The University of Edinburgh

School of Informatics

Cartpole Control with Dynamic Programming, Optimal Control, and Reinforcement Learning

---

Methods in Dynamic Programming, Optimal Control, and Reinforcement Learning

Authors: Varun Rayamajhi
Kamil Balinski

Abstract

We study cartpole stabilization through three complementary control approaches: dynamic programming (DP), model predictive control (MPC) with iterative linear quadratic regulation (iLQR), and tabular Q-learning. We first examine how discounting, dense running costs, and state-space resolution shape the value function and the estimated recovery boundary. We then analyze the linearized dynamics used by MPC, determine the planning horizons required for reliable control, and compare MPC and iLQR under disturbances. For Q-learning, we study the non-monotonic training behavior and the roles of learning-rate and exploration schedules. Finally, we compare the methods in terms of performance, computational efficiency, and robustness to changes in the pole length. Our results show that Q-learning achieves the best average return, MPC is more consistent and naturally responsive to online changes, and DP provides an interpretable solution whose accuracy is limited by state-space discretization.

Introduction

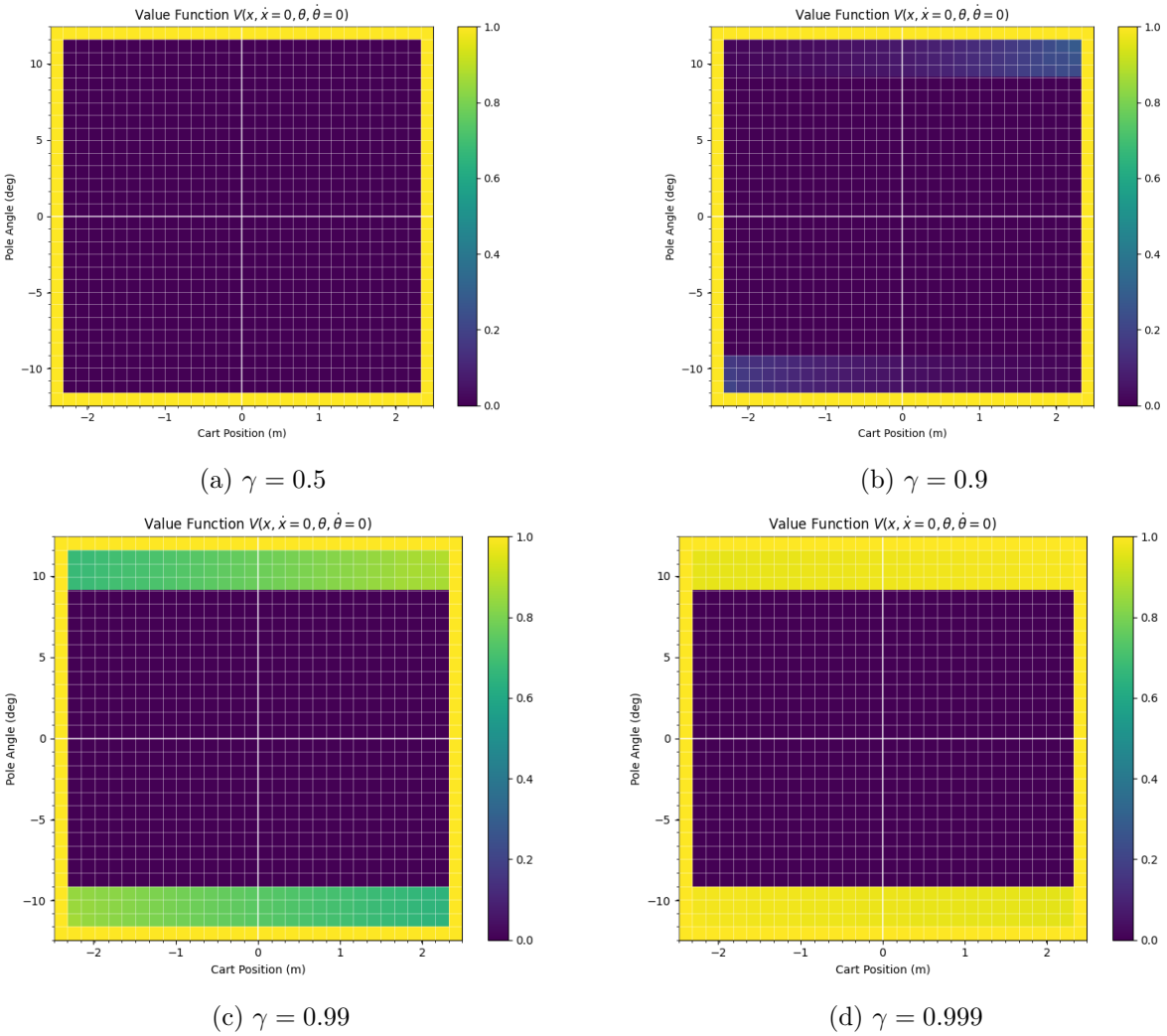
The cartpole system provides a compact setting for comparing model-based planning and model-free learning on the same underactuated control problem. Although each method attempts to keep the pole upright, it represents the problem differently. DP solves a discretized model globally, MPC repeatedly optimizes a finite-horizon trajectory in continuous state and action spaces, and Q-learning estimates action values from sampled interaction. These differences affect not only final performance, but also how each method uses computation, responds to disturbances, and generalizes when the physical system changes.

In this report, we develop the comparison progressively. We begin with the structure of the DP value function, move to finite-horizon optimal control and disturbance recovery, and then study the training behavior of a tabular Q-learning agent. We conclude by evaluating the three approaches under a common cartpole setting and by examining the effect of a pole-length perturbation as a simple sim-to-real gap.

Contents

1 Dynamic Programming

1.1 The Role of Gamma



(a) $\gamma = 0.5$

(b) $\gamma = 0.9$

(c) $\gamma = 0.99$

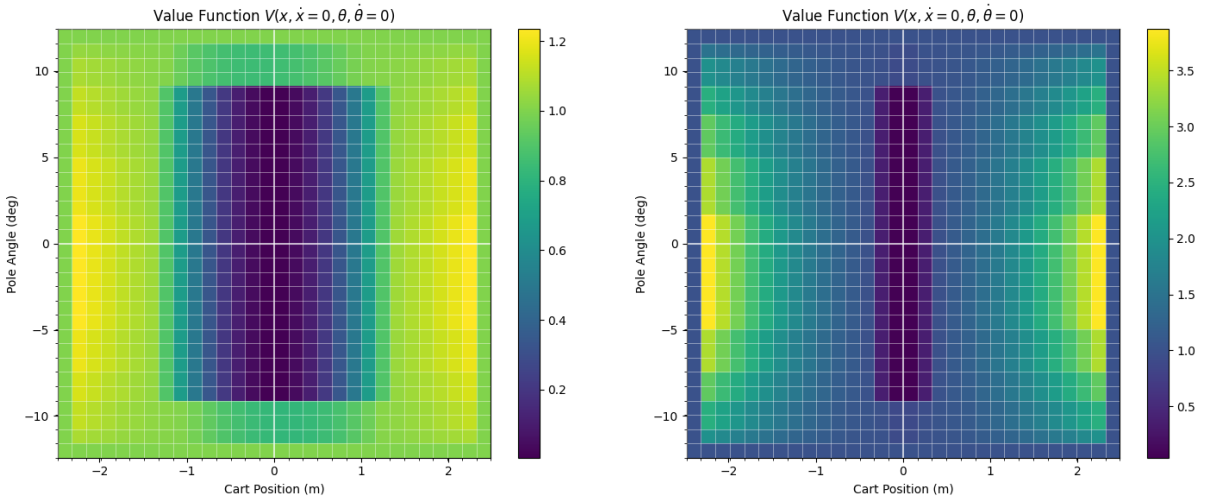
(d) $\gamma = 0.999$

Figure 1: Value function slices $V(x, \dot{x} = 0, \theta, \dot{\theta} = 0)$ for increasing discount factors $\gamma \in \{0.5, 0.9, 0.99, 0.999\}$ with $c_1 = c_2 = 0$.

