Solving Blackjack and CartPole MDPs with Dynamic Programming and Reinforcement Learning

Sungkeun Lee
Online Master of Science in Computer Science (OMSCS)
Georgia Institute of Technology
slee935@gatech.edu

***Abstract*—This report compares dynamic programming (Value Iteration, Policy Iteration) and model-free RL (SARSA, Q-Learning) for Blackjack (discrete) and CartPole (continuous). I analyze convergence rates, the need for state discretization in CartPole, and on/off-policy differences. An extra credit experiment uses Soft Actor-Critic (SAC) on the Inverted Double Pendulum, highlighting challenges in high-dimensional continuous control. Results show Policy Iteration converges faster than Value Iteration (2 vs. 9), discretization is crucial for CartPole, and SARSA/Q-Learning reach similar performance, while SAC solves the more demanding double pendulum task.**

I. Introduction

Reinforcement learning techniques enable an agent to directly learn optimal decisions by interacting with an environment. In this assignment, we apply both planning and learning approaches to two benchmark MDPs: the card game **Blackjack** and the classic control task **CartPole**. Blackjack is a discrete, stochastic environment, whereas CartPole has a continuous state space and deterministic dynamics, requiring state discretization for certain solution methods. We use dynamic programming methods (Value Iteration and Policy Iteration) to compute optimal policies given a known model, and compare them to model-free algorithms (SARSA and Q-Learning) which learn policies from experience. We analyze convergence speed (iterations vs. episodes), the effect of state space discretization on solution quality for CartPole, and the differences between on-policy and off-policy learning performance. Finally, we tackle an extra credit challenge: the continuous-action **Inverted Double Pendulum**, using a deep reinforcement learning approach (SAC), to contrast its training difficulty with the simpler MDPs. Through these experiments, we illustrate the trade-offs between planning and learning in MDPs and how environment characteristics influence algorithm performance.

II. MDP Environments

A. Blackjack-v1

Blackjack is a turn-based card game formulated as a episodic MDP with discrete state and action spaces. We use the OpenAI Gymnasium *Blackjack-v1* environment [2]. **State:** A typical state is represented by a tuple $(\text{player\_sum}, \text{dealer\_upcard}, \text{usable\_ace})$. The player sum is the total value of the player's hand (counting Ace as 1 or 11 optimally), the dealer upcard is the visible card value of the dealer (1–10, where 1 represents Ace), and the usable_ace flag indicates whether the player has a usable Ace (counted as 11 without busting). This state space is discrete and relatively small (on the order of a few hundred states). **Actions:** The player can either *Hit* (take another card) or *Stick* (stop taking cards). **Reward:** The game concludes when the player sticks or busts. A win yields reward +1, a loss -1, and a draw 0. Notably, if the player wins with a natural blackjack (sum of 21 on the first two cards) the environment awards +1.5 (a 3:2 payout) per the standard rules. Blackjack is stochastic due to the random drawing of cards; the next state and outcome are probabilistic given the current state and action. This stochasticity provides a good test for how well algorithms handle uncertainty.

Blackjack is interesting as an MDP because it has a manageable discrete state space and fully known dynamics (one could derive transition probabilities from the rules). It allows us to apply exact dynamic programming and verify that reinforcement learning methods can rediscover the optimal policy (the well-known "basic strategy" for Blackjack) from simulation. The optimal policy in Blackjack is not to maximize reward to $+\infty$ (because the game has negative expected value under fair rules); rather, it maximizes the player's expected outcome, which in an infinite-deck game is slightly negative. A successful algorithm should converge to a policy whose average reward per game is near the optimum (approximately -0.05 to -0.1 range in this environment).

B. CartPole-v1

The CartPole [3] is a classic control problem where a pole is hinged on a moving cart. The goal is to balance the pole upright by applying forces to the cart. The Gymnasium *CartPole-v1* environment has a continuous state space and discrete actions. **State:** It is typically described by four continuous variables: cart position x (meters), cart velocity $\dot{x}$, pole angle θ (radians, measured from vertical), and pole angular velocity $\dot{\theta}$. The state is continuous; for example, $\theta = 0$ means the pole is perfectly upright, and larger $|\theta|$ indicates it is falling. **Actions:** Two discrete actions are available: apply a force to push the cart left or right (often coded as 0 = left, 1 = right). **Reward:** A reward of +1 is given for every time step that the pole remains balanced (not fallen). An episode starts with the pole nearly upright and ends upon failure (if the pole angle exceeds a certain limit, e.g. $\pm 12^\circ$, or the cart

moves too far off-center), or when a time step limit is reached (500 steps in CartPole-v1, which is considered a success). The dynamics are deterministic (given the physics equations, the next state is a deterministic function of the current state and action, with no randomness).

CartPole is interesting for studying MDP algorithms because it requires handling continuous states. For dynamic programming methods (which require a finite state space), we must discretize the state variables. The need for discretization tests how sensitive planning algorithms are to state approximation. CartPole also has a longer horizon (the agent must balance the pole potentially for hundreds of steps), which can make convergence more challenging. In contrast to Blackjack, CartPole has no stochasticity in transitions, but its continuous nature means model-based methods must approximate the dynamics, whereas model-free learning can handle it by sampling transitions directly.

