Comparison of Model Predictive Control and Deep Reinforcement Learning Control algorithms applied to the Inverted Pendulum Problem

Victor Vadakechirayath George
Embedded Systems Engineering
Albert-Ludwigs-Universität Freiburg
Freiburg, Germany

Sai Sourabh Tiruvaipati
Embedded Systems Engineering
Albert-Ludwigs-Universität Freiburg
Freiburg, Germany

***Abstract*—Balancing of an inverted pendulum has been a classical control task in both Control Systems and Reinforcement learning community. Current work is aimed at solving the control problem using approaches from different domains: Reinforcement learning techniques as well as Direct methods in Optimal Control. The nature of the algorithms and respective convergence, stability behaviours have been compared and explored in detail.**

***Index Terms*—Optimal Control, Reinforcement Learning, MPC, DDPG, Multiple Shooting, Actor-Critic, OpenAI Gym, RK4 integration.**

I. Introduction

A. Reinforcement Learning

Reinforcement Learning, hereon referred to as RL, refers to the learning method in which an agent which interacts with it's environment and takes actions based on response received to the actions it took, the response is quantified to a value termed as reward. Learning happens via trial and error and could be imagined as cause, effect relationship between actions and rewards [1]. Actions which resulted in favourable rewards are reinforced and are more likely to be taken again when similar situation recurs. From computational point of view, RL agent chooses actions which minimize accumulated costs or maximise collected rewards over it's long-term interaction with environment. A cost function which captures the performance criteria, is used to quantitatively evaluate actions of agent in the form of numerical reward.

B. Optimal Control

Model Predictive Control (MPC) refers to a class of control strategies that utilize an explicit process model to predict the future responses of a system. It is an advanced method of process control that is used to control a process while satisfying a set of constraints. The main goal of MPC is to find an optimal vector of control functions that minimize or maximize a performance index subject to a given process model as equality constraints, and boundary conditions as inequality constraints on the states and controls [9]. The problem to find the optimal control at each time-step can be solved using either Direct or Indirect Methods. In this approach, we are mainly focusing on Direct methods. In all direct methods, the control trajectory will be parameterized and the state trajectories computed using either sequential or simultaneous approaches. In the sequential approach, the state variables are considered as an implicit function of control trajectories, where the ODEs are addressed as an initial value problem using one of the dedicated integration methods like Runge-Kutta or Euler algorithms. In the simultaneous approach, state trajectories are parameterized, too, and we deal with all of parameterized variables (states and controls) as optimization variables in the NLP. The ODEs will be represented as equality constraints,either with collocation on finite elements [2] or with multiple shooting. [7].

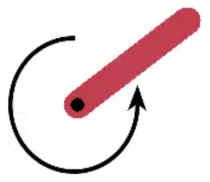


Fig. 1: Inverted pendulum environment in OpenAI Gym [3]

C. OpenAI Gym

OpenAI Gym is an open-source toolkit for developing and comparing reinforcement learning algorithms. It makes no assumptions about the structure of your agent, and is compatible with any numerical computation library. Gym comes with a diverse suite of environments that range from easy to difficult and involve many different kinds of data, with the Inverted

Pendulum problem being part of Classical Control suite. In the inverted pendulum environment in Gym, the pendulum starts in a random position, and the goal is to swing it up so it stays upright as shown in Figure 1.

D. Problem Statement

The goal of the problem is to swing up the pendulum to the position θ = 0 such that it is stationary at this position, implying that ω = 0, where θ, ω constitute the state space of the problem. They denote angular displacement and angular velocity respectively. All the dynamics are described in the Gym Pendulum environment along with specifications such as length and weight of the pendulum, acceleration due to gravity etc.
Input torque to the pendulum, denoted by τ is the control variable. Pendulum is free to swing around the pivot point. The problem is continuous in state and action space with defined boundaries.