As γ increases, the high-cost region near the termination boundaries spreads further into the state space and the low-cost (safe) region shrinks. With only a terminal cost, the value mainly measures the discounted time-to-failure. If a state s typically fails after $\tau(s)$ steps, then roughly $V(s) \approx \gamma^{\tau(s)} c_{\text{terminal}}$. Thus, a larger γ makes future failure matter more (longer effective planning horizon) and pushes high cost further inward.

The pattern is rectangular because termination is enforced separately as $|x| \leq x_{\max}$ and $|\theta| \leq \theta_{\max}$ with different limits. Thus, the boundary in the (x, θ) slice is an axis-aligned rectangle. In our plots, this rectangle is for the specific slice $\dot{x} = 0$ and $\dot{\theta} = 0$; the recoverable region would change for other velocities. The value changes more sharply along θ than along x because the pole angle is the unstable mode. Once θ grows, the system typically fails sooner, so $\tau(s)$ decreases faster when moving toward the θ limits than when moving toward the x limits.

1.2 Cost Function



(a) $c_1 = c_2 = 0.01$ (b) $c_1 = c_2 = 0.1$

Figure 2: Value function slices $V(x,\, \dot{x}=0,\, \theta,\, \dot{\theta}=0)$ under quadratic running costs.

Adding a quadratic running cost yields a smooth value function with a minimum near the equilibrium $(x, \theta) = (0, 0)$ and gradual growth as state deviations increase. Increasing c_1, c_2 from 0.01 to 0.1 steepens the value gradients and raises the overall magnitude, reflecting stronger penalization of cart displacement and pole angle. Concretely, the discounted objective becomes

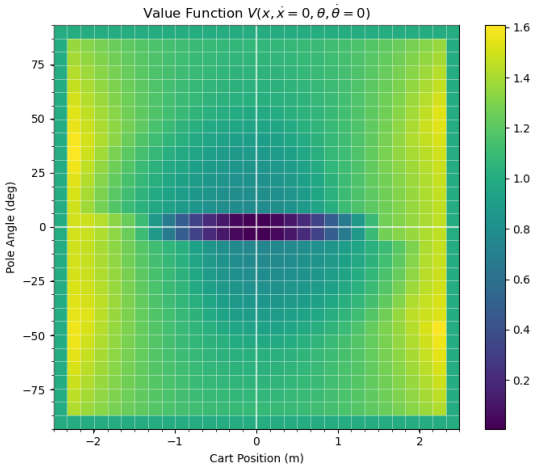
$$J \;=\; \sum_{t=0}^{T-1} \gamma^t \Big(c_1 x_t^2 + c_2 \theta_t^2 \Big) \;+\; \gamma^T c_{\text{terminal}}. \tag{1}$$

Thus, larger c_1, c_2 increases the contribution of the running cost relative to the terminal penalty.

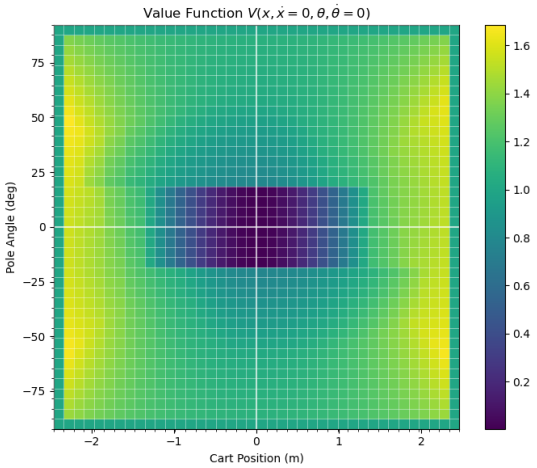
Between the two settings (Figure 2), $c_1 = c_2 = 0.01$ is superior for this task. We observe that it provides useful shaping while keeping a relatively large low-cost region inside the termination bounds. Thus, the value function still mainly reflects the objective of avoiding failure. On the other hand, with $c_1 = c_2 = 0.1$ setting, the low-cost region collapses into a narrow band near $x \approx 0$. This indicates that the running cost dominates and can discourage necessary cart motion that is sometimes required to stabilize the pole.

One advantage of dense running costs is that they provide informative feedback at every step (not only at termination), which typically improves learning speed and stability. One disadvantage is that they can bias the learned policy away from the true objective (survival), e.g., by encouraging overly conservative behaviour that prioritizes small state deviations over long-term balance.

