, eliminating the need for strategies like epsilon-greedy used in DQNs.
- **No Experience Replay Needed:** Policy networks directly optimize policies without requiring large memory for replay buffers, as used in Q-learning.
- **Natural Handling of Stochasticity:** With probability distributions over actions, policy networks naturally handle exploration during training.

Policy Gradient Methods

Policy gradient methods aim to optimize the policy function directly by maximizing the expected reward. These methods use the gradient of the **objective function** with respect to the network parameters to update the policy.

Example: Stochastic Policy Gradient

Given the state s, the network outputs a vector of probabilities for each action: $P(A \mid s)=[0.50,0.0,0.25,0.25]$
This means the agent has a 50% chance to go **up**, no chance to go **down**, and a 25% chance for **left** or **right**.

If the agent revisits the same state, the action may differ each time, adding variability to the training process.

Key Challenges in Training Policy Networks

- **Defining the Objective Function:** Unlike Q-learning, which uses mean squared error (MSE) between predicted and target Q-values, training a policy network requires optimizing the **expected cumulative reward** directly.
- **Exploration vs. Exploitation:** Early exploration helps the agent discover better strategies. However, the policy should converge toward exploitation (optimal behavior) over time.
- **Balancing Stochastic and Deterministic Outputs:** While stochastic policies encourage exploration, some environments benefit from deterministic policies, requiring fine-tuning.

Policy functions implemented with neural networks offer a powerful way to model actions directly in reinforcement learning. By outputting probability distributions, these networks inherently allow exploration, simplifying the complexity of action selection compared to value-based methods. Policy gradient algorithms are effective in optimizing these networks, with stochastic policies providing robustness in dynamic or uncertain environments.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-3
TOPIC : REINFORCING GOOD ACTIONS: THE POLICY GRADIENT ALGORITHM

The policy gradient algorithm is a class of reinforcement learning algorithms that aims to optimize a policy function, which outputs a probability distribution over actions given a state.

Introduction to Policy Gradient Algorithms

- **Objective:** Policy gradient methods aim to create a probability distribution over possible actions given the current state.
- **Approach:** Instead of predicting action values like Q-learning, these algorithms focus on directly optimizing the policy to encourage actions that lead to positive rewards.
- **Stochastic Policies:** Policy gradients commonly employ stochastic policies, meaning actions are sampled based on probabilities, ensuring adequate exploration of the environment.

Objective Function

The policy gradient algorithm requires a differentiable objective function with respect to the network weights (parameters). The objective is to maximize the expected reward by adjusting the policy based on the actions taken and the rewards received.

A policy network, denoted $\pi\theta(s)$, parameterized by weights θ, outputs a probability distribution over actions for a given state s.

Training Objective: Maximize the probability of actions that result in positive rewards.

- Example: An agent receives a reward of +10 for an action, encouraging the network to assign higher probability to that action in future encounters.

Figure: The general overview of policy gradients for an environment with four possible discrete actions. First we input the state to the policy network, which produces a probability distribution over the actions, and then we sample from this distribution to take an action, which produces a new state.

Action Reinforcement

To reinforce good actions, the algorithm updates the policy based on the rewards received. The last experience in an episode is particularly important, as it provides a basis for adjusting the action probabilities.

Example of Action Reinforcement

If the policy network produces an initial action probability distribution of [0.25,0.25,0.25,0.25] and the agent takes action 3, resulting in a reward of +10, the goal is to increase the probability of selecting action 3 in similar future states.

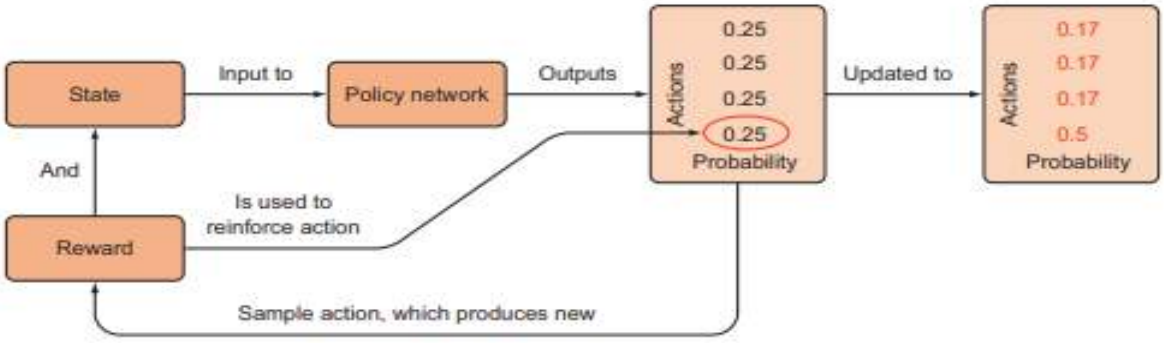


Figure: Once an action is sampled from the policy network's probability distribution, it produces a new state and reward. The reward signal is used to reinforce the action that was taken, that is, it increases the probability of that action given the state if the reward is positive, or it decreases the probability if the reward is negative. Notice that we only received information about action 3 (element 4), but since the probabilities must sum to 1, we have to lower the probabilities of the other actions

Using Log Probability for Optimization Stability

- **Issue with Probabilities:** Optimizing raw probabilities can be unstable due to their small numeric range.
- **Solution:** The **logarithm** of the probability (log probability) is used in the loss function: $\text{Loss} = -\log \pi_\theta(a \mid s)$
- **Advantage:** The log function stabilizes optimization by converting multiplications into sums, reducing numerical errors.

Credit Assignment

Not all actions in an episode deserve equal credit. Actions closer to the reward receive more credit than earlier ones. The algorithm must address the challenge of credit assignment, which involves determining how much credit to assign to each action taken during an episode. The objective function is adjusted to reflect the importance of actions based on their contribution to the final reward.