II. Model and Problem Formulation

A. Reinforcement Learning Model

To deal with the continuous state and action space, an Actor-Critic(AC), model-free algorithm based on the deterministic policy gradient, with Deep-Q learning [6] involving multiple neural networks to solve the task. Typically, the AC architecture consists of two Neural Networks (NN) – an actor NN and a critic NN. The critic NN approximates the evaluation function, mapping states to an estimated measure of the value function, while the action NN approximates an optimal control law and generates actions or control signals.

1) Background: A standard reinforcement learning setup consisting of an agent interacting with an environment E in discrete timesteps is considered. At each timestep t, the agent receives an observation x_t, takes an action a_t and receives a scalar reward r_t.
An agent's behavior is defined by a policy π, which maps states to a probability distribution over the actions as shown,

$$\Pi : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A}). \tag{1}$$

The return from a state is defined as the sum of discounted future reward $R_t = \sum_{i=t}^{T} \gamma^{(i-t)} \cdot r(s_i, a_i)$ with a discounting factor $\gamma \; \epsilon \; (0, 1]$. Note that the return depends on the actions chosen, and therefore on the policy Π, and may be stochastic. The goal in reinforcement learning is to learn a policy which maximizes the expected return from the start distribution.
The action-value function describes the expected return after taking an action a_t in state s_t and thereafter following policy Π:

$$Q^{\Pi}(s_t, a_t) = E(R_t | s_t, a_t) \tag{2}$$

Using recursive Bellman Equation, we get,

$$Q^{\Pi}(s_t, a_t) = E[r_t(s_t, a_t) + \gamma \cdot E[Q^{\Pi}(s_{t+1}, a_{t+1})] \tag{3}$$

Function approximators are parameterized by θ^Q which can be optimized by minimizing the loss

$$L(\theta^Q) = E[(Q(s_t, a_t|\theta^Q) - y_t)^2] \tag{4}$$

where y_t is given by,

$$y_t = r(s_t, a_t) + \gamma \cdot Q(s_{t+1}, a_{t+1}|\theta^Q) \tag{5}$$

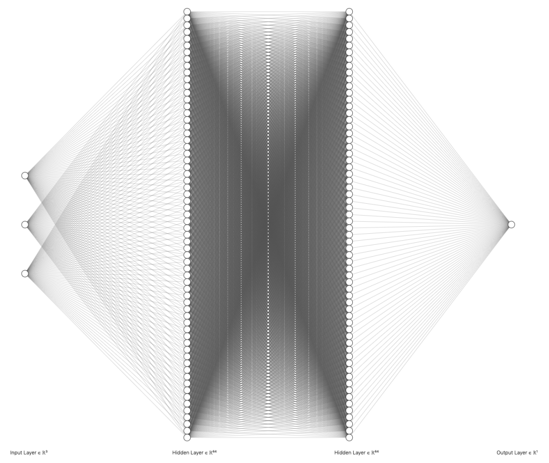


Fig. 2: actor network maps states to actions

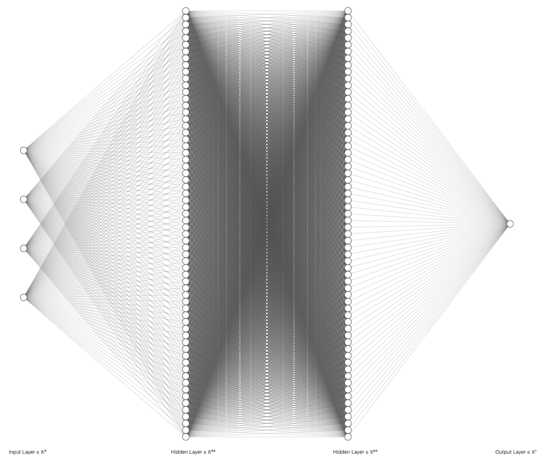


Fig. 3: critic network maps states, actions to corresponding action values

2) DDPG Algorithm: Deep Deterministic Policy Gradient algorithm is a model free, off policy actor critic algorithm using deep functional approximators that can learn policies in high-dimensional, continuous action spaces [5]. Naive application of actor-critic method with function approximators is unstable for challenging tasks. Stability issues are taken care by following workarounds:

- The network is trained off-policy with samples from a replay buffer to minimise correlations between samples.
- The network is trained with a target Q network to give consistent targets during temporal difference backups.