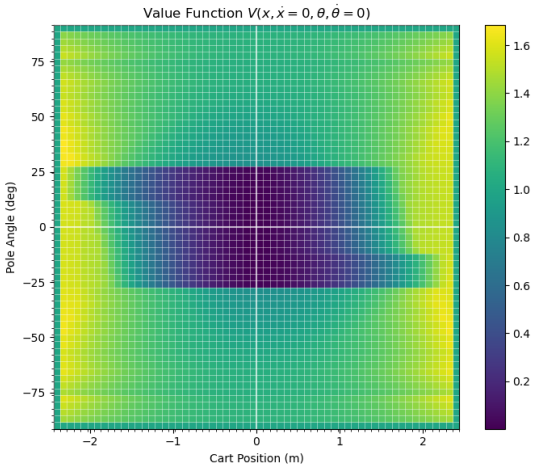
1.3 Termination Boundary



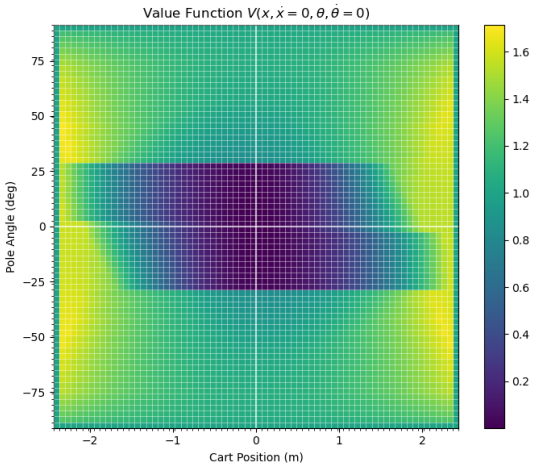
(a) $R = 30$



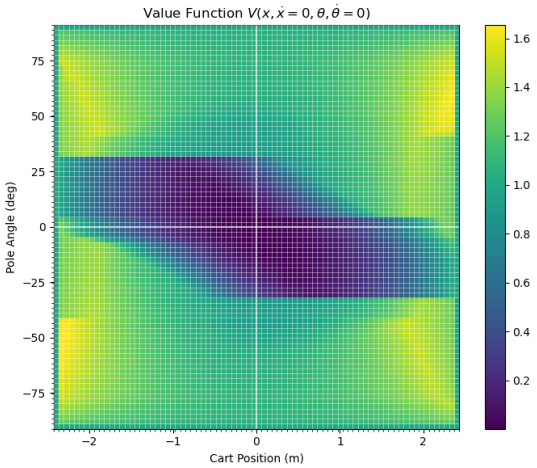
(b) $R = 40$



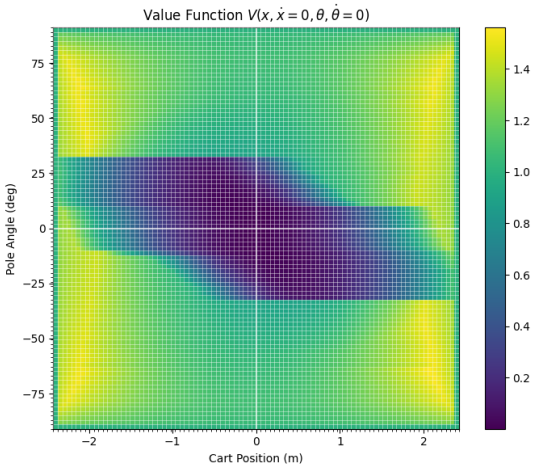
(c) $R = 60$



(d) $R = 70$



(e) $R = 80$



(f) $R = 90$

Figure 3: Value function slices showing refinement of the recovery boundary with increasing discretization R. Higher resolutions produce smoother value gradients and a more sharply defined recoverable region.

With $\theta_{\max} = 90°$, the value-function slices in Figure 3 show a clear boundary between low-value or dark region[1] near $(x, \theta) = (0, 0)$ and higher-value or brighter region[2]. Using small discretizations ($R = 40$-50), we get a conservative estimate of the recovery limit around $|\theta| \approx 21°$. As we refine the grid ($R = 60$-70), this boundary becomes sharper and the estimated recovery limit increases and stabilizes at approximately $|\theta| \approx 27°$. This indicates reduced discretization error. Our experiments at higher resolutions ($R = 80$ and $R = 90$)[3] suggest that the it is possible to recover from approximately $\theta = 32°(\pm 1°)$ and survive until the end of the episode. We note that this threshold is computed for the slice $\dot{x} = \dot{\theta} = 0$, and the recoverable region may differ for states with non-zero velocities.

[1] These regions correspond to the states that can be stabilised and survive until the end of the episode.

[2] These regions correspond to the states that typically fail before the horizon

[3] For $R > 90$, we were not able to experiment beyond $R = 90$ due to our computational capacity. We suspect that the exponential growth of the discretized state space resulted in memory limitations in our computers.

2 Model Predictive Control

2.1 Derivatives

Finite Difference Method

Advantages:

1. **Easy to implement.** Derivatives can be approximated numerically without the need to know the underlying (dynamics) function and hand-deriving analytical derivatives.
2. **Flexible.** Any changes to dynamics function (eg. adding terms) doesn't require much (or any) changes in re-computing derivatives.

Disdvantages:

1. **Computationally Intensive.** It requires multiple dynamics evaluations (e.g., $2n_x + 2n_u = 10$ extra dynamics evaluations per timestep per iLQR iteration in our case[4]).
2. **Numerical Instability.** The derivative accuracy depends on the step size ϵ. Too small ϵ amplifies floating-point noise and too large ϵ introduces bias. It can produce noisy Jacobians and degrade iLQR convergence.

Analytical Derivative Method

Advantages:

1. **Computationally efficient and stable.** We compute Jacobians directly without repeated dynamics function calls. It often gives smoother $\mathbf{A}$, $\mathbf{B}$ and more reliable iLQR convergence.
2. **Higher accuracy.** It avoids finite-difference approximation error and ϵ-tuning. Thus, the derivatives better reflect the true local linearization.

Disdvantages:

1. **Difficult to implement.** It requires deriving and correctly implementing all partial derivatives and is prone to algebraic or sign errors.
2. **Less flexible.** Any change to the dynamics model requires updating the derivatives as well. Handling non-smooth parts like saturation becomes piecewise and time-consuming.

The linearized discrete-time dynamics matrices are:

$$A = \frac{\partial f(\mathbf{x}, \mathbf{u})}{\partial \mathbf{x}} = \mathbf{I} + \Delta t \frac{\partial \dot{\mathbf{x}}}{\partial \mathbf{x}} = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & \frac{\partial \ddot{x}}{\partial \theta}\Delta t & \frac{\partial \ddot{x}}{\partial \dot{\theta}}\Delta t \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & \frac{\partial \ddot{\theta}}{\partial \theta}\Delta t & 1 + \frac{\partial \ddot{\theta}}{\partial \dot{\theta}}\Delta t \end{bmatrix}$$

$$B = \frac{\partial f(\mathbf{x}, \mathbf{u})}{\partial \mathbf{u}} = \mathbf{0} + \Delta t \frac{\partial \dot{\mathbf{x}}}{\partial u} = \begin{bmatrix} 0 \\ \frac{\partial \ddot{x}}{\partial u}\Delta t \\ 0 \\ \frac{\partial \ddot{\theta}}{\partial u}\Delta t \end{bmatrix}$$

Qualitative analysis

$\mathbf{A}$ reflects the kinematic relationships of the Cartpole system. We know that the position x_{t+1} depends directly on the velocity $\dot{x}_t$ and the angle θ_{t+1} depends directly on the angular velocity $\dot{\theta}_t$. Therefore, this kinematic relationship produces the Δt terms in the corresponding off-diagonal entries.

The partial-derivative terms involving $\ddot{x}$ and $\ddot{\theta}$ reflect the nonlinear coupling between the cart and pole dynamics. From the continuous-time equations of motion, we have $\ddot{x} = f_1(\theta, \dot{\theta}, u)$ and

[4] In our case, we have state dimension $n_x = 4$ and control input dimension $n_u = 1$.

$\ddot{\theta} = f_2(\theta, \dot{\theta}, u)$. It means that both accelerations depend on the pole angle θ and angular velocity $\dot{\theta}$. Consequently, we have $\frac{\partial \ddot{x}}{\partial \theta} \neq 0$ and $\frac{\partial \ddot{\theta}}{\partial \theta} \neq 0$. Physically, this means that changes in the pole angle or angular velocity influence the cart acceleration, and cart motion in turn affects the pole dynamics. These interactions appear in the linearized matrix $\mathbf{A}$ through the corresponding partial-derivative terms.

The system is underactuated since the control input u does not directly affect x or θ, but only their accelerations. Hence, $\mathbf{B}$ has nonzero entries only in the $\dot{x}$ (second) and $\dot{\theta}$ (fourth) rows.