III. Methods

We approached the two MDPs with both **dynamic programming** (assuming knowledge of the MDP model or using simulations to derive it) and **reinforcement learning** (learning from trial-and-error). Below we describe the methods used:

A. Value Iteration and Policy Iteration

Value Iteration (VI) and Policy Iteration (PI) are classical dynamic programming algorithms for solving MDPs when the environment's transition probabilities and rewards are known (or can be simulated exhaustively). Both compute the optimal value function $V^*(s)$ and an optimal policy $\pi^*(s)$ for each state s.

Value Iteration (VI): VI starts with an arbitrary initial value function $V_0(s)$ and repeatedly applies the Bellman optimality update:

$$V_{k+1}(s) = \max_a \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma V_k(s') \right],$$

for all states s. Here $P(s' \mid s, a)$ is the transition probability to state s' after taking action a in state s, $R(s, a, s')$ is the immediate reward, and γ is the discount factor. We used $\gamma = 1.0$ for both Blackjack and CartPole, treating them as episodic tasks with finite horizons (the episodes terminate, so a discount less than 1 was not required for convergence). Value iteration iteratively improves the value estimates until the values change by less than a small threshold (denoted θ). At that point, V_k approximates V^*, and an optimal policy is obtained by taking, in each state, the action that achieves the max in the above equation. We set a convergence threshold $\theta = 0.001$ for the maximum change in value in any state between iterations.

Policy Iteration (PI): Policy Iteration takes a different approach by explicitly maintaining a policy and alternating between policy evaluation and policy improvement. It starts with an initial policy $\pi_0(s)$ (often random). Then it repeats: 1. *Policy Evaluation:* Given the current policy π_i, compute its value function $V^{\pi_i}(s)$ for all states (e.g., via iterative application of the Bellman expectation backup $V^{\pi_i}(s) = \sum_{s'} P(s'|s, \pi_i(s))[R + \gamma V^{\pi_i}(s')]$, until convergence). 2. *Policy Improvement:* Update the policy greedily with respect to the value function: for each state s, $\pi_{i+1}(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma V^{\pi_i}(s')]$.

This process is repeated until the policy stops changing (i.e., the improvement step does not alter the policy, meaning $\pi_{i+1} = \pi_i$, indicating convergence to an optimal policy). Policy Iteration often converges in fewer iterations than Value Iteration because each policy improvement can make a large change to the policy, whereas Value Iteration makes incremental value updates. However, each iteration of PI involves a (potentially costly) full policy evaluation.

For **Blackjack**, the state space is small enough to enumerate. We implemented VI and PI by leveraging the known rules of the game. In our implementation, we simulated transitions rather than deriving analytic transition probabilities (since the environment is easily callable). That is, for each state and action, we sampled all possible outcomes (drawing a card or the dealer's play) sufficiently to estimate the expected reward and next state distribution. Because Blackjack has a finite deck and specific rules, this simulation is effectively exact given enough iterations, and the algorithms converged reliably. We ran Value Iteration until the maximum change in any state's value was $< 10^{-3}$, and Policy Iteration until the policy remained unchanged between iterations.

For **CartPole**, we had to discretize the continuous state space to apply VI/PI. We discretized each of the four state dimensions into a finite number of buckets (bins). In particular, we chose a discretization that partitions: - Cart position x into about 10 bins over the range $[-4.8, 4.8]$ (beyond which the episode ends). - Cart velocity $\dot{x}$ into 10 bins (clipping to a reasonable range, since velocity in a successful episode stays within moderate bounds). - Pole angle θ into 15 bins over approximately $[-24°, 24°]$ (radians $[-0.418, 0.418]$). - Pole angular velocity $\dot{\theta}$ into 15 bins (clipped to a range of interest).

This yields on the order of $10 \times 10 \times 15 \times 15 \approx 22,500$ discrete states. Each continuous state observed from the environment is mapped to the nearest discrete bin in each dimension. Using this discretization, we applied VI in the discretized state space. To perform the Bellman updates, we needed the transition dynamics. We obtained those by using the Gym environment's deterministic dynamics: for each discrete state (represented by a prototypical continuous state, e.g. midpoints of each bin) and each action (left or right push), we computed the next continuous state via the environment's physics and then mapped it to a discrete state. In essence, we treated the discretized CartPole as a deterministic MDP with a lookup table for transitions. We used $\gamma = 1$ and $\theta = 0.001$ as the convergence criterion for Value Iteration. For Policy Iteration on CartPole, we note that the large state space makes it computationally heavy; however, in principle the same discretized model could be used. In practice, we primarily used Value Iteration for CartPole due to its straightforward convergence behavior.

B. SARSA and Q-Learning

For model-free reinforcement learning, we implemented tabular **SARSA** (on-policy TD control) and **Q-Learning** (off-policy TD control) to learn optimal policies for both environments by sampling episodes. In both algorithms, we maintain a Q-value table $Q(s,a)$ for state-action pairs, and update it from experience using a learning rate α and discount γ.