The full algorithm can be referenced in subsection VI-B.

3) DDPG Algorithm Implementation: DDPG algorithm maintains a parameterized actor function $\mu(s|\theta^{\mu})$ which specifies current policy by deterministically mapping states to a specific action. The critic $Q(s, a)$ is learned using Bellman Equations as in Q-learning.
Introducing non-linear function approximators implies that convergence is no longer guaranteed. One challenge when using neural networks for reinforcement learning is that most optimization algorithms assume that the samples are independently and identically distributed. Obviously, when the samples are generated from exploring sequentially in an environment this assumption no longer holds. We used a replay buffer to address these issues. The replay buffer is a finite sized cache $\mathcal{R}$. Transitions were sampled from the environment according to the exploration policy and the tuple (s_t, a_t, r_t, s_{t+1}) was stored in the replay buffer [5]. At each step actor and critic are updated by sampling a mini-batch uniformly from the buffer.
We create a copy of actor and critic networks, $\mu'(s|\theta^{\mu\prime})$ and $Q'(s, a|\theta^{Q\prime})$ respectively, that are used for calculating the target values. The weights of these target networks are then updated by having them slowly track the learned networks:

$$\theta^{'} \leftarrow \tau\theta + (1-\tau)\,\theta^{'} \qquad \text{where} \quad \tau << 1 \tag{6}$$

This means that the target values are constrained to change slowly, greatly improving the stability of learning. This simple change moves the relatively unstable problem of learning the action-value function closer to the case of supervised learning, a problem for which robust solutions exist [5]. Having both a target μ' and Q' was required to have stable targets y_i in order to consistently train the critic without divergence.This might slow down the learning, since the target network delays the propagation of value estimations.

4) Network Architecture: Including target networks, we are using total of four neural networks: two actor networks and two critic networks.
Actor network consist of three fully connected linear networks connected together with non-linearity provided by Hyperbolic Tangent (Tanh) activated layers in between. Input layer size is three with 64 nodes in hidden layers followed by single scalar output denoting the resulting action in Figure 2. Similarly, critic network has input size of four, accounting for both state and action inputs and outputs scalar Q-value shown in Figure 3. We are using an Adam Optimiser to update the weight parameters such that loss function is minimized. Loss function used is mean square errors since output values are continuous.
In each iteration of training, following steps are executed:

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
        $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
    Update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for
```

Fig. 4: Complete DDPG algorithm [5]

- From the replay buffer $\mathcal{R}$, mini-batch of tuple: (s_t, a_t, r_t, s_{t+1}) is sampled.
- Next actions a_{t+1} are updated as:
  $a_{t+1} \leftarrow actor\_target(s_{t+1})$ via actor target network.
- $Q(s_{t+1}, a_{t+1}) \leftarrow critic\_target(s_{t+1}, a_{t+1})$ via the critic target network.
- Set Q_target as
  $Q\_target \leftarrow r_t + \gamma \cdot Q(s_{t+1}, a_{t+1})$.
- Set $Q(s_t, a_t) \leftarrow critic(s_t, a_t)$.
- Critic loss is given as
  $critic\_loss \leftarrow MSE(Q\_target, Q(s_t, a_t))$
  where MSE denotes mean square error of Q_target and $Q(s_t, a_t)$ vectors.
- Actor loss is given as
  $actor\_loss \leftarrow -critic(s, a^{'}) \cdot mean()$ where
  $a^{'} = actor(s_t)$

B. Optimal Control Model

Model Predictive Control with a receding horizon (Receding Horizon Control) is an example of an online model-based approximate optimal control method which solve the optimal control problem over a finite time horizon at every state transition leading to a state feedback optimal control solution. We employ Receding Horizon Control algorithm using Simultaneous Direct Multiple Shooting to solve for the optimal control trajectory. We use multiple shooting for discretizing the dynamic system, so that the optimal control problem is transformed to a NLP problem in which continuity conditions in each shooting are considered as equalities and state constraints at the end of each shooting as inequalities [9] with RK4 integration method being used as a dedicated integrator.

1) Mathematical Formulation: The algorithm for the proposed Model Predictive Control (RHC in this case) is described in algorithm 1. At time-step k the state observation is represented by $X[K]$ and the optimal control trajectory is represented by u_k^*

Algorithm 1: MPC with Receding Horizon