2.2 Horizon Length

We ran MPC for $H \in \{1, 2, \cdots, 12\}$[5] and counted the number of terminations per 100 episodes. We observed 0 terminations for $H \geq 9$. Therefore, the minimum horizon required for no episode termination is $H = 9$.

To find the minimum horizon required for a mean return less than 0.01, we used a binary search over $H \in [1, 100]$. We first ran MPC with $H = 50$ and found the mean return was below 0.01, so we restricted our search to $H \in [1, 50)$. Continuing this process, we narrowed the candidates to $H \in \{28, 29, 30, 31\}$ and found that the mean return drops below 0.01 starting at $H = 30$[6]. Thus, the required $H_{\min}$ for final mean return of less than 0.01 is 30.

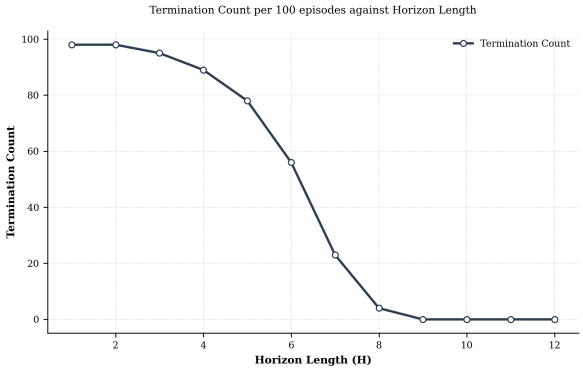


Figure 4: $H_{\min}$ required for no termination

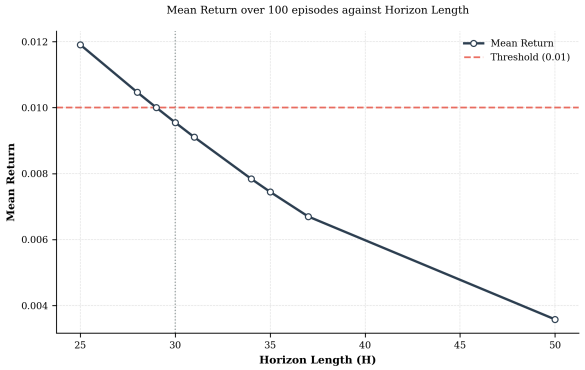


Figure 5: $H_{\min}$ required for mean return less than 0.01

With a very short horizon, MPC is too short-sighted. It optimizes only the next few steps, so it can't see that the pole will fall immediately or shortly after the horizon ends. As such, it prefers actions that reduce the immediate cost but do not build the cart motion needed to stabilize the pole. Likewise, recovery requires planning ahead, i.e. moving the cart now to affect the pole motion later. Therefore, by the time the pole angle approaches the termination limit, it is too late to correct the cart motion. Therefore, MPC with a very short horizon often leads to failure.

2.3 Disturbance Recovery

Small Tweak. The required tweak to get iLQR working is changing the value of the hyperparameter `max_iters` when initializing CartpoleMPC for iLQR in `mpc.py`. We increased it to 40 in our implementation, i.e. `ilqr = CartpoleMPC(H_ILQR, max_iters=40)`.

We increase `max_iters` because in closed-loop iLQR we replan at every timestep over the full horizon $H_{\text{iLQR}} = 100$ and then apply only the first control u_0 from that plan. iLQR is iterative:

[5]For sanity check, we ran MPC with $H \in [13, 20]$ and found the termination count to be 0.

[6]We set `seed=episode` so each episode has a different but reproducible initial state. This makes comparisons across different horizons consistent.

it linearizes $\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k)$, builds a local quadratic model, and updates via $\delta u_k = k_k + K_k\, \delta x_k$ with $k_k = -Q_{uu}^{-1} Q_u$ and $K_k = -Q_{uu}^{-1} Q_{ux}$. With a long horizon and too few iterations, we may not reach the stationary condition $Q_u \approx 0$. Therefore, u_0 is under-optimized and can destabilize the system. This is further amplified here because $\mathbf{A}_k, \mathbf{B}_k$ are computed by finite differences in `get_jacobians()`. This makes the local quadratic model noisier and typically requires more backward-forward pass iterations (i.e. a higher `max_iters`) to converge. **Robustness.** We define robustness as having a survival rate of 100%, where survival rate is defined as

$$\text{Survival Rate} = \left(1 - \frac{\text{Termination Count}}{\text{Number of episodes}}\right) \cdot 100\%. \tag{2}$$

For evalation[7], we've chosen the number of episodes to be 100. In the case of MPC, we found that the survival rate drops below 100% between the disturbance level 5 and 6 (Figure 6). In the case of iLQR, we found that the survival rate drops below 100% between the disturbance level 155 and 156 (Figure 7). Therefore, we conclude that MPC can handle disturbance level upto 5 and iLQR can handle disturbance level upto 155. We explain why iLQR better handles disturbance than MPC in our setting in Appendix 7.1.

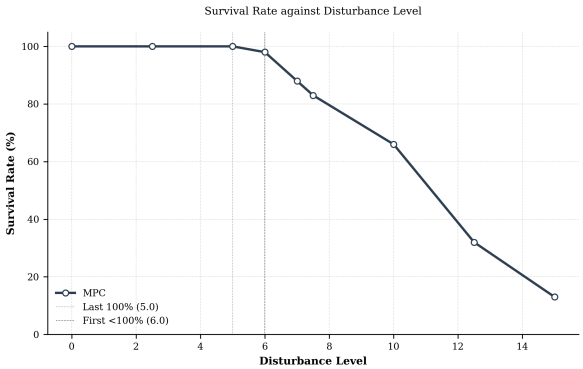


Figure 6: MPC: Survival Rate vs. Disturbance

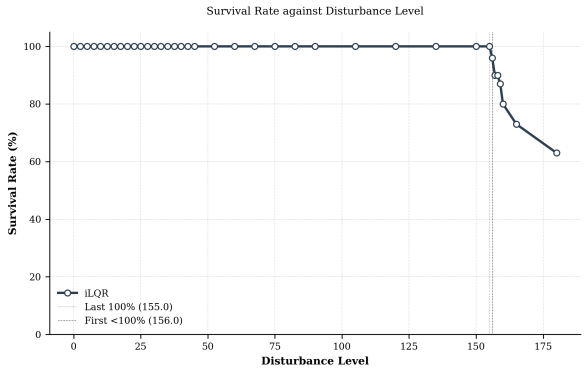


Figure 7: iLQR: Survival Rate vs. Disturbance

Caregiving scenario. The observed advantage of iLQR at different disturbance levels in our setup does not necessarily generalize because the cartpole spends most of its time near the upright equilibrium. In that region, the pole angle θ is small and thus the dynamics are close to linear (since $\sin(\theta) \approx \theta$ and $\cos(\theta) \approx 1$ for small θ). Hence, iLQR's local linear feedback works very well ($u_k = \overline{u}_k + k_k + K_k(x_k - \overline{x}_k)$) and can quickly correct small deviations.

However, for the caregiving scenario, the main disturbance is the human moving in an unpredictable way. In this setting, we'd prefer MPC because it re-plans online using the latest measured states of humans, and it can explicitly enforce safety constraints by accounting for uncertainty in the human's future states (motion). For example, MPC can enforce a minimum safe distance constraint to the human over the horizon.