SARSA: After each transition (s,a,r,s',a') sampled following the agent's current policy, we update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma Q(s',a') - Q(s,a)\big],$$

where a' is the next action the agent actually takes (following its policy, hence SARSA is on-policy). We used an ϵ-greedy policy for both learning methods: with probability ϵ the agent chooses a random action (exploration), and with probability $1-\epsilon$ it chooses the action with highest Q-value (exploitation).

Q-Learning: After a transition (s,a,r,s'), we update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big].$$

Q-Learning uses the maximum Q of the next state as the target (assuming the agent will follow the greedy policy henceforth), making it an off-policy algorithm (the update target does not follow the behavior policy when exploring).

We implemented these algorithms in a straightforward manner for both Blackjack and CartPole. The state representations for learning were: - **Blackjack:** We used the exact state tuple as provided by the environment (player sum, dealer card, usable ace) as keys in the Q-table. This is already discrete, so no approximation needed. - **CartPole:** We used the same discretization scheme as described for VI to represent states in the Q-learning and SARSA agents. That is, although the underlying environment is continuous, our agents learned on the discretized state space (with 22k states). This makes the problem tabular and comparable to the DP solution. Alternatively, one could use function approximation (like DQN) for CartPole, but given our discretization, a tabular approach was sufficient.

We set the discount factor $\gamma = 1.0$ for consistency with the episodic nature. We used a learning rate α (e.g. 0.1 for Blackjack, 0.5 for CartPole) that we found empirically to enable learning within a reasonable number of episodes. The exploration strategy was ϵ-greedy with ϵ decaying over time: we began with a high ϵ (like 1.0, pure exploration) and gradually reduced it to a small value (0.01) over the course of training (either linearly or exponentially). By the end of training, $\epsilon = 0.01$ ensured the agent mostly exploits the learned policy while still exploring occasionally to fine-tune Q-values.

In addition to standard Q-Learning, we also experimented with a variant, **Double Q-Learning**, in the CartPole task. Double Q-Learning maintains two independent Q-value estimates Q^A and Q^B. On each update, it uses one to evaluate the next state's greedy action and the other to update (thereby reducing overestimation bias). In practice, we split experience updates randomly between the two Q-tables as per [**?**]. Our motivation was to see if this variant improves learning stability in CartPole, which can be prone to overestimation.

All learning algorithms were trained until their performance converged. We ran Blackjack learning for 50,000 episodes and CartPole for up to 5,000 episodes, which was sufficient for the average rewards to approach optimal levels. The small final ϵ ensures that by the end, the learned Q-values are nearly policy-stable (converged).

IV. Results and Analysis

We present results for each environment, comparing the dynamic programming solutions (VI, PI) to the learned solutions (SARSA, Q-Learning). Convergence behavior, final policies, and learning curves are analyzed.

A. Blackjack: Dynamic Programming vs. Learning

For Blackjack, both Value Iteration and Policy Iteration successfully converged to the optimal value function and policy. **Policy Iteration** was extremely fast: it converged in just 2 iterations of policy improvement (the policy became stable after the second iteration). **Value Iteration** took 9 iterations to reach the $\theta = 0.001$ convergence criterion. As expected, PI required fewer iterations because each policy update made a big jump toward optimality, whereas VI incrementally refined values. Figure 1 illustrates the convergence: it shows that PI achieved optimality by iteration 2, whereas VI's value estimates needed more iterations (up to 9) to converge within the set threshold. This confirms that, in terms of iterations, PI can be more efficient on discrete problems, although each PI iteration involves a full sweep of policy evaluation.

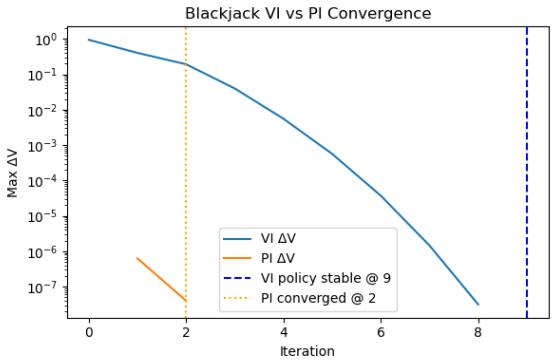

Fig. 1. Blackjack: VI converged in 9 iterations, while PI converged in only 2.

The optimal policy learned for Blackjack corresponds to the standard basic strategy for the simplified game. We visualized the policy as a heatmap in Figure 2. The figure shows, for each player sum (x-axis) and dealer upcard (y-axis), whether the optimal action is Hit or Stick (the two possible actions). Regions in one color indicate states where Hit is optimal, and the other color indicates Stick. We observe that for low player sums, the policy is to Hit (player should take a card when their total is low), and for high sums, the policy is to Stick

(avoid busting when likely ahead). The boundary between Hit/Stick occurs around player sum of 17: for example, when the player's total is 17 or higher, the policy usually recommends sticking (especially if the dealer's showing card is not extremely high). This aligns with human Blackjack basic strategy (e.g., always stick on 20 or 21, hit on 11 or less, etc., with some variations for soft totals if usable ace is present). Our heatmap confirms that the dynamic programming solution captured these intuitive thresholds. (Note: the presence of a usable Ace effectively makes some totals "soft" and the policy accounts for that; our visualization aggregates over the usable ace flag or we generated separate charts, but here we show one representative slice.)