Exploration vs. Exploitation

A key aspect of the policy gradient algorithm is maintaining a balance between exploration (trying new actions) and exploitation (choosing known rewarding actions). Initially, the action selection is uniform, allowing the agent to explore the action space.

Stochastic vs. Deterministic Policies

While deterministic policies can lead to optimal actions, they may hinder exploration. Stochastic policies, on the other hand, retain some randomness, which is beneficial for exploring the environment effectively.

Conclusion

The policy gradient algorithm is a powerful approach in reinforcement learning that directly optimizes the policy for action selection. By reinforcing good actions and addressing credit assignment, the algorithm enables agents to learn from their experiences and improve their decision-making over time.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-3
TOPIC : THE REINFORCE ALGORITHM

The REINFORCE algorithm is one of the fundamental methods for training policy-based reinforcement learning models. It is a Monte Carlo-based policy gradient method that uses the reward signal to learn optimal policies by updating the parameters of a policy network.

Policy Network:

The policy network is a neural network that takes the current state as input and outputs a probability distribution over possible actions. For example, in the CartPole environment, the policy network might output a vector like [0.25, 0.75], indicating the probabilities of taking action 0 or action 1, respectively.

Action Selection:

Actions are selected based on the probability distribution provided by the policy network. The np.random.choice function is used to sample an action according to the predicted probabilities.

Environment Interaction:

The agent interacts with the environment using the step method, which returns the next state, reward, done flag (indicating if the episode has ended), and additional info. The reset method initializes the environment and returns the initial state.

Reward Calculation:

The algorithm computes discounted rewards from the sequence of rewards received during an episode. This transforms the linear sequence of rewards into an exponentially decaying sequence, which is then normalized to the interval [0, 1] .

Training Loop:

The training loop involves collecting states, actions, and rewards throughout an episode. Once the episode ends, the algorithm computes the discounted rewards and updates the policy network based on the actions taken and the rewards received.

Implementation Steps

Initialize the Environment:

Use OpenAI Gym to create the environment (e.g., env = gym.make('CartPole-v0')).

Setting Up the Policy Network

The policy network is a neural network that takes state vectors as inputs and outputs a probability distribution over the available actions. This network serves as the agent's policy.

python
Copy code

```
import torch
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(4, 150),        # Input layer: 4-dimensional state vector
    nn.LeakyReLU(),           # Activation function
    nn.Linear(150, 2),        # Output layer: 2 actions (left or right)
    nn.Softmax(dim=-1)        # Softmax to create a probability distribution
)
```

- Input Layer: Takes the 4-dimensional state vector (e.g., for the CartPole problem).
- Hidden Layer: A fully connected layer with 150 neurons and Leaky ReLU activation.
- Output Layer: Outputs probabilities for two possible actions using Softmax.

The learning rate for training is set using an optimizer, such as Adam:

Python Code

```
learning_rate = 0.0009
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Interacting with the Environment

The agent interacts with the environment using the policy network to choose actions based on the state.

Python Code

```
import numpy as np

# Example interaction
state = env.reset()  # Reset environment to initial state
pred = model(torch.from_numpy(state).float())  # Predict action probabilities
```

```
action = np.random.choice([0, 1], p=pred.data.numpy())  # Sample an action
state, reward, done, info = env.step(action)  # Take the action in the environment
```

- State: Current state from the environment.
- Action: Chosen based on the predicted probabilities.
- Reward: Immediate reward received after taking the action.
- Done: Boolean indicating if the episode has ended.
- Info: Diagnostic information (not used in this case).

Training the Policy Network

Training the REINFORCE algorithm involves calculating the loss using the rewards collected over an episode and then using gradient descent to update the network parameters.

Calculating the Loss

The loss function for REINFORCE is based on the negative log probability of the actions taken, weighted by the discounted returns.

Python Code

```
def loss_fn(preds, rewards):
    return -1 * torch.sum(rewards * torch.log(preds))
```

This loss function ensures that actions with higher rewards increase in probability, while actions with lower rewards decrease in probability.

Computing Discounted Rewards

Rewards are discounted to account for the long-term impact of actions. The discount factor, gamma, typically ranges between 0 and 1.

Python Code

```
def discount_rewards(rewards, gamma=0.99):
    len_r = len(rewards)
    disc_return = torch.pow(gamma, torch.arange(len_r).float()) * rewards
    disc_return /= disc_return.max()  # Normalization for stability
    return disc_return
```

- Discount Factor: Exponentially decays the reward over time to prioritize earlier rewards.
- Normalization: Keeps rewards within a stable range to ensure efficient learning

Full Training Loop

The training loop collects experiences, computes the loss, and updates the policy network.

python
Copy code

```
MAX_EPISODES = 500
gamma = 0.99
optimizer = torch.optim.Adam(model.parameters(), lr=0.0009)

for episode in range(MAX_EPISODES):
    state = env.reset()  # Reset the environment
    done = False
    transitions = []

    while not done:
        action_probs = model(torch.from_numpy(state).float())
        action = np.random.choice([0, 1], p=action_probs.data.numpy())
        next_state, reward, done, _ = env.step(action)
        transitions.append((state, action, reward))
        state = next_state

    # Collect rewards, states, and actions for the episode
    rewards = torch.Tensor([t[2] for t in transitions])
    disc_rewards = discount_rewards(rewards, gamma)

    states = torch.Tensor([t[0] for t in transitions])
    actions = torch.Tensor([t[1] for t in transitions]).long()

    # Compute action probabilities and subset for the taken actions
    preds = model(states)
    action_probs = preds.gather(1, actions.view(-1, 1)).squeeze()
    loss = loss_fn(action_probs, disc_rewards)

    # Update the network
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Explanation of the Training Loop

1. Initialization: Reset the environment and start a new episode.
2. Collect Transitions: For each time step, use the policy network to choose an action, execute it, and store the state, action, and reward.
3. Calculate Discounted Rewards: Compute the total rewards, applying the discount factor to emphasize earlier rewards.
4. Compute Loss: Use the negative log probability of the chosen actions, weighted by the rewards.
5. Backpropagation: Use the optimizer to update the policy network's parameters.