```
1 At time step $k$ , solve the OLOCP (using Simultaneous
    Direct Multiple Shooting method) on-line with initial
    state of the iteration being the latest state observation
    $\bar{x_0} = X[k]$.
2 The OCP outputs a control trajectory from which the
    first control input is applied $U_k = u^*[0]$
3 The new state value after applying the control input is
    obtained and the initial state is updated with the state
    observation and the OLOCP is solved at time $k$+1
    with $\bar{x_0}$ = $X[k+1]$.
4 The steps are repeated at each sample time with the
    Horizon length of each iteration decremented by a
    value of 1.
```

The OLOCP is formulated as an objective function that minimises cost over the Horizon length. The cost term penalises magnitude of state and control variables such that equilibrium position with minimal cost is achieved. The Horizon length is represented by N.

The objective of the open-loop optimal control problem is described as,

$$\min_{x_1 \dots x_N, u_0 \dots u_{N-1}} \qquad L_N \tag{7}$$

where L_N is the accumulated cost at time-step N with a forgetting/discount factor. This was done for our method to be as similar to the RL algorithm as possible. The equation is given as,

$$L_i = E_i(\theta_i, \omega_i, u) + \gamma * L_{i-1} \quad \forall i \in 1, 2..N \tag{8}$$

with the discounting factor set to $\gamma = 0.99$ and $E_i(\theta_i, \omega_i, u)$ representing the penalty on the state and control magnitudes at each time-step i, given by,

$$E_i(\theta_i, \omega_i, u) = \theta_i^2 + 0.1 * \omega_i^2 + 0.001 * u_i^2 \quad u_i \in \mathbb{R} \tag{9}$$

the equation to simulate the next state is given by,

$$x_{i+1} = f_i(x_i, u_i) \qquad \forall i \in 0, 1..N-1 \tag{10}$$

where, $x_i = [\theta_i, \omega_i]^T$ and $\theta, \omega \in \mathbb{R}$. The Equation 10 represents the system dynamic of the pendulum, given by

$$\dot{\theta} = \omega \quad \text{and} \quad \dot{\omega} = -1.5 * \frac{g * sin(\theta)}{l} + 3 * \frac{u}{m * l^2} \tag{11}$$

with $\{g = 9.8ms^{-2}, \quad m = 1kg, \quad l = 1m\}$ being the values pertaining to characteristics of the pendulum and acceleration due to gravity as given in the Gym environment. The dynamics in Equation 11 are approximations to describe the movement of the pendulum in an ideal scenario, i.e., with no friction or similar opposing forces.