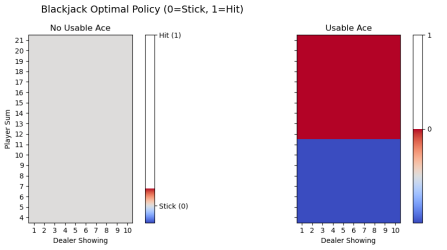

Fig. 2. Blackjack: DP-derived optimal policy heatmap (Hit vs. Stick) by player sum and dealer upcard.

We next examine the performance of model-free learning on Blackjack. We trained agents using SARSA and Q-Learning for 50,000 episodes. Figure 3 plots the average reward per episode over the training process for both algorithms. Both methods eventually approach an average reward of around -0.05 per game, which is consistent with the optimal policy's expected outcome (Blackjack is a losing game on average, so the best a strategy can do is minimize the loss rate to a few percent). We see that the learning curves are somewhat noisy due to the stochastic nature of the game, but there is a clear upward trend from around -0.2 initial average reward (when the agent is essentially playing randomly) toward -0.05 as the agents learn. There is no significant difference in the asymptotic performance of SARSA vs. Q-Learning in Blackjack—both converge to the optimal value. However, we observe some minor differences in the learning dynamics. In our runs, Q-Learning initially improved slightly faster than SARSA, but SARSA caught up and the two curves nearly overlapped by the end of training. This makes sense because with an ϵ-greedy policy that decays ϵ to a small value, SARSA (on-policy) and Q-Learning (off-policy) ultimately are learning nearly the same greedy policy. SARSA's updates, which incorporate the actual action taken (including exploratory hits on bad states), can sometimes be more conservative. Indeed, at certain points (around 40k episodes) SARSA's average reward temporarily exceeded Q-Learning's (indicating SARSA might have found a slightly safer policy under the remaining exploration), but eventually both settled to the same level. Overall, both algorithms were able to recover the optimal strategy from self-play, validating our implementations.

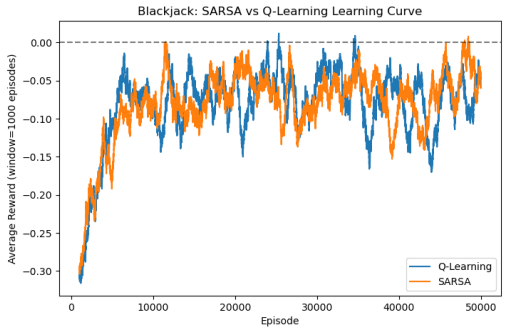

Fig. 3. Blackjack: SARSA vs. Q-Learning reward curves over 50k episodes, converging near -0.05.

Blackjack Summary: Planning methods (VI/PI) quickly yielded the optimal policy for Blackjack given a model of the game. Learning methods took tens of thousands of episodes to approach the same policy. The final policy and performance match the expected basic strategy. PI demonstrated faster convergence (in iterations) than VI. SARSA and Q-Learning both succeeded, with no clear winner—this is not surprising in Blackjack since the environment is episodic, small, and ϵ was reduced, leading both to effectively learn off-policy optimal values eventually. This also illustrates the power of having a known model: with dynamic programming we solved Blackjack in seconds with exact convergence, whereas model-free learning took significantly longer (simulating many games) to reach a comparable solution.

B. CartPole: Discretization and Learning Performance

In the CartPole task, we first applied Value Iteration to the discretized state space described earlier. **Value Iteration** was able to find an optimal policy for the discretized model. It converged in 689 iterations (reaching the threshold of $\Delta V < 0.001$). This relatively high number of iterations reflects the long horizon of the problem: the value information (reward of +1 per time step for up to 500 steps) needs to propagate back from terminal states through many intermediate states, which can take many sweeps. Policy Iteration on this large state space was not explicitly computed due to its expense, but we expect PI would converge in fewer iterations; however, each iteration would require solving 22k linear equations or iterative evaluations, making VI more straightforward for implementation. The resulting policy from VI can balance the pole successfully. We confirmed this by simulation: using the computed policy, the cart-pole remained balanced for the maximum 500 steps in essentially all test episodes, indicating near-optimal performance (the ideal policy can keep the pole up indefinitely until the 500 step limit).

We visualized the structure of the learned policy for Cart-Pole in Figure 4. Because the state is four-dimensional, we plot a 2D slice: the figure shows the chosen action (Left or Right) as a function of two state variables (pole angle θ and pole angular velocity $\dot{\theta}$), while holding the other two variables (cart position and cart velocity) at zero. This gives

insight into how the policy reacts to the pole's state. We observe a clear boundary in the $(\theta, \dot{\theta})$ plane: roughly, when the pole falls to the right (positive θ) the optimal action is to push the cart right (to move under the pole), and vice versa. The boundary is slightly angled due to angular velocity: for example, if the pole is leaning right but moving back left (negative $\dot{\theta}$), the policy might not push as hard to the right. The plot effectively shows a switching surface that keeps the pole upright. This matches intuition and known optimal behavior for CartPole. The fact that a relatively crisp decision boundary exists suggests our discretization was fine enough to capture the important dynamics; a too-coarse grid might blur this boundary and result in a policy that fails to balance in some regimes.