The REINFORCE algorithm is a simple yet effective approach for policy optimization in environments with a small action space. While it performs well for tasks like CartPole, more complex environments may require advanced methods for better efficiency and stability.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-3
TOPIC : TACKLING MORE COMPLEX PROBLEMS WITH ACTOR-CRITIC METHODS

Actor-critic methods are a significant advancement in reinforcement learning (RL), particularly useful for tackling complex environments where simpler algorithms like REINFORCE or Deep Q-Networks (DQN) face limitations. These methods combine the benefits of value-based and policy-based approaches to enhance sample efficiency and stability.

REINFORCE Algorithm:

The REINFORCE algorithm is a vanilla policy gradient method that updates model parameters only after completing an entire episode. This episodic nature allows for the collection of rewards throughout the episode, which are then used to update the policy.

Limitations of REINFORCE:

While effective for simple tasks, REINFORCE struggles with more complex environments due to its episodic nature and the variance in reward signals. This can lead to inefficient learning and slow convergence.

Actor-Critic Models:

Actor-critic methods combine the benefits of both value-based and policy-based approaches. The "actor" refers to the policy function that selects actions, while the "critic" evaluates the actions taken by the actor by estimating the value function.

This combination allows for more stable and efficient learning, as the critic can provide feedback to the actor, reducing the variance of the policy updates.

Advantages of Actor-Critic Methods:

Sample Efficiency: By using a critic to evaluate actions, the model can learn from fewer samples, improving the overall efficiency of the learning process.

Reduced Variance: The critic helps to stabilize updates by providing a more accurate estimate of the value of actions, which mitigates the high variance often seen in policy gradient methods.

Direct Sampling: Unlike DQN, which requires a separate policy function (e.g., epsilon-greedy), actor-critic methods allow for direct sampling from the probability distribution over actions, simplifying the action selection process.

Advantage Function for Efficiency:

The advantage function is introduced to speed up convergence. It quantifies how much better an action is compared to the average, which helps in fine-tuning the policy more effectively.

This results in more stable and faster learning as updates become more focused on improving actions with higher advantages.

Online Learning and Sample Efficiency:

Actor-critic methods often operate in an online fashion, updating parameters incrementally. This is a contrast to Monte Carlo approaches like REINFORCE, which require full episodes to be sampled before updates.

Distributed implementations, like Distributed Advantage Actor-Critic (DA2C), eliminate the need for experience replay buffers, commonly used in DQN, making the learning process more efficient.

Parallelizing Training:

Actor-critic models can leverage parallelization to speed up training. This approach distributes the computation across multiple instances of the environment, collecting more experience in less time and thus improving the model's performance more rapidly.

Combining Value and Policy Functions

Actor-critic methods merge the strengths of policy gradients and Q-learning:

Q-learning (e.g., DQN): Learns the value of actions based on expected rewards and chooses actions that maximize these values. It requires policies like epsilon-greedy and a replay buffer to handle variance.

Policy Gradients: Reinforce actions leading to positive outcomes and penalize negative ones. These methods directly sample actions from the policy, leading to a more natural strategy selection.

By integrating these approaches:

Actor (Policy Function): Learns to select actions by optimizing a probability distribution.

Critic (Value Function): Assesses the action taken, providing feedback that improves the policy's efficiency and stability.

Example Applications

Game Environments: In tasks like pinball, the actor learns to operate controls optimally, while the critic evaluates the outcomes, helping refine the actor's strategy.

Complex Robotics: Real-world robotic control problems benefit from the stability and efficiency of actor-critic methods.

Mathematical Formulation

Policy Loss Function:
$\text{Loss} = -\log(P(a|S)) \times R$

Here, minimizing the negative log-probability of actions taken, scaled by the reward, encourages the selection of actions that yield high rewards. Using logarithms ensures numerical stability and efficient gradient computation.

Advantage Function: The advantage function, A(s,a), helps refine the policy by considering how much better an action is compared to others, leading to faster convergence and more effective learning.

Actor-critic methods are a sophisticated RL approach that balance the sample efficiency of value-based methods with the flexibility of policy gradients. They are well-suited for non-episodic environments and can be scaled using distributed training to solve real-world problems effectively.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-3

TOPIC : COMBINING VALUE AND POLICY FUNCTIONS IN REINFORCEMENT LEARNING

In reinforcement learning, two primary methods are used for training agents: value-based and policy-based methods. Value-based methods, like Q-learning, predict the expected rewards (values) of actions, while policy-based methods learn a probability distribution over actions, directly reinforcing actions that yield positive rewards. The approach of combining these two methodologies aims to leverage the strengths of each, producing an efficient learning algorithm known as the "actor-critic" model.

Value Function:
The value function estimates the expected return (reward) from a given state or state-action pair. It can be represented as:

- **State-Value Function (V(S))**: The expected return from state S.
- **Action-Value Function (Q(S, A))**: The expected return from taking action A in state S.

Policy Function:
The policy function defines the behavior of the agent by mapping states to actions. It can be deterministic or stochastic, providing a probability distribution over actions given a state.

Benefits of Combining Value and Policy Functions:

1. **Sample Efficiency**: By combining these functions, we can make updates more frequently, increasing sample efficiency.
2. **Variance Reduction**: Using the value function reduces the variance in rewards that train the policy, yielding more stable training.

Concept of Advantage:

To integrate both approaches, the **Advantage** function is used. It quantifies how much better a particular action is compared to the expected baseline reward. Mathematically, this advantage is represented as:

$$\text{Advantage} = R - V(S)$$

where R is the observed return and V(S) is the value of the current state. By using this advantage term, the policy function updates focus on actions that perform better than expected.