On the other hand, standard iLQR optimizes around a nominal trajectory and uses local linearizations. Therefore, if a human moves in a way that is not captured by the nominal trajectory prediction, the planned trajectory can become unsafe unless additional constraint-handling or robustness is added [1]. For these reasons, MPC approaches are more commonly used in safety-critical robotic applications [2], and thus we'd prefer to use MPC in the caregiving scenario.

[7]We also fixed the random seed for both MPC and iLQR so that they are evaluated on the same initial conditions and disturbance realizations. This makes the comparison fair and the results reproducible.

3 Q-Learning

3.1 Mean Returns Pattern

In reinforcement learning, returns often don't increase monotonically like in supervised learning mainly because of (i) the exploration-exploitation tradeoff and (ii) Q-learning bootstrapping from estimates that are still changing.

With ϵ-greedy exploration, the action is chosen as

$$a_t = \begin{cases} \arg\max_a Q(s_t, a) & \text{with probability } 1 - \epsilon, \\ \text{a random action} & \text{with probability } \epsilon. \end{cases}$$

So even late in the training, the agent sometimes takes random actions with probability ϵ. This suggests that a single bad exploratory action can cause an early termination. It then shows up as a sudden drop (i.e. spike downward) in the moving average return.

In addition, Q-learning updates use a bootstrap target, i.e. they depend on other Q-values. However, in the early phases of training, many of those Q-values are inaccurate. In our case, we discretize the 4D state into 30 bins per dimension, so the table is very large and many bins are rarely visited. This means that some Q-values remain poorly estimated for a long time. As a consequence, if the agent lands in one of these rarely visited bins, it may take a suboptimal action and the return can change sharply.

In Cartpole, a physical example is: the agent has mostly learned to keep the pole near upright, but when the pole is close to the angle limit, an exploratory action pushes the cart in the wrong direction. This increases the pole's angular velocity, the pole crosses the termination threshold (fell/out of bounds), and the episode ends early. Since return is strongly affected by hitting a termination, this creates a sharp drop in the moving average return.

3.2 Learning Rate

Finding a good learning rate. We define a good learning rate α as one that achieves a high final mean return while maintaining stable learning. From Figures 8 and 9, we observed that mid-range values produced the best performance: large values (e.g., $\alpha = 0.5$) learned quickly but showed oscillations and worse final plateaus due to overly aggressive TD updates. We therefore selected $\alpha = 0.05$ as the best trade-off between learning speed and stability (trained for 10M timesteps).

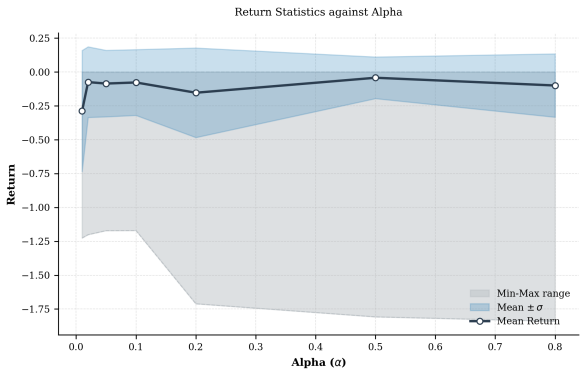


Figure 8: Statistics for different α

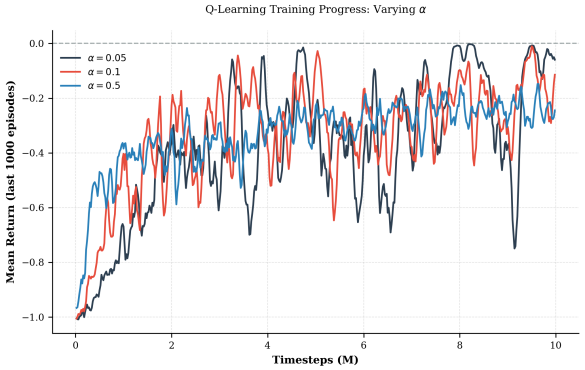


Figure 9: Training with different α (not all curves are plotted for visual clarity)

Choosing an α-schedule. For the α-scheduler, we used a decaying learning-rate schedule motivated by [3]. We use a state-action visit-count polynomial decay so that updates are larger

for rarely visited $(s,\, a)$ pairs and smaller for frequently visited pairs. This helps the Q-values stabilize and thus important for our large discretized state space. Our α-schedule[8] is defined as:

$$\alpha_t(s,a) = \max\left(\alpha_{\min},\ \frac{c}{\big(c + (1-\gamma)\, N_t(s,a)\big)^{\omega}}\right). \tag{3}$$

$N_t(s,a)$ is the number of times the state-action pair (s,a) has been updated, c and ω are hyperparameters, and γ is the discount-factor.[9]

Effect of α-Schedule. The learning curve with and without α-scheduler is provided in Figure 10. We define convergence as the point when the mean return reaches a stable plateau and shows no upward trend (allowing for occasional spikes due to exploration). From Figure 10, the α-scheduler reaches the plateau of 0.0 at around 1M timesteps[10]. Without scheduling, learning remains noisy and fails to reach a stable near-zero plateau even after 10M timesteps.

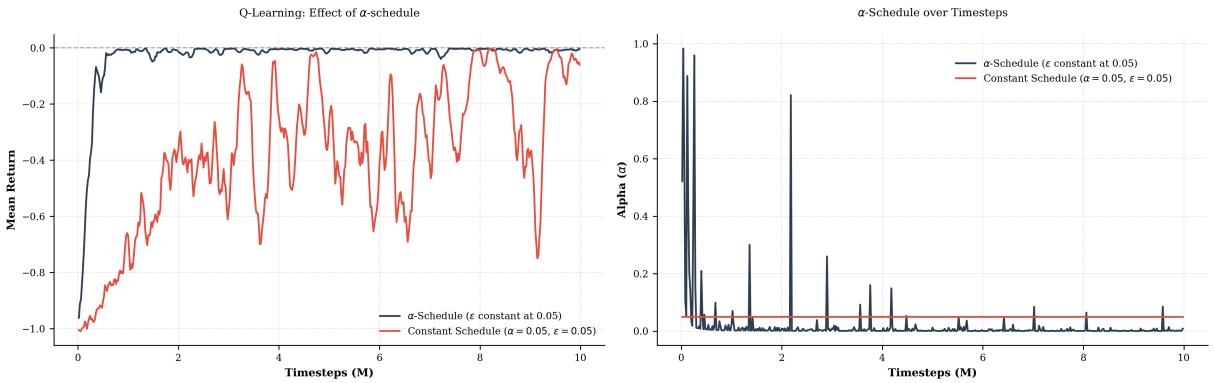


Figure 10: Q-learning: Learning with and without α-Schedule

The improvement occurs because decreasing α reduces the variance of TD updates once state-action values are well estimated. This also prevents late-stage oscillations in Q-values. Thus, the scheduler improves both stability and final mean return. We provide the evaluation results over 1000 episodes with and without the scheduler(s) in Appendix 7.3.