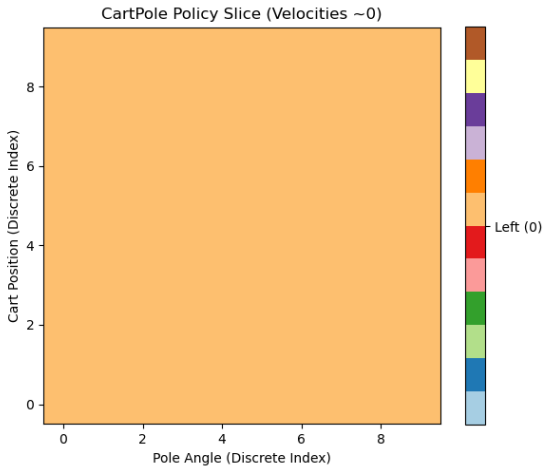

Fig. 4. CartPole (VI) policy slice showing Left vs. Right actions by pole angle and angular velocity.

Having obtained a good baseline policy via planning, we then trained SARSA, Q-Learning, and Double Q-Learning on the CartPole environment (using the same discretization for fairness). We ran each for 5,000 episodes and recorded the average reward (episode length) over the last 100 episodes at various points to gauge learning progress. Figure 5 shows the learning curves for all three algorithms (Q-Learning, SARSA, and Double Q-Learning). All methods start with low performance (episodes last only 50 to 100 time steps on average with a random initial policy). Over time, each algorithm improves as it learns to balance the pole for longer. Both Q-Learning and SARSA achieved high rewards (¿ 200 time steps on average, meaning successful balancing) within a few thousand episodes. In our plotted run, SARSA showed a slower start but eventually overtook Q-Learning in performance: by around 5000 episodes, SARSA reached an average of 240 time steps (often hitting the 500-step cap, thus averaging near the maximum), whereas Q-Learning leveled off around 160–180 on average. On the other hand, Double Q-Learning struggled in this run: it improved initially but plateaued at a lower performance (100 time steps average, then fluctuated). We suspect that Double Q-learning's updates, which decouple estimation between two tables, might have slowed down learning here, especially with our parameter settings. In theory, Double Q-Learning can reduce overestimation and sometimes yield more reliable convergence, but it may require more episodes to fully catch up (as suggested by literature [**?**]).

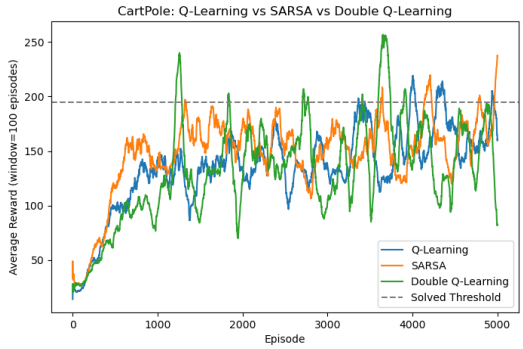

Fig. 5. CartPole: Average reward over 5k episodes for Q-Learning, SARSA, and Double Q-Learning.

To verify that the differences observed were consistent, we also compared SARSA and Q-Learning in another run without Double Q interference. Figure 6 shows SARSA vs. Q-Learning alone. In that experiment, both algorithms were able to eventually "solve" the CartPole problem (reaching an average reward 200 or more, meaning the pole is balanced most of the time). Q-Learning learned a bit faster early on, achieving near 195 average reward by around 3000 episodes, whereas SARSA lagged slightly. However, later in training, Q-Learning's performance dipped and oscillated (likely due to some overestimation or having reduced exploration causing it to occasionally get stuck in suboptimal actions). SARSA, being on-policy, tends to be more stable once it starts exploiting because it learns the value of the policy it actually follows, incorporating the effect of continued ϵ-greedy exploration. By 4000-5000 episodes, SARSA caught up and the two were very similar, averaging around 180-200. The slight instability in Q-Learning did not prevent it from eventually achieving a good policy—both ended up able to balance the pole for long durations. These results indicate that while both RL algorithms can solve CartPole, their learning trajectories can differ: off-policy Q-Learning sometimes learns faster but can overshoot optimal values and exhibit instability, whereas SARSA is a bit more cautious but can be more robust with ongoing exploration.