Actor-Critic Model:

Actor: The policy network, which generates actions based on the policy learned.

Critic: The value network, which evaluates actions taken by the actor and provides feedback in terms of advantages. This feedback is integrated into the actor's loss function, helping it to refine its action choices.

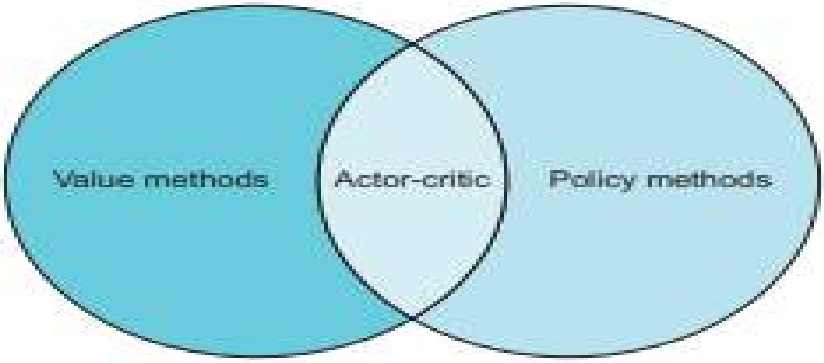


Figure: Q-learning falls under the category of value methods, since we attempt to learn action values, whereas policy gradient methods like REINFORCE directly attempt to learn the best actions to take. We can combine these two techniques into what's called an actor-critic architecture.

Implementation in Neural Networks:

Instead of a simple look-up table, neural networks are used to parameterize both the actor and the critic functions. The actor's network produces action probabilities, while the critic's network predicts the state or action values, which serve as baselines for the advantage calculation.

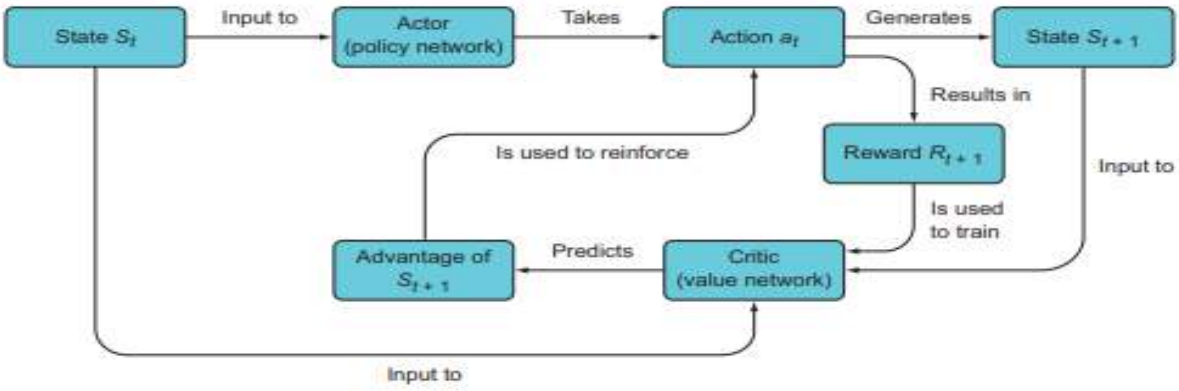


Figure: The general overview of actor-critic models. First, the actor predicts the best action and chooses the action to take, which generates a new state. The critic network computes the value of the old state and the new state. The relative value of St+1 is called its advantage, and this is the signal used to reinforce the action that was taken by the actor.

Advantages of the Actor-Critic Approach:

- **Reduced Reward Sparsity Issue**: The critic can guide the actor even when rewards are sparse, making it suitable for environments with infrequent reward signals.
- **Bootstrapping**: The critic leverages bootstrapping, where predictions for current states help in estimating future states. This process reduces variance at the cost of introducing bias but enhances overall training stability.

Bias-Variance Tradeoff:

Combining value and policy functions balances the high-bias, low-variance nature of value predictions with the low-bias, high-variance nature of policy predictions. This balance results in a model with moderate bias and variance, optimizing the agent's learning performance in complex, dynamic environments.

Combining value and policy functions in actor-critic models allows agents to take advantage of value-based stability and policy-based flexibility. Through advantage calculations, frequent updates, and reduced variance, this approach facilitates efficient, stable learning in reinforcement learning environments.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-3

TOPIC : DISTRIBUTED TRAINING

Distributed training is a technique used in machine learning to improve the efficiency and speed of model training by leveraging multiple processing units. This approach is particularly beneficial for deep learning models, where the computational demands can be significant. The distributed advantage actor-critic (DA2C) model is a specific implementation that combines the advantages of actor-critic methods with distributed training.

Batch Training and Gradient Averaging:

In deep learning, batch training is essential to average out noise and obtain stable gradient updates. Training with single samples can lead to high variance in gradients, preventing convergence. By batching, we compute gradients over a group of samples to average noise, allowing stable parameter updates.

Reinforcement Learning and Experience Replay:
In reinforcement learning, experience replay buffers are often used to store past experiences, allowing the model to learn from them. However, maintaining a sufficiently large replay buffer can be memory-intensive and impractical in complex environments.

The Markov property, which states that the optimal action for a state can be computed without reference to prior states, is crucial for simpler games but may not hold in more complex scenarios. In such cases, recurrent neural networks (RNNs) can be employed to retain information about past states .

Parallel Agents and Environments:

Distributed training circumvents the need for a replay buffer by running multiple copies of the agent in parallel across different processes. Each agent explores a separate instance of the environment, allowing experiences to be gathered simultaneously. The gradients from each agent are averaged, reducing variance and stabilizing learning.