3.3 Exploration-Exploitation

Choosing an ϵ-decay schedule. We implemented a decaying ϵ-greedy schedule that gradually reduces exploration during training. This allows for broader state-action coverage early and more greedy behavior later. With motivation from [4], we used the following inverse-decay schedule:

$$\epsilon(t) = \max\left(\epsilon_{\min},\ \frac{\epsilon_0}{1 + k \cdot t}\right), \tag{4}$$

where k is chosen such that $\epsilon(T_{\text{total}}) = \epsilon_{\min}$[11]. The required k is

$$k = \frac{\dfrac{\epsilon_0}{\epsilon_{\min}} - 1}{T_{\text{total}}}.$$

[8] We also tried simpler global schedules (e.g., exponential decay and power-law decay in time), but Eq. (3) produced the cleanest plateau near 0.0.

[9] In our implementation, we use $\gamma = 0.99$ as provided, $c = 1.0$, $\omega = 0.85$, and $\alpha_{\min} = 10^{-4}$.

[10] Although there is a brief downward spike around 1.7M, the return quickly recovers back to the same plateau, so we still consider the method converged at 1M steps.

[11] In our implementation, we used $\epsilon_0 = 0.05$, $\epsilon_{\min} = 0.01$, and $T_{\text{total}} = 10$M.

Comparing ϵ- and α-schedules. α-scheduler and ϵ-scheduler affect learning in different ways. The ϵ-scheduler affects which actions are sampled by controlling exploration. On the other hand, the α-scheduler changes how strongly Q-values are updated. As a result, ϵ-schedule mainly reduces variance caused by random actions. And α-schedule stabilizes value estimates by preventing large TD updates late in training. Comparing to using only the α-scheduler (Figure 10), we observe that introducing ϵ-schedule produced more stable learning and a smoother performance plateau later in the training. This occurs because high exploration early improves state-action coverage and reduced exploration later allows the agent to exploit learned Q-values. Consequently, ϵ-schedule mainly imroves stability and convergence behavior when used with α-schedule (Figure 12) rather than significantly changing final performance alone as shown in Figure 11.

We provide the evaluation results over 1000 episodes with and without the scheduler(s) in Appendix 7.3 as well as discuss the effect of using only the ϵ-scheduler in Appendix 7.2.

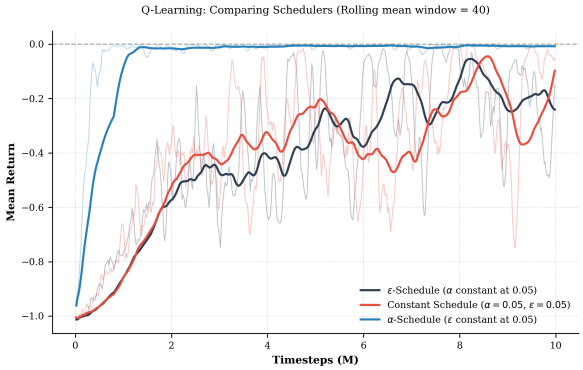


Figure 11: Comparing ϵ and α schedulers

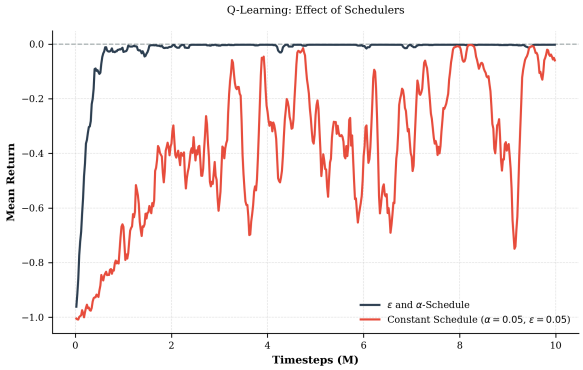


Figure 12: Effect of both schedulers

4 Method Comparison

4.1 Performance

Table 1: Method performance comparison ($\theta_{\max} = 25°$, $R = 40$). Cumulative episodic cost/reward reported to 4 decimal places.

Method	Mean	σ	Min	Max
DP	0.0160	0.0124	0.0014	0.0639
MPC	0.0060	0.0023	0.0033	0.0104
Q-Learning (return)	−0.0018	0.0028	−0.0495	−0.0002

Table 1 compares episodic performance under $\theta_{\max} = 25°$ and $R = 40$ (DP/MPC: cumulative cost; Q-learning: episodic return, where values closer to 0 indicate lower cost under our sign convention). On average, Q-learning performs the best: its mean return is −0.0018 (i.e., very close to zero), which corresponds to a low average cost. This is lower than MPC's mean cost of 0.0060. However, we see that MPC is slightly more consistent. It has a tighter spread and a better worst case (MPC max cost is 0.0104). On the other hand, Q-learning shows much worse episodes (min return of −0.0495 corresponds to a much larger cost). It suggests that there are occasional failures due to approximation/discretization and insufficient coverage of less-visited states. DP performs worst on average (0.0160 ± 0.0124), which is expected because it is limited by discretization error and solves on a fixed grid. Likewise, discretization and limited actions introduce sub-optimality for the underlying continuous dynamics. DP is globally optimal only for the discretized problem it solves on the grid. In our implementation, it computes the optimal value function for the grid states via the Bellman optimality update

$$V^*(s) = \min_{a \in \mathcal{A}} \Big(C(s,a) + \gamma\, V^*(s') \Big), \qquad s' = f(s,a).$$

Here, f is the grid-based transition obtained by simulating the continuous dynamics and then mapping the next state to the nearest grid cell (via `state_to_indices`), and $\mathcal{A}$ is the discrete action set. Thus, we conclude that DP is optimal on the grid up to the convergence tolerance. However, it is not guaranteed to be globally optimal for the underlying continuous Cartpole because the dynamics and terminal boundary are approximated by discretization and the action space is restricted. It is not a strictly "fair" comparison because MPC, DP, and Q-learning are (in general) solving different optimization problems. Therefore, changing cost weights changes that objective itself, not just the method. In DP, we minimize a discounted MDP cost of the form

$$J_{\mathrm{DP}}(s_0) = \sum_{t=0}^{T-1} \gamma^t\, C(s_t, a_t), \qquad C(s,a) = c_1 x^2 + c_2 \theta^2 + c_{\mathrm{terminal}},$$

over a discretized state grid and discrete actions $a \in \{0, 1\}$. In Q-learning, we instead maximize the reward function $R(s,a) = -C(s,a)$ (same weights as DP). Thus, the learned policy is optimal for that reward definition. On the other hand, MPC solves the following continuous finite-horizon optimal control problem:

$$J_{\mathrm{MPC}} = \Phi(\mathbf{x}_H) + \sum_{k=0}^{H-1} L(\mathbf{x}_k, \mathbf{u}_k) \quad \text{s.t. } \mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k), \qquad L(x,u) = q_1 x^2 + q_2 \dot{x}^2 + q_3 \theta^2 + q_4 \dot{\theta}^2 + r u^2,$$

with continuous actions $u \in [-1, 1]$. This means that MPX explicitly penalizes velocities and control input in a way that DP and Q-learning do not. Therefore, if MPC uses different weights than DP or Q-learning, we are comparing policies optimized for different objectives. Thus, the reported performance reflects both the method and the objective choice.