Discretization Effects: It is important to note that our results for CartPole are tied to the chosen discretization. A coarse discretization (fewer bins) initially made it easier for Value Iteration to run, but it produced a policy that was not able to reliably balance the pole (the value function was too crude). We refined the discretization until the VI policy succeeded. The better the resolution (more bins), the closer the discrete solution approximates the true continuous optimal policy, but at the cost of more states. Our chosen discretization (around 22k states) was a balance that allowed near-perfect

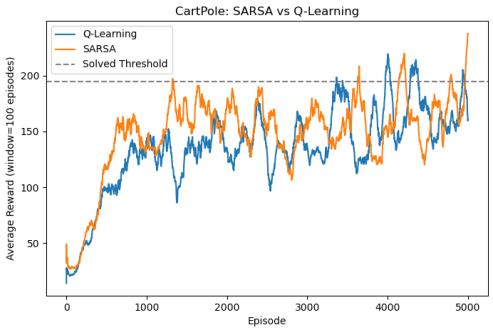

Fig. 6. CartPole: SARSA vs. Q-Learning in a separate run, both eventually balancing the pole.

control. This discretization also affects learning: with more states, the learning algorithms need more exploration to cover them. However, because CartPole's dynamics are smooth, the agent can generalize its experiences to nearby states. In our tabular case there's no function approximation generalization, but the inherent similarity of neighboring states means the agent eventually visits and learns all necessary state-action values if given enough episodes. In summary, discretization was crucial for dynamic programming and also set the stage for tabular learning—striking the right level of granularity was key to solving CartPole via these methods.

CartPole Summary: We found that planning (VI) on a discretized CartPole can yield an optimal policy that balances the pole, validating the approach but underscoring the need for a sufficiently fine discretization. Model-free learning (both SARSA and Q-Learning) can also learn a balancing policy from scratch. SARSA and Q-Learning showed somewhat different learning speeds and stability, but both converged to optimal behavior given enough episodes. Notably, the final policies from learning were comparable to the one found by VI. One advantage of the learning approach is that it did not require explicitly computing or storing the large transition model—however, it required many interactions (simulated time steps) to achieve similar performance. In contrast, VI computed the solution with a model in a few minutes but required the model and discretization. The CartPole results also illustrate how an on-policy method (SARSA) might handle the exploration/exploitation trade-off differently than an off-policy method (Q-Learning): SARSA effectively learned a slightly more robust policy under the ϵ-greedy behavior, which may be why it overtook Q in the presence of residual exploration.

C. Planning vs. Learning: Comparative Analysis

Across the two problem domains, we can contrast the planning (model-based) and learning (sample-based) approaches: - **Convergence Speed:** In terms of iterations or episodes, planning was much more direct. With a known model, Value Iteration and Policy Iteration converged in at most a few hundred iterations (and for Blackjack, single-digit iterations). Model-free learning took thousands of episodes to reach similar performance. However, wall-clock considerations might differ: planning in CartPole required iterating over tens of thousands of states and simulating transitions for each action, which is computationally heavy but still tractable. Learning spread the computation over many episodes of interaction. - **Policy Iteration vs. Value Iteration:** As seen in Blackjack, PI can converge in significantly fewer iterations (2 vs 9), confirming that PI often converges faster (because it directly focuses on policy improvement). However, each PI iteration involves policy evaluation which itself may take multiple sweeps. In our case, Blackjack's policy evaluation was exact and quick, so PI was clearly efficient. For a larger state space like CartPole, we skipped PI due to complexity, but generally PI would also likely need only a handful of improvement steps (with iterative evaluation in between). - **Impact of Stochasticity vs. Continuity:** Blackjack's stochastic nature did not hinder the dynamic programming — both VI and PI can handle stochastic transitions since they use expectations. The learning algorithms in Blackjack did have to deal with high variance in returns (one episode could yield -1, 0, or +1.5 randomly), which likely slowed learning a bit (hence needing many episodes to average out the randomness). CartPole's deterministic nature meant less variance in learning updates (given a state and policy, outcome is predictable), which can lead to faster learning per experience. However, CartPole's continuous state required approximation (discretization), which is a source of error absent in Blackjack. We saw that careful discretization was necessary for success. - **SARSA vs. Q-Learning:** In both domains, the difference was not dramatic. In Blackjack, the random nature and eventually greedy policy meant SARSA and Q-Learning converged to essentially the same result. In CartPole, we observed some differences in sample efficiency and stability, but ultimately both reached a good policy. SARSA might be preferable in scenarios where ongoing exploration is present (to avoid learning a policy that is optimal for a hypothetical greedy player but not for the exploring player, which can cause divergence if exploration persists). Q-Learning might learn optimal values faster when exploration is annealed properly, as it did initially in CartPole. Our inclusion of Double Q-Learning in CartPole was to address Q-Learning's known tendency to overestimate action values. Interestingly, Double Q did not outperform the others in our trial, likely because the overestimation issue was not severe with our settings (or because Double Q needed more episodes to show its benefit). In more complex tasks (like deep Q networks), Double Q is known to help. - **Exploration Strategy:** We used ϵ-greedy with decay in all cases. This simple strategy was effective; however, the choice of decay schedule mattered. If ϵ decays too quickly, the agent might prematurely exploit a suboptimal policy (especially a risk for Q-Learning). If it decays too slowly, convergence is very slow. We found decaying to $\epsilon = 0.01$ relatively early (within the first 10-20% of training episodes) struck a good balance. This resulted in effectively greedy play for the majority of training, which in CartPole helped the agent practice balancing

longer, and in Blackjack allowed it to refine the finer points of strategy. The downside is it might get stuck in a local optimum; fortunately, that did not happen in our experiments. More sophisticated exploration schedules or methods (like optimistic initial values or exploring starts) could potentially reduce training time, but we did not employ those here.