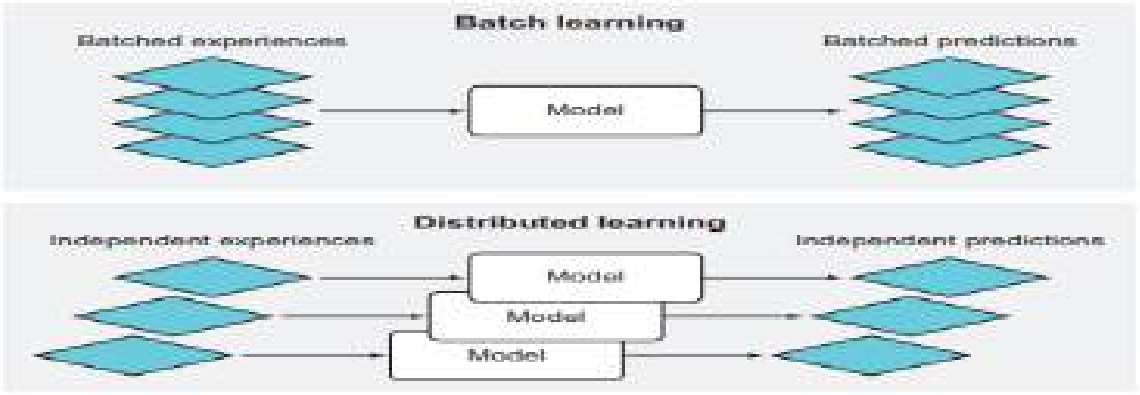
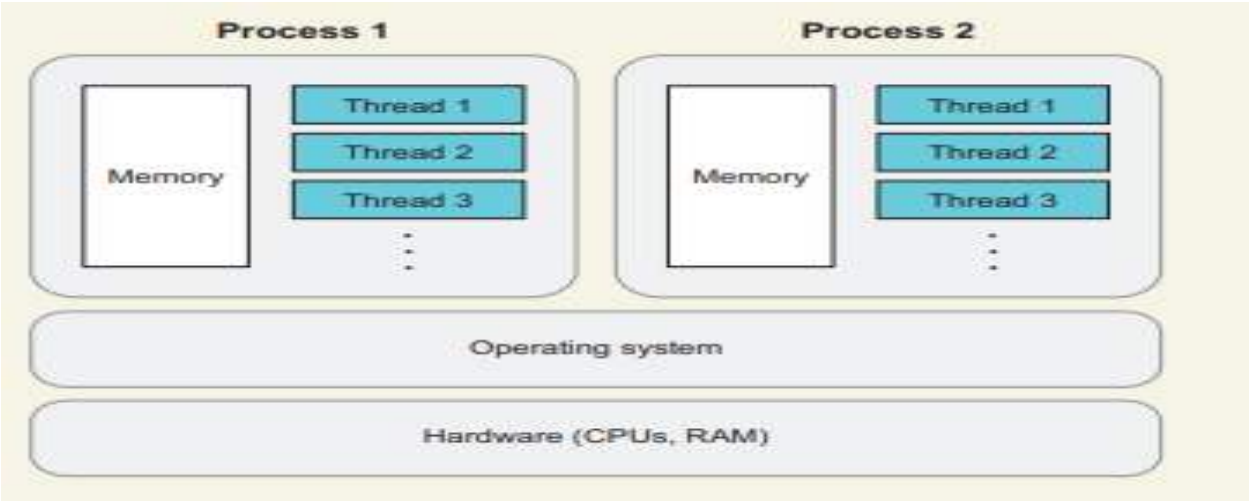


Figure: The most common form of training a deep learning model is to feed a batch of data together into the model to return a batch of predictions. Then we compute the loss for each prediction and average or sum all the losses before backpropagating and updating the model parameters. This averages out the variability present across all the experiences. Alternatively, we can run multiple models with each taking a single experience and making a single prediction, backpropagate through each model to get the gradients, and then sum or average the gradients before making any parameter updates.

Multiprocessing vs. Multithreading:

- **Multiprocessing**: True parallelism, where computations are handled on distinct CPU cores, is ideal for machine learning, which demands high computational throughput. In Python, the multiprocessing library supports running processes across multiple cores.
- **Multithreading**: Although efficient for I/O-bound tasks, multithreading is not true parallelism; instead, it involves switching between tasks, which limits performance gains for compute-intensive tasks.



Python Implementation:

Using Python's multiprocessing library, we can distribute reinforcement learning computations across several processes. Each process independently computes gradients based on its observations, and the gradients are aggregated to update the model's parameters. An example using the multiprocessing library demonstrates dividing tasks across CPU cores, improving computational efficiency.

Benefits of Distributed Training:

- **Eliminates Replay Buffer Requirements**: By distributing agents, experience replay can be avoided, making training fully online without the need to revisit past experiences.
- **Low Variance Gradients**: Aggregating gradients across agents provides a low-variance gradient estimate, which is beneficial for stable learning.
- **Efficiency**: Distributing computation across multiple cores accelerates learning, especially in environments where agents can gather diverse experiences simultaneously.

Distributed training in DA2C combines the benefits of the advantage actor-critic model with multiprocessing. This approach allows reinforcement learning algorithms to scale across multiple agents, enabling efficient, stable learning without replay buffers, even in non-Markovian or complex environments.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-3

TOPIC : ADVANTAGE ACTOR-CRITIC

The Advantage Actor-Critic (A2C) is a reinforcement learning algorithm that combines the benefits of both policy-based and value-based methods. It utilizes two neural networks: the actor, which proposes actions, and the critic, which evaluates the actions taken by the actor. This approach allows for more efficient learning by leveraging the advantages of both methods.

Actor

- The actor is responsible for selecting actions based on the current policy.
- It outputs a probability distribution over possible actions given the current state.
- The actor is trained to maximize the expected return by adjusting its policy based on feedback from the critic.

Critic

- The critic evaluates the action taken by the actor by estimating the value of the current state.
- It computes the advantage, which is the difference between the expected return and the value predicted by the critic.
- The critic's loss is computed without backpropagating through the actor's network, ensuring that the actor's weights are not modified during the critic's update.