4.2 Computational Efficiency

MPC is fundamentally an online optimization method. At each time step, it solves a finite-horizon optimization problem using the current state, applies only the first action, and then re-plans at the next step. On the other hand, DP and Q-learning are primarily offline. They compute an approximate value function and policy for a fixed MDP objective ahead of time. For example, DP does so by dynamic programming over a discretized model and Q-learning does so by sampling experience to estimate Q. Then, the execution is simply a lookup table (or $\arg\max_a Q(s,a)$) with negligible compute per step. Consequently, MPC trades high per-timestep compute for better online reactivity and DP/Q-learning trade high training compute for low-cost deployment.

With resolution R per dimension in a 4D state grid, the number of states is $|S| = R^4$. A sequential value-iteration implementation would, per iteration, loop over all states and all actions, giving roughly

$$\text{sequential steps} \approx N_{\text{iter}} \times |S| \times N_{\text{actions}} = N_{\text{iter}} \times R^4 \times N_{\text{actions}}.$$

For $R = 40$, $R^4 = 40^4 = 2{,}560{,}000$. With $N_{\text{iter}} = 100$, this is approximately

$$100 \times 2.56 \times 10^6 \times N_{\text{actions}} \;\approx\; 2.56 \times 10^8 \times N_{\text{actions}}$$

sequential state-action backups (e.g., $\approx 5.12 \times 10^8$ if $N_{\text{actions}} = 2$).

For tabular Q-learning, each environment step performs one (or a small constant number of) Q-updates, so the sequential cost scales approximately with the total number of interaction steps:

$$\text{sequential steps} \approx N_{\text{episodes}} \times T,$$

where T is the episode length (capped by the time limit). Unlike DP, Q-learning does not require sweeping the full R^4 grid each iteration; however, it may require a very large number of samples to converge, especially when failures are rare or exploration is inefficient. In practice, DP has heavy offline compute that grows rapidly with R (curse of dimensionality), whereas Q-learning shifts cost to data collection and may need many more sequential steps to reach stable performance.

In our implementation, Q-learning was trained for 10^7 interaction steps (i.e., 10 million sequential updates), compared to approximately 5.12×10^8 sequential state-action backups required for DP with $R = 40$ and $N_{\text{iter}} = 100$.

GPU acceleration primarily benefits model-free reinforcement learning methods such as Q-learning and policy-gradient algorithms. These methods are typically limited by data collection rather than computation per update. Modern GPU-accelerated RL mostly reduces wall-clock training time by generating experience faster, rather than reducing how many environment steps are needed. If a method needs N environment steps to reach a target score, then

$$T_{\text{wall}} \approx \frac{N}{S},$$

where S is the number of environment steps generated per second. Running K environments in parallel can increase S by up to $\approx K$ until hardware limits dominate. GPU simulators can step very large numbers of environments at once, which greatly increases S and makes training much faster in real time.[5] Distributed model-free RL systems use many parallel actors to collect data quickly (high frames/steps per second). This speeds up learning in wall-clock time, even though the required number of steps N is not fundamentally changed. In contrast, DP benefits less from GPU parallelism since its main bottleneck is the exponential growth of the discretised state space rather than environment sampling.

4.3 Generalization

To examine generalization, we chose Q-learning and varied the pole length during evaluation. We trained the agent for 10M steps with both of our schedulers (Eq. (3), Eq. (4)) enabled and with pole length $L_0 = 0.5$m. We then evaluated the learned policy with length $L \in [0.5, 2.0]$ (200 samples).

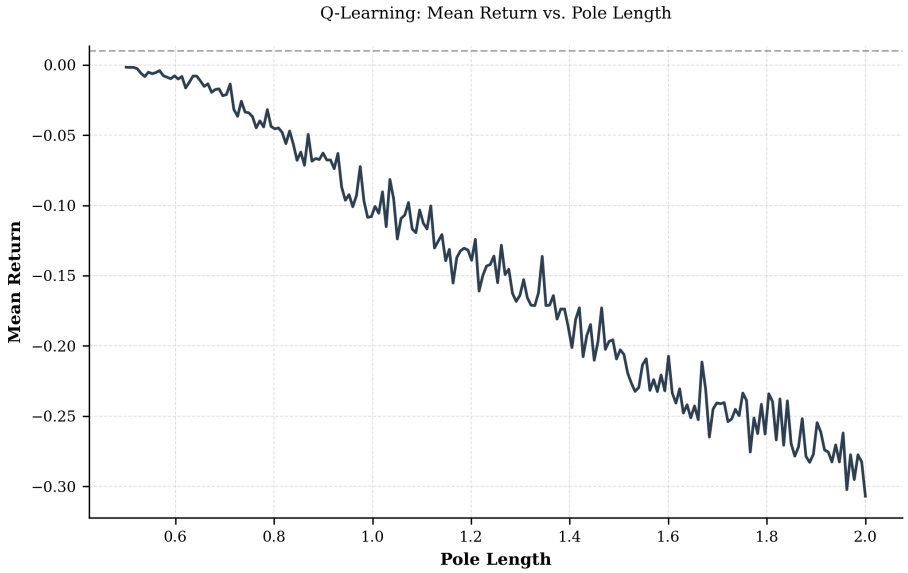


Figure 13: Q-learning: effects of changing pole length

We're not surprised with the results in Figure 13. We observe that the mean return over 1000 evaluation episodes degrades as L moves away from L_0. This happens because changing L changes the environment dynamics (i.e. the underlying MDP). The transition model depends on L, i.e. $s_{t+1} \sim P_L(\cdot \mid s_t, a_t)$. Therefore, the optimal action-value function satisfies a different Bellman optimality equation for each L, i.e.

$$Q_L^*(s,a) = \mathbb{E}_{s' \sim P_L}\left[r(s,a,s') + \gamma \max_{a'} Q_L^*(s',a') \right]. \tag{5}$$

Our learned policy is greedy with respect to the Q-table learned at L_0, i.e.

$$\pi_{L_0}(s) = \arg\max_a Q_{L_0}(s,a). \tag{6}$$

Thus, when we evaluate our agent at $L \neq L_0$, we are effectively using a policy optimized for P_{L_0} on a different transition model P_L. Therefore, performance drops under this sim-to-real gap. We also provide a Physics-inspired answer in Appendix 7.4.

Among DP, Q-learning, iLQR, and MPC, MPC is theoretically the best suited for online adaptation because it recomputes a control action online from the current measured state by repeatedly solving a finite-horizon optimal control problem. At each timestep, MPC solves

$$\mathbf{u}_t^* = \arg\min_{\mathbf{u}_{0:H-1}} \sum_{k=0}^{H-1} L(\mathbf{x}_k, \mathbf{u}_k) + \Phi(\mathbf{x}_H) \quad \text{s.t.} \quad \mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k). \tag{7}$$

It then applies only the first action $\mathbf{u}_t = \mathbf{u}_0^*$ and re-plans at the next timestep using the new measured state. This makes MPC naturally responsive to changes and disturbances compared to DP/Q-learning. Other methods produce fixed policies after training/evaluation, i.e.

$$\pi(s) = \arg\max_a Q(s,a) \quad \text{(Q-learning)}, \qquad \pi(s) \text{ from DP}. \tag{8}$$

iLQR provides local feedback around a nominal trajectory (via gains K_t). It helps in handling disturbance, but it is still a local, model-based method and does not directly adapt a learned value function online. Hence, in the sim-to-real context where the real environment might be different from the simulated environment, MPC's online re-planning makes it the most suitable method for adaptation.