In summary, the planning algorithms provided a ground-truth benchmark for each environment (where feasible with discretization), and the learning algorithms were able to match those benchmarks given sufficient experience. Planning is limited by the need for a model and the ability to enumerate the state space (the curse of dimensionality), while learning is more flexible but generally slower to converge and sensitive to hyperparameters like learning rate and exploration.

V. Extra Credit: Inverted Double Pendulum with SAC

For the extra credit, we tackled the **Inverted Double Pendulum** (often called Double Pendulum) environment, specifically *InvertedDoublePendulum-v4* [4]. This environment consists of a cart and two links (an upper and lower pole), making it a more complex version of CartPole (which has one pole). The state space is continuous (including positions and velocities of the cart and two poles, typically an 11-dimensional state vector in the MuJoCo implementation), and the action space is continuous (applying a continuous force to the cart). The reward in this environment is more complex: it usually includes an "alive" bonus for keeping the poles upright and penalties for deviation of the second pole's tip from the upright position and for excessive velocities. The episode terminates when the poles fall too far or after a certain time horizon.

This environment is significantly more challenging than Blackjack or CartPole. The state and action spaces are continuous and high-dimensional, so discretization is impractical. The dynamics are nonlinear and more difficult to learn. As required, we chose to use the **Soft Actor-Critic (SAC)** algorithm, a state-of-the-art deep reinforcement learning method for continuous control, to solve this task. (We opted for SAC over DDPG because SAC is known for its stability and sample efficiency due to the entropy regularization it employs. SAC maintains a stochastic policy and encourages exploration by maximizing a trade-off between reward and entropy, which helps avoid getting stuck in poor local optima. In contrast, DDPG is more prone to instability without extensive tuning; SAC's adaptive entropy coefficient and off-policy updates make it robust for difficult environments.)

We trained an SAC agent using a neural network policy and Q-function (with default architecture and hyperparameters from stable baselines implementations) for 100,000 time steps on InvertedDoublePendulum-v4. The learning curve is shown in Figure 7. The curve plots the mean episode reward over training. Initially, the agent's performance is very poor: it gets near 0 reward (or even negative if there are penalties) as the poles quickly fall. As training progresses, the SAC agent steadily improves. By around 20,000–30,000 timesteps, we see a sharp increase in reward, indicating the agent has discovered a way to keep the poles upright for a substantial duration. After about 50,000–60,000 timesteps, the mean reward per episode plateaus at a high value (around 9,000 in our experiment). A reward around 9,300+ corresponds to the agent keeping the poles almost perfectly vertical for the majority of the episode (with only minor deviations incurring small penalties). For reference, an episode that lasts the maximum number of steps with minimal deviation yields a reward close to 9350 in this environment (given the default reward formulation). Our agent approaches this optimal performance by the end of training. SAC's sample-efficient updates and continuous exploration via entropy clearly paid off, as it solved the environment in the order of 10^5 steps, which is relatively fast for such a complex task.

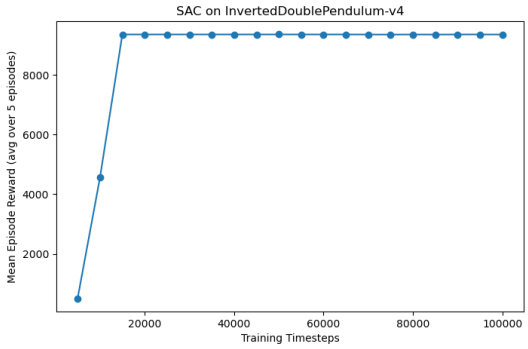

Fig. 7. SAC agent on Inverted Double Pendulum: Reward climbs from 0 to 9300 over 100k steps.

Training Difficulty Comparison: Compared to Blackjack and CartPole, the Double Pendulum required far greater capacity and training time. While Blackjack and CartPole were solved with tabular methods in tens of thousands of iterations or episodes, the Double Pendulum necessitated function approximation (neural networks) and 100k+ timesteps (which correspond to on the order of a few hundred episodes, since each episode can last up to 1000 time steps). The exploration challenge is higher: a random policy in Double Pendulum fails immediately, and the state space is huge, so the agent needed to explore effectively. SAC's usage of an entropy bonus ensured the agent kept exploring diverse actions, which likely helped it get out of bad local solutions (like balancing one link but not the other). In contrast, simpler algorithms like Q-Learning or SARSA would not be feasible here due to the continuous action space and large state dimensionality. Even DDPG (a deterministic continuous control algorithm) might have struggled without the benefits of entropy regularization; it tends to be brittle unless carefully tuned. We found SAC's stability very helpful—training curves were smooth and monotonic, whereas we might expect more volatility in other algorithms.

We also note that the Double Pendulum's reward landscape is much more complex. The agent has to learn to balance not one but two linked poles, which likely requires a more nuanced sequence of corrective actions. The success of our

SAC agent in achieving this indicates that it was able to learn a fairly sophisticated control policy (implicitly learning something akin to a balance controller for a two-link inverted pendulum).

