**Thomas Martin**

**CS-370-11456-M01 Current/Emerging Trends in CS**

**6-2 Assignment: Cartpole Revisited**

**Southern New Hampshire University**

**August 10, 2024**

The CartPole problem is a classic reinforcement learning task where the objective is to balance a pole on a cart by moving the cart left or right. In this context, we will evaluate the performance of two algorithms—REINFORCE and A2C—in solving this problem. Our primary goal is to determine which algorithm excels in terms of learning efficiency, stability, and overall effectiveness.

To effectively compare the REINFORCE and A2C algorithms, we will focus on key metrics such as average episode length, total rewards per episode, and learning stability. Our approach involves recording variance, plotting rewards and episode lengths, and analyzing the learning curves. The final conclusions will be drawn from a statistical analysis of average performance, variance, and convergence speed, supported by both numerical metrics and visualizations. These plots and statistical summaries will allow us to determine which algorithm performs better overall, which one is more stable, and which one learns faster.

**REINFORCE Algorithm**

The REINFORCE algorithm is a Monte Carlo-based policy gradient method that seeks to optimize the policy by adjusting its parameters in the direction that increases the expected reward. The policy network, a neural network, maps states action probabilities. It evaluates the current state (e.g., the position and velocity of the cart and pole) as input.

**class PolicyNetwork(nn.Module):**

**def \_\_init\_\_(self, num\_inputs, num\_actions, hidden\_size):**

**super(PolicyNetwork, self).\_\_init\_\_()**

**self.linear1 = nn.Linear(num\_inputs, hidden\_size)**

**self.linear2 = nn.Linear(hidden\_size, num\_actions)**

**def forward(self, state):**

**x = F.relu(self.linear1(state))**

**x = F.softmax(self.linear2(x), dim=1)**

**return x**

For each step in an episode, the policy network generates a probability distribution over the possible actions based on the current state. The action with the highest probability is more likely to be chosen.

**def get\_action(state, policy\_network):**

**state = torch.from\_numpy(state).float().unsqueeze(0)**

**probs = policy\_network(state)**

**action = np.random.choice(num\_actions, p=np.squeeze(probs.detach().numpy()))**

**return action, torch.log(probs.squeeze(0)[action])**

Once an action is chosen, it is executed in the environment, and the resulting state and reward are logged and stored for later use in updating the policy. At the end of the episode, the discounted returns are computed, and the policy network is updated using the policy gradient. This process is repeated over multiple episodes, allowing the policy network to learn how to maximize cumulative rewards, leading to the agent successfully balancing the pole for longer periods.

The following graph shows the steps per episode for the REINFORCE method over 5000 episodes. Initially, the steps per episode increase rapidly as the agent learns, reaching the maximum episode length of 200 steps. The progress is more gradual, and there is significant variance in performance during training.

A blue and orange graph

Description automatically generated

**REINFORCE Algorithm Pseudocode Summary**

**Initialize policy network with random weights θ**

**for each episode do:**

**Initialize environment and get initial state s0**

**for each time step t do:**

**Select action a\_t based on the policy πθ(a\_t | s\_t)**

**Execute action a\_t and observe reward r\_t and new state s\_t+1**

**Store log probability logπθ(a\_t | s\_t) and reward r\_t**

**If episode is done, break**

**Compute discounted returns G\_t for each time step t**

**Normalize returns**

**Compute policy gradient ∇θJ(θ) = Σ(logπθ(a\_t | s\_t) \* G\_t)**

**Update policy network parameters θ using gradient ascent**

**A2C Algorithm**

The CartPole problem can also be effectively solved using the Advantage Actor-Critic (A2C) algorithm, a reinforcement learning approach that integrates both value-based and policy-based methods. The A2C algorithm utilizes two neural networks: the Policy (Actor) Network and the Value (Critic) Network. The Policy Network learns a policy that maps states to actions, outputting the probability of selecting each action given the current state, while the Value Network estimates the value of the current state, which represents the expected sum of future rewards starting from that state.

The key advantage of using these two networks is that the critic’s value estimate is used to compute an advantage function. This function indicates whether the action taken by the actor was better or worse than expected, based on the critic's evaluation. The actor’s policy is then updated to favor actions that yield higher advantages. Essentially, the actor and critic work together, with the critic guiding the actor's policy updates.

Here is how the Actor-Critic network might be structured in code:

**class ActorCritic(nn.Module):**

**def \_\_init\_\_(self, num\_inputs, num\_actions, hidden\_size):**

**super(ActorCritic, self).\_\_init\_\_()**

**self.common = nn.Linear(num\_inputs, hidden\_size)**

**self.actor = nn.Linear(hidden\_size, num\_actions)**

**self.critic = nn.Linear(hidden\_size, 1)**

**def forward(self, state):**

**x = F.relu(self.common(state))**

**action\_probs = F.softmax(self.actor(x), dim=1)**

**state\_value = self.critic(x)**

**return action\_probs, state\_value**

During each step of an episode, both networks assess the current state and determine the appropriate action. The chosen action is executed in the environment, and the resulting new state, reward, and done flag (indicating whether the episode has ended) are recorded. The actor network is updated using a policy gradient that is adjusted by the computed advantage, while the critic network is updated by minimizing the mean squared error between its predicted value and the actual observed return. This process is repeated across many episodes until the actor network consistently learns to select actions that maximize cumulative rewards.