Advantage Calculation: Advantage A(s,a) is defined as:
$A(s,a)=r+\gamma V(s')-V(s)$
where:

    - r is the reward,
    - V(s) is the predicted value of the current state, and
    - V(s′) is the predicted value of the next state.
- The advantage reflects the difference between the observed and expected values, making the training more efficient by focusing on deviations from typical outcomes.

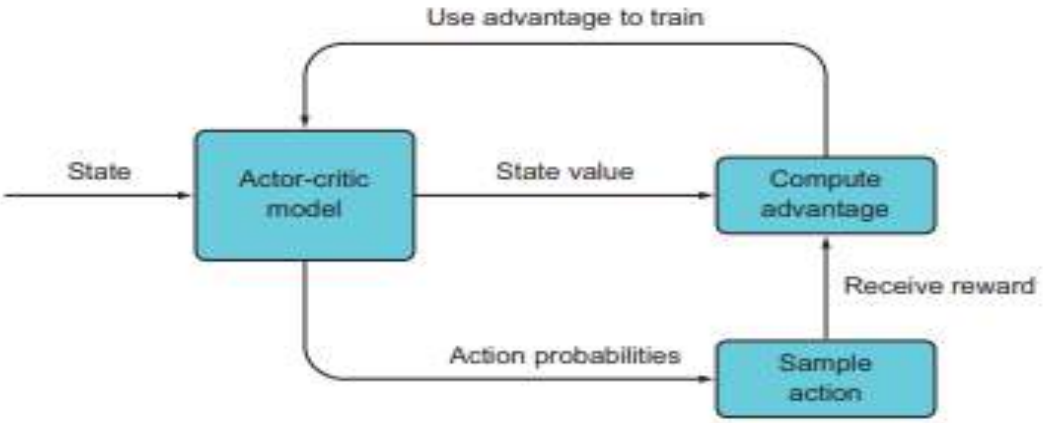


Figure: An actor-critic model produces a state value and action probabilities, which are used to compute an advantage value and this is the quantity that is used to train the model rather than raw rewards as with just Q-learning.

Algorithm Steps

1. **Initialization**: Set the discount factor γ and initialize the actor and critic networks.
2. **Episode Loop**:
    - Obtain the current state from the environment.
    - Use the critic to estimate the value of the current state.
    - Use the actor to sample an action based on the current policy.
    - Execute the action in the environment to receive the next state and reward.
    - Estimate the value of the next state using the critic.
    - Calculate the advantage using the reward and the values from the critic.
    - Compute the loss for the actor based on the advantage and update the actor's weights.
3. **Training**: Repeat the process for multiple epochs to improve the policy and value estimates.

Implementation Example

The following pseudocode illustrates the basic structure of the Advantage Actor-Critic algorithm:

```
gamma = 0.9
for i in epochs:
   state = environment.get_state()
   value = critic(state)
   policy = actor(state)
   action = policy.sample()
   next_state, reward = environment.take_action(action)
   value_next = critic(next_state)
```

```
advantage = reward + (gamma * value_next - value)
loss = -1 * policy.logprob(action) * advantage
minimize(loss)
```

Training Dynamics

- The actor and critic have an adversarial relationship, where the actor aims to improve its policy while the critic tries to accurately predict state values.
- This competition can lead to chaotic loss plots, but the overall performance of the agent improves over time.

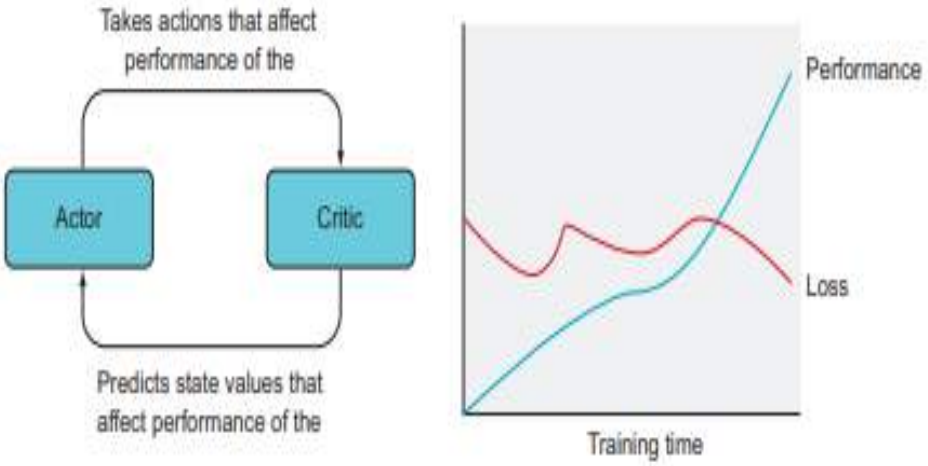


Figure: The actor and critic have a bit of an adversarial relationship since the actions that the agent take affect the loss of the critic, and the critic makes predictions of state values that get incorporated into the return that affects the training loss of the actor. Hence, the overall loss plot may look chaotic despite the fact that the agent is indeed increasing in performance.

Key Features:

- **Two-Headed Model**: A2C can utilize a neural network with two outputs: one for the actor and one for the critic. This allows shared parameters between the actor and critic for efficiency.
- **Asynchronous Updates**: In distributed setups, multiple agents interact with the environment, asynchronously updating the shared model parameters after each episode.