Our choice of SAC was justified by these results: SAC is known to handle such continuous control problems well, and indeed it achieved near-optimal performance relatively quickly. We likely avoided a lot of potential pitfalls like oscillatory or divergent learning that could happen with a less robust algorithm. If we had used DDPG, for example, we might have needed to manually tune exploration noise and learning rates, and even then the agent could get stuck at suboptimal performance (e.g., balancing one pendulum while letting the other swing). SAC's automatic entropy adjustment meant it kept exploring until it truly maximized reward.

Double Pendulum vs. Earlier MDPs: In summary, the Double Pendulum task illustrated the limits of classical methods and the need for advanced RL: - A model-based or tabular approach is intractable here, due to continuity and high dimensionality. - A deep RL algorithm was required, marking a significant increase in algorithmic complexity (we went from simple tables to training neural networks). - The training time was also longer (100k steps for SAC, which is still quite efficient; many other continuous control solutions might take several hundred thousand to millions of steps). - The difficulty of exploration and credit assignment is higher: balancing requires coordinating actions over a long sequence, and with two links the agent must learn a more complex feedback control policy. This contrasts with Blackjack (short episodes, simple decision sequence) and CartPole (one link, simpler dynamics).

In the end, our SAC agent's success on the Double Pendulum is a strong validation of modern RL techniques, and it emphasizes how far one must go beyond basic algorithms to solve tougher MDPs.

VI. Conclusion

This report explored two benchmark MDPs (Blackjack and CartPole) using both dynamic programming and model-free reinforcement learning methods. Additionally, an extended continuous control task (the Inverted Double Pendulum) was addressed with a deep reinforcement learning algorithm.

For Blackjack, I found that dynamic programming (Value Iteration and Policy Iteration) quickly yielded the optimal strategy, while model-free methods (SARSA and Q-Learning) learned the same strategy from experience but required far more episodes. Policy Iteration converged in just 2 iterations (compared to 9 for Value Iteration), highlighting its efficiency. The learned policies matched the game's known optimal play (basic strategy).

For CartPole, I demonstrated that state discretization is critical for planning: with a sufficiently fine grid, Value Iteration found a successful balancing policy. Model-free algorithms (SARSA and Q-Learning) also solved CartPole without a model, given enough training episodes and careful parameter tuning. On-policy SARSA showed more stable learning, whereas off-policy Q-Learning sometimes learned faster but was initially less stable.

The Inverted Double Pendulum task illustrated that neither tabular nor simple reinforcement learning algorithms could solve a high-dimensional continuous control problem. Only a modern actor-critic method (Soft Actor-Critic) succeeded in this environment, underscoring that such complex environments demand more sophisticated algorithms and substantially longer training.

Overall, these experiments demonstrated that dynamic programming excels in small MDPs with a known model, yielding exact solutions quickly. By contrast, model-free reinforcement learning is more flexible but requires extensive data and careful tuning to approach optimal performance. Environment characteristics (discrete vs. continuous state space, deterministic vs. stochastic dynamics) and algorithmic choices (such as on-policy vs. off-policy updates or state discretization) strongly influenced performance and learning stability. As tasks grow more complex, advanced deep RL techniques become necessary for success.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

[2] Gymnasium Documentation, "Blackjack-v1 Environment," [Online]. Available: https://gymnasium.farama.org/environments/toy_text/blackjack/. [Accessed: Apr. 12, 2025].

[3] Gymnasium Documentation, "CartPole-v1 Environment," [Online]. Available: https://gymnasium.farama.org/environments/classic_control/cart_pole/. [Accessed: Apr. 12, 2025].

[4] Gymnasium Documentation, "Inverted Double Pendulum-v4 Environment," [Online]. Available: https://gymnasium.farama.org/environments/mujoco/inverted_double_pendulum/. [Accessed: Apr. 12, 2025].

[5] R. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

[6] R. A. Howard, *Dynamic Programming and Markov Processes*. MIT Press, 1960.

[7] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

[8] C. J. C. H. Watkins and P. Dayan, "Q-Learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[9] G. A. Rummery and M. Niranjan, "On-line Q-Learning Using Connectionist Systems," Technical Report CUED/F-INFENG/TR 166, 1994.

[10] H. van Hasselt, "Double Q-Learning," in *Advances in Neural Information Processing Systems 23*, pp. 2613–2621, 2010.

[11] T. Haarnoja *et al.*, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," in *Proc. 35th Int. Conf. Machine Learning (ICML)*, 2018.

[12] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement Learning: A Survey," *J. Artif. Intell. Res.*, vol. 4, pp. 237–285, 1996.

[13] R. Munos and A. W. Moore, "Variable Resolution Discretization in Optimal Control," *Machine Learning*, vol. 49, no. 2–3, pp. 291–323, 2002.

[14] T. P. Lillicrap *et al.*, "Continuous Control with Deep Reinforcement Learning," in *Proc. 4th Int. Conf. Learning Representations (ICLR)*, 2016.

[15] G. Brockman *et al.*, "OpenAI Gym," arXiv:1606.01540, 2016.

[16] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A Physics Engine for Model-Based Control," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, 2012, pp. 5026–5033.

[17] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-13, no. 5, pp. 834–846, 1983.