In summary, the A2C algorithm effectively addresses the CartPole problem by leveraging the strengths of both policy and value-based learning. The actor network progressively learns to select actions that optimize future rewards, while the critic network offers valuable feedback to refine the actor's choices. This synergy results in a robust and efficient learning process, enabling the agent to achieve better performance over time.

The following graphs illustrate the performance of the A2C method. Over 500 episodes, there's a noticeable increase in episode lengths as the agent improves its policy, with average episode lengths showing a consistent upward trend despite some variability. Similarly, the rewards received per episode also display an upward trend, indicating gradual improvement, though variability remains. The similarity between the episode length and total reward graphs is expected in environments like CartPole, where the reward structure is directly tied to the duration of the episode. Both metrics essentially measure the same aspect of the agent's performance, just from slightly different perspectives.

A graph of blue and orange lines

Description automatically generated

A graph of blue and orange lines

Description automatically generated

**A2C Algorithm Pseudocode Summary**

**Initialize actor-critic network with random weights θ (actor) and φ (critic)**

**for each episode do:**

**Initialize environment and get initial state s0**

**for each time step t do:**

**Select action a\_t based on policy πθ(a\_t | s\_t)**

**Execute action a\_t and observe reward r\_t and new state s\_t+1**

**Estimate state value Vφ(s\_t) and Vφ(s\_t+1)**

**Compute advantage A\_t = r\_t + γVφ(s\_t+1) - Vφ(s\_t)**

**Store log probability logπθ(a\_t | s\_t), state value Vφ(s\_t), and reward r\_t**

**If episode is done, break**

**Compute actor loss using advantage A\_t**

**Compute critic loss using temporal difference error**

**Update actor-critic network parameters θ and φ using gradient descent**

**Comparison of Approaches**

Policy gradient and value-based approaches, such as Q-learning, are two core strategies in reinforcement learning for developing optimal policies that maximize cumulative rewards. In policy gradient methods, the policy is explicitly represented as a probability distribution over possible actions for a given state. These methods directly optimize the policy by adjusting its parameters to maximize the expected cumulative reward. Because policy gradient methods inherently produce stochastic policies, exploration is naturally integrated into the learning process. However, they can suffer from high variance in gradient estimates, potentially leading to slower and less stable convergence.

In contrast, value-based methods derive the policy indirectly from the value function. Techniques like Q-learning focus on learning an optimal value function that estimates the expected return for each action in each state. Since the policy is derived from this value function, value-based methods often require explicit exploration strategies to ensure sufficient exploration. When enhanced with techniques such as experience replay and target networks—especially in deep Q-learning (DQN)—value-based methods can achieve more stable and reliable learning.

Actor-critic approaches differ from value-based and policy-based methods by integrating elements of both into a cohesive framework. In policy-based methods like REINFORCE, the policy is directly parameterized as a probability distribution over actions for a given state. These methods focus on directly optimizing the policy by calculating the gradient of the expected return with respect to the policy parameters. In contrast, value-based methods like Q-learning derive the policy indirectly from a value function, which estimates the expected return for each action in each state.

Actor-critic methods combine the strengths of both approaches. The actor is responsible for learning the policy and selecting actions, while the critic evaluates these actions by estimating the value function. The actor uses the policy gradient to improve its decisions, guided by feedback from the critic. In terms of optimization, actor-critic methods simultaneously optimize both the policy (actor) and the value function (critic). This dual optimization balances exploration and exploitation: the actor's inherent stochastic policy drives exploration, while the critic’s value function stabilizes and guides this process, reducing variance and improving learning stability. By combining the strengths of both policy-based and value-based methods, actor-critic approaches effectively bridge the gap between the two, providing a balanced and robust framework for solving complex reinforcement learning problems. This results in more efficient and stable learning, making actor-critic methods a versatile choice in a wide range of applications.

References:

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.).

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... & Kavukcuoglu, K. (2016). *Asynchronous methods for deep reinforcement learning.* In International conference on machine learning (pp. 1928-1937).

Konda, V. R., & Tsitsiklis, J. N. (2000). *Actor-critic algorithms.* Advances in Neural Information Processing Systems, 12, 1008-1014.

Williams, R. J. (1992). *Simple statistical gradient-following algorithms for connectionist reinforcement learning.* Machine Learning, 8(3-4), 229-256.