5 Conclusion

We compared dynamic programming, model predictive control, iLQR, and tabular Q-learning on cartpole stabilization. DP made the roles of discounting, running cost, and discretization especially clear: increasing the grid resolution sharpened the estimated recovery boundary, but its R^4 state-space growth quickly became a computational limitation. MPC avoided this fixed grid and achieved reliable closed-loop control once the horizon was long enough, while iLQR's local feedback and longer planning horizon produced stronger disturbance recovery in our particular setup.

Q-learning achieved the best average return after sufficient training, and scheduling the learning rate was especially important for stable performance. However, it also produced a worse tail of occasional episodes and degraded when the pole length moved away from the training value. Overall, no single method dominated every criterion. DP offered interpretability and grid-level optimality, Q-learning provided inexpensive policy execution after a large offline training cost, and MPC offered the strongest basis for online adaptation because it continuously re-plans from the latest measured state.

References

[1] J. Chen, W. Zhan, and M. Tomizuka, "Constrained iterative lqr for on-road autonomous driving motion planning," in *2017 IEEE 20th International conference on intelligent transportation systems (ITSC)*. IEEE, 2017, pp. 1–7.

[2] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. Scokaert, "Constrained model predictive control: Stability and optimality," *Automatica*, vol. 36, no. 6, pp. 789–814, 2000.

[3] E. Even-Dar and Y. Mansour, "Learning rates for q-learning," *Journal of machine learning Research*, vol. 5, no. Dec, pp. 1–25, 2003.

[4] D. Perkins, O. J. Escobar, and L. Green, "Dqn performance with epsilon greedy policies and prioritized experience replay," *arXiv preprint arXiv:2511.03670*, 2025.

[5] V. Makoviychuk, L. Wawrzyniak, Y. Guo, M. Lu, K. Storey, M. Macklin, D. Hoeller, N. Rudin, A. Allshire, A. Handa *et al.*, "Isaac gym: High performance gpu-based physics simulation for robot learning," *arXiv preprint arXiv:2108.10470*, 2021.

[6] K. M. Lynch and F. C. Park, *Modern robotics*. Cambridge University Press, 2017.

Appendix

7.1 Why does iLQR better handle disturbance than MPC in our setting?

In this setup, the iLQR forward pass applies a feedback and feedforward policy $u_k = \overline{u}_k + k_k + K_k(x_k - \overline{x}_k)$, which defines a time-varying linear feedback controller. When a disturbance pushes the cart away from the nominal trajectory, the feedback term $K_k(x_k - \overline{x}_k)$ immediately corrects the deviation at the same timestep.

In contrast, our MPC implementation re-plans at every timestep (closed-loop execution). However, after solving iLQR internally, it applies only the first action of the optimized control sequence. Therefore, disturbance handling for MPC mainly occurs through re-optimization at the next timestep rather than through an explicit feedback law during execution.

Additionally, iLQR plans over the full episode horizon ($H_{\text{iLQR}} = 100$), while MPC uses s shorter horizon $H_{\text{MPC}} = 30$. The longer horizon together with the locally optimal feedback gains K_k makes iLQR appear more robust to disturbances in this particular cartpole setup.

7.2 Effect of ϵ-schedule with constant α

Compared to the constant ϵ, we expected the scheduler to reduce random actions in later stages of training and produce a smoother learning curve with a more stable plateau. This occurs because high exploration early improves state-action coverage and reduced exploration later allows the agent to exploit learned Q-values instead of introducing exploration noise. However, our empirical result show that our scheduler doesn't improve training performance much as shown in Figure 14.

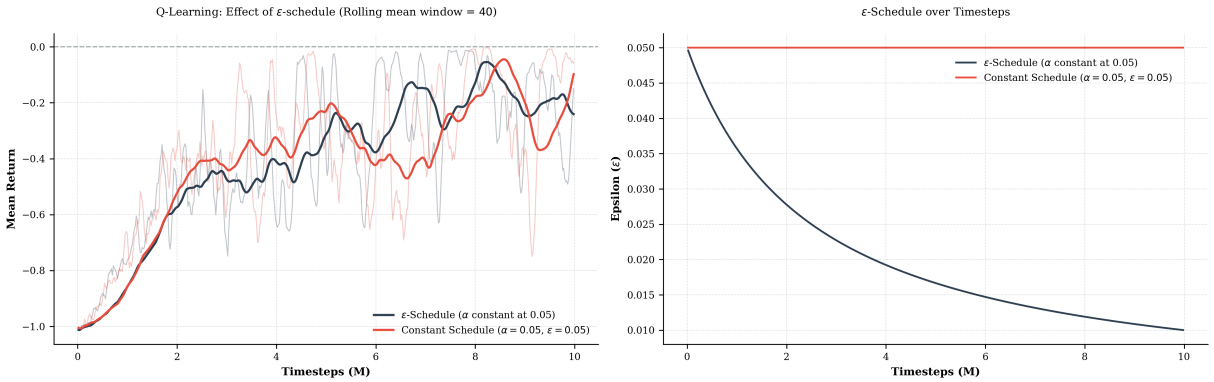


Figure 14: Q-learning: Learning with and without ϵ-Schedule

7.3 Evaluation of Q-learning agent with and without α- and ϵ- schedules

Using constant $\alpha = 0.05$ and constant $\epsilon = 0.05$, the final evaluation return over 1000 episodes was -0.0852 ± 0.2455. α-scheduler (with ϵ still constant at $\epsilon = 0.05$) significantly improved the final performance to a mean return of -0.00268 ± 0.00227 and produced a stable plateau close to 0.0. However, using only the ϵ-scheduler (with constant $\alpha = 0.05$) degraded performance (mean -0.385 ± 0.492). This suggests that in our discretized Q-learning setting, tuning α over time is more important than decaying ϵ alone. Finally, enabling both schedulers gave the best overall stability (mean -0.00236 ± 0.00193)[12].

[12]We observe that the improvement over the α-scheduler alone was small.

7.4 Sensitivity to Model Mismatch

From a physics perspective, changing L directly changes the cartpole accelerations. In our cartpole dynamics, the angular acceleration is inversely proportional to L:

$$\ddot{\theta} = \frac{g \sin\theta - \cos\theta \cdot \mathrm{D}}{L\left(\frac{4}{3} - \frac{m_p \cos^2\theta}{M}\right)}, \qquad \mathrm{D} = \frac{F + m_p L \dot{\theta}^2 \sin\theta}{M}. \tag{9}$$

Thus, when we increase L, we're changing how quickly the pole rotates and how the applied force F affects pole motion through D. As a result, the same action in the same discretized state leads to different next-state behavior when L changes. Consequently, the Q-values learned at L_0 no longer correspond to the true outcomes under $L \neq L_0$.