The equality constraints of the problem are given by $g_i(x_i, u_i) = 0$ where g consists of the following equations,

$$\begin{bmatrix} x_{i+1} - f_i(x_i, u_i) \\ x_0 - \bar{x_0} \\ E_0 \\ E_N \end{bmatrix} = 0 \tag{12}$$

with E_0, E_N being penalty at initial and terminal state respectively.

The inequality constraints of the problem are given by $h_i(x_i, u_i) \geq 0$ where h consists of the following equations,

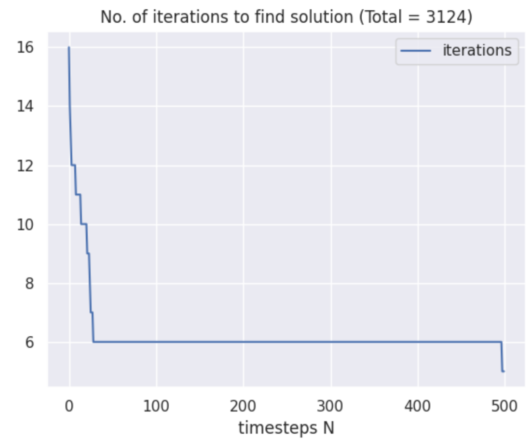
$$\begin{bmatrix} u_i + T_{max} \\ T_{max} - u_i \\ \omega_i + \omega_{max} \\ \omega_{max} - \omega_i \\ \theta_i + \theta_{max} \\ \theta_{max} - \theta_i \end{bmatrix} \geq 0 \tag{13}$$

where u_i exists for i in {0,1...N-1} and ω_i and θ_i for i in {0,1...N} with the limits being $T_{max} = 2\,\text{Nm}$, $\omega_{max} = 8\,s^{-1}$, $\theta_{max} = \pi\,\text{Radians}$. These limits are set to ensure that the state and control values remain constrained in a given range. The optimiser is free to choose any value in these limits and optimise for the cost. The full formulation can be referenced in subsection VI-A.

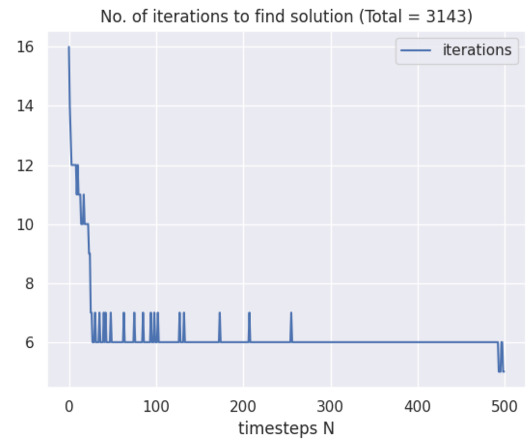
2) Algorithm Implementation: The objective of OLOCP expresses our desire to bring the state close to the origin, while using not too much control effort, as quantified by deviations of the control variable from 0. In simultaneous approach, both states and controls are kept as decision variables, then a larger problem is obtained with a special sparsity structure [4]. The function $f_i(x_i, u_i)$ in Equation 10 is an RK4 integrator [8] with 10 intermediate RK4 steps over a total integration time of h = 0.1. We implement the algorithm with discrete time horizon length N = 500 such that the continuous time horizon T = 50s. The initial state $\bar{x_0}$ would be a combination of θ and ω in the limits, randomly sampled from a uniform distribution. The algorithm upon solving returns the optimal control trajectory. With implementation of Model Predictive Control, the most recent state is observed after the first control input is administered to the system. Upon setting the initial state for the next open-loop OCP as the observed state, the updated problem is solved again for a new optimal control trajectory. This process is repeated for set horizon length. Receding Horizon Control was implemented to reduce the dimensions as we iterate, thus, leading to faster solving times. Implementing vanilla Model Predictive Control without receding Horizon leads to similar outputs albeit with high computational cost.