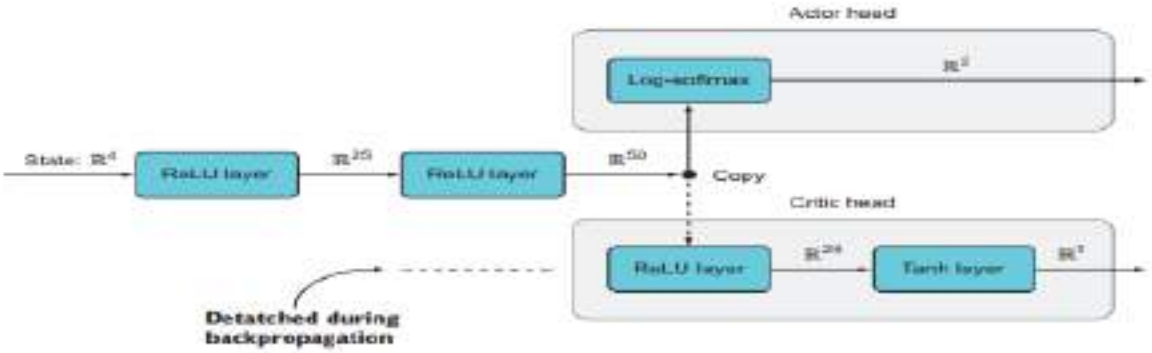


Figure: This is an overview of the architecture for our two-headed actor-critic model. It has two shared linear layers and a branching point where the output of the first two layers is sent to a logsoftmax layer of the actor head and also to a ReLU layer of the critic head before finally passing through a tanh layer, which is an activation function that restricts output between –1 and 1. This model returns a 2-tuple of tensors rather than a single tensor. Notice that the critic head is detached (indicated by the dotted line), which means we do not backpropagate from the critic head into the actor head or the beginning of the model. Only the actor backpropagates through the beginning of the model

Applications: A2C is commonly applied to environments such as CartPole and more complex games in OpenAI Gym, making it flexible across different levels of complexity.

N-Step Learning: In an extension of A2C called N-step Actor-Critic, the model updates after a predefined number N of steps rather than waiting for a full episode, balancing the trade-off between sample efficiency and update frequency.

The Advantage Actor-Critic method is a powerful reinforcement learning technique that effectively combines the strengths of both policy and value-based approaches. By utilizing both an actor and a critic, it can learn more efficiently and adaptively in complex environments.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-3

TOPIC : N-STEP ACTOR-CRITIC

The N-step Actor-Critic is a reinforcement learning (RL) extension that modifies the standard actor-critic model by incorporating updates after a fixed number of steps, N, instead of waiting until the end of an episode. This allows for more frequent updates and a balance between sample efficiency and variance reduction.

N-step Learning:

- N-step learning involves calculating the loss and updating parameters after N steps, where N can be adjusted based on the specific requirements of the task.
- If N is set to 1, it represents fully online learning, while a very large N approaches Monte Carlo learning. The optimal N typically lies somewhere in between, allowing for more frequent updates without the high variance associated with Monte Carlo methods.

Bootstrapping:

- Bootstrapping is a technique where the critic uses its own predictions to estimate future returns. This is particularly useful in N-step learning, as it allows the critic to make more accurate predictions by incorporating information from multiple steps.
- Without bootstrapping, the critic relies solely on actual returns, which can lead to high variance and instability in training.

Critic Model:

- The introduction of a critic model helps to reduce the variance of policy function updates by directly modeling the state value. This moderation prevents large swings in parameter updates that could destabilize learning.
- The advantage function is used to train the policy based on how much better or worse an action performed compared to the critic's prediction, rather than on raw returns.

Purpose of N-Step Learning:

- **Bias-Variance Trade-Off**: Single-step (1-step) updates are quick but can introduce bias, while full-episode (Monte Carlo) updates capture all rewards but can have high variance. N-step learning finds a middle ground by adjusting N based on the task.

- **Bootstrapping**: With N-step bootstrapping, we estimate future rewards based on both observed rewards and predicted values from the critic, enhancing stability and reducing variance.

Workflow of N-Step Actor-Critic:

1. **Initialize Hyperparameters**: Set N, the discount factor γ, and the learning rate.
2. **Run Episode for N Steps**:
    - Get the current state, predict the action probabilities and state value from the actor and critic, and store them.
    - Sample an action from the actor's policy, execute it in the environment, and receive the next state and reward.
    - Continue for N steps or until the episode ends.
    - If the episode ends before N steps, set the return for that step to zero; otherwise, use the critic's predicted value for bootstrapping.
3. **Calculate Return and Advantage**:
    - Compute the return G for the N-step sequence by adding rewards and bootstrapping from the critic's value.
    - Calculate the advantage A(s,a), which adjusts the return by comparing it with the critic's baseline.
4. **Update Actor and Critic**:
    - **Actor Loss**: Update based on the advantage. $\text{loss}_{\text{actor}} = -\log(\pi(a|s)) \cdot A(s,a)$
    - **Critic Loss**: Minimize the squared error between predicted and actual returns. $\text{loss}_{\text{critic}} = (G - V(s))^2$**Optimize**: Minimize the combined loss function.

Example Code for N-Step Learning: The N-step update process primarily modifies the run_episode function to track N steps and bootstraps from the last state if the episode does not end.

Python Code

```
def run_episode(env, model, N_steps=10):
    state = torch.from_numpy(env.state).float()
    values, logprobs, rewards = [], [], []
    done = False
    G = torch.Tensor([0])
    for _ in range(N_steps):
        policy, value = model(state)
        values.append(value)
        action = policy.sample()
        logprobs.append(policy.log_prob(action))

        # Take action and observe outcome
        state, reward, done, _ = env.step(action.item())
        rewards.append(reward)

        # Bootstrapping if episode not done
        if done:
```

```
            G = 0
            break
        G = value.detach()  # Last value for bootstrapping

    return values, logprobs, rewards, G
```

Benefits of N-Step Learning:

- **Improved Stability**: By blending observed rewards and predictions, N-step learning stabilizes the critic's training, leading to smoother learning curves.
- **Sample Efficiency**: Reduces variance in the training data, allowing the model to converge more quickly than Monte Carlo updates.