III. Implementation

We have used different test-benches to implement Reinforcement learning and Optimal Control algorithms. For Deep RL Control, PyTorch deep learning framework was used to construct the neural networks and train them.
For implementing MPC, we have used Casadi for Python 3.6. For testing and rendering both the algorithms, we have used a customised OpenGym AI pendulum environment executed on Ubuntu 18.04 OS running on a Intel Core i5 processor @ 1.6Ghz.



(a) Without additive noise



(b) With additive noise

Fig. 5: Iterations to solve each OLOCP in MPC

IV. Results

To showcase the results we have chosen a fixed initial state $\bar{x_0} = [\pi, 0]$ such that performance of both the algorithms have comparable outputs. To mimic a more real-world approach we have considered a case where there is a noise in the angular velocity when observing the pendulum state. This means that the angular velocity (ω) varies slightly in the observation than in the simulated state. When the feedback loop completes, the algorithm has to adjust to this changes. The noise added to angular velocity is here-forth mentioned as additive noise.

Figure 6 and Figure 7 showcase the outputs of the MPC and Deep RL control respectively in the case of no additive noise on the state, i.e., an ideal scenario. We can observe that the state trajectories are quite smooth and converge in under 100 time-steps. The reward and cost trajectories are identical for both the approaches but the negative of each other.

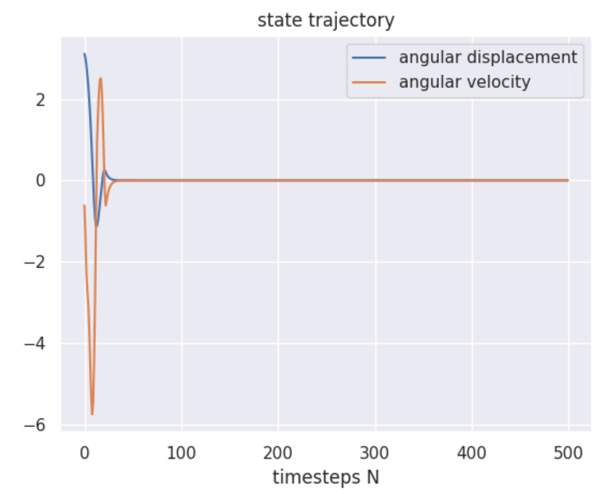
Figure 8 and Figure 9 showcase the outputs of the algorithms in the case of additive noise on the state, i.e., a more closer to real-world scenario. We can observe that the state trajectory is considerably smoother in the output of MPC. The RL agent tries to find the goal state, but fails to reach the upright condition and stay in that position. This can be observed more clearly in the control trajectory output of RL agent in Figure 9b. In both cases, the states do not fully converge to zero in the given time horizon as they keep oscillating slightly around the goal state. The cost/reward in both cases converges to a value close to zero.