Use Cases: N-step learning is ideal for complex environments, such as high-dimensional games where faster updates and efficient sample use are necessary for timely convergence and practical implementation.

N-step actor-critic methods provide a robust framework for reinforcement learning, combining the benefits of both online learning and Monte Carlo methods. By leveraging bootstrapping and a critic model, these methods enhance the stability and efficiency of training, making them a valuable tool for complex problem-solving in reinforcement learning.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-3
TOPIC : WORKING WITH OPENAI GYM

OpenAI Gym is an open-source suite of environments designed for testing reinforcement learning algorithms. It provides a common API that allows users to interact with various environments, making it an ideal platform for developing and evaluating reinforcement learning models.

Key Features

- **Variety of Environments**: The Gym contains hundreds of environments (797 in version 0.9.6), ranging from simple tasks to complex challenges that require sophisticated deep reinforcement learning approaches.
- **Common API**: A standardized interface for interacting with different environments, simplifying the process of testing and comparing algorithms.

Environment Categories
OpenAI Gym environments are categorized into several types:

- **Algorithms**
- **Atari games**
- **Box2D simulations**
- **Classic control problems**
- **MuJoCo robotic environments**
- **Robotics tasks**
- **Toy text simulations**

You can check the full list of supported environments on the OpenAI Gym website or through Python using:Python code

```
from gym import envs
print(envs.registry.all())
```

Example Environment: CartPole
One of the simplest environments to start with is CartPole, which falls under the Classic Control category. The objective is to balance a pole on a cart by applying small left or right movements.

- **Actions**: The environment has two discrete actions:
    - Action 0: Push left
    - Action 1: Push right
- **State Representation**: The state is represented as a vector of length 4, indicating:

    - Cart position
    - Cart velocity
    - Pole angle
    - Pole velocity
- **Reward Structure**: The agent receives a reward of +1 for every time step the pole remains upright. The episode ends when the pole falls over or the cart moves outside a specified window.

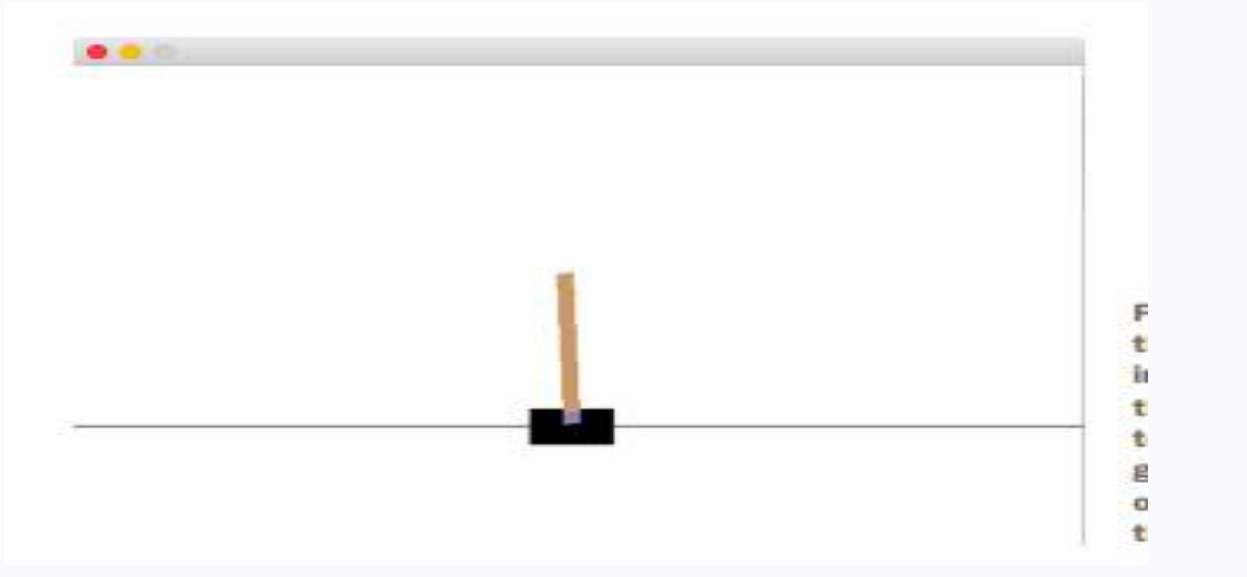


Figure: A screenshot from the CartPole game environment in OpenAI Gym. There is a cart that can roll left or right, and on top of it is a pole on a pivot. The goal is to balance the pole upright on the cart by carefully moving the cart left or right.

Important Methods:

1. **Initialization and Reset**:
    - Use env.reset() to initialize or reset the environment. It returns the initial state.
2. **Sampling Actions**:
    - Actions can be sampled using env.action_space.sample() to explore random actions. Eventually, you'll use an RL agent to determine the best actions.
3. **Step Method**:
    - The core method to interact with the environment is env.step(action), which returns:
        - state: The next state after taking the action.
        - reward: The reward received for the action.
        - done: A boolean indicating if a terminal state is reached.

        - info: A dictionary with additional debug information.

Interacting with the Environment
To interact with an environment in OpenAI Gym, follow these steps:
Import Gym:
import gym
Create the Environment:
env = gym.make('CartPole-v0')
Reset the Environment: This initializes the environment and returns the initial state.
state = env.reset()
Sample an Action: Use the action space to sample a random action.
action = env.action_space.sample()
Take a Step: Execute the action and receive feedback from the environment.
state, reward, done, info = env.step(action)

- **Parameters Returned**:
    - state: The next state after the action is taken.
    - reward: The reward received at that time step.
    - done: A boolean indicating if the episode has ended.
    - info: A dictionary with diagnostic information (not always used).

OpenAI Gym provides a robust framework for developing and testing reinforcement learning algorithms. By utilizing its diverse environments, such as CartPole, users can gain valuable insights into the performance of their models and refine their approaches to solving complex problems in reinforcement learning.