In the case of Model Predictive Control, it can be observed that the number of iterations required to solve the open-loop OCP reduce over time. This could be attributed to the solver converging on the solution as well as reducing dimension due to receding horizon. In the case of added noise, the solver required more slightly more iterations to arrive at the goal state.

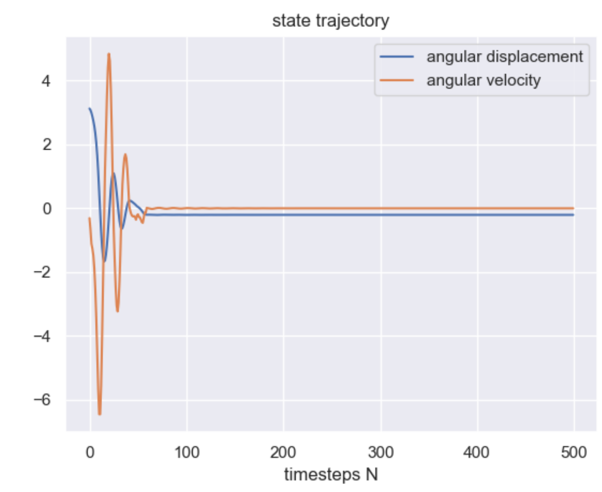
V. Conclusion and outlook

To conclude, both MPC and Deep RL control algorithms are robust to initial state, having similar results with random starts. The control trajectory outputted by Deep RL agent is considerably noisy when compared to the control output by MPC, thus needing more refinements if needed to apply for control problems in real world. The observed positives for Optimal Control are that the algorithm required lesser iterations for converging to goal state, it also had a smoother convergence while at the same time does not need pre-training. At the same time, RL agent can be made to learn the optimal policy for control from scratch, it does not require a defined model. RL models always require huge amount of training data, the agent we presented can also perform better if more training data is provided. Both the algorithms are closely similar in terms of reward or cost function used. The forgetting factor introduced to the OCP formulation makes it converge slightly faster, since the latest cost is given a higher weight.

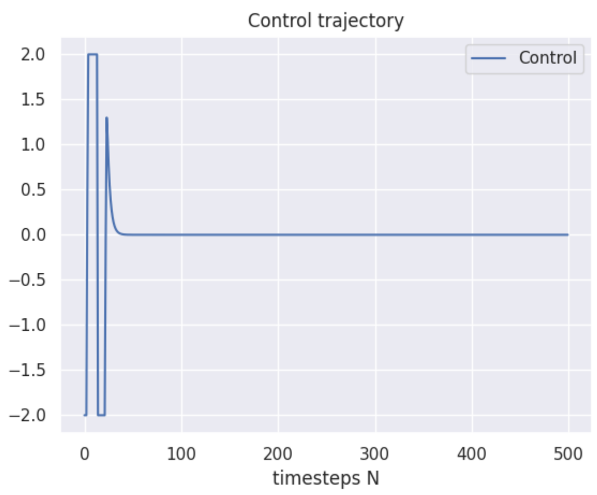
Reinforcement algorithms are sample intensive and requires data, high computing for training purpose. Whereas, focus of MPC would be extract the next best possible response with minimum latency and to solve the non-linear optimal control problem at every iteration in minimum time. As a result, presently, MPC controllers are deployed in real time embedded use cases compared to RL controller. RL algorithms have the potential for becoming smarter and more stable, although they need some improvements from the current models. This would be an interesting research topic for future work.



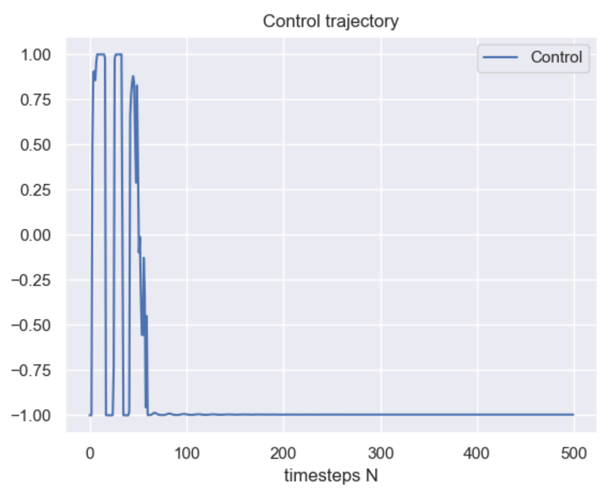
(a)



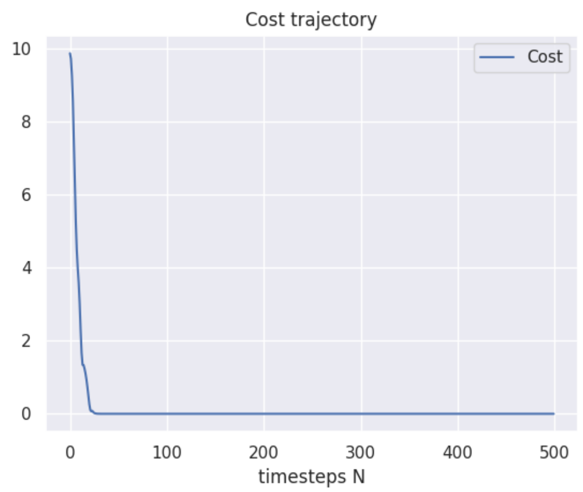
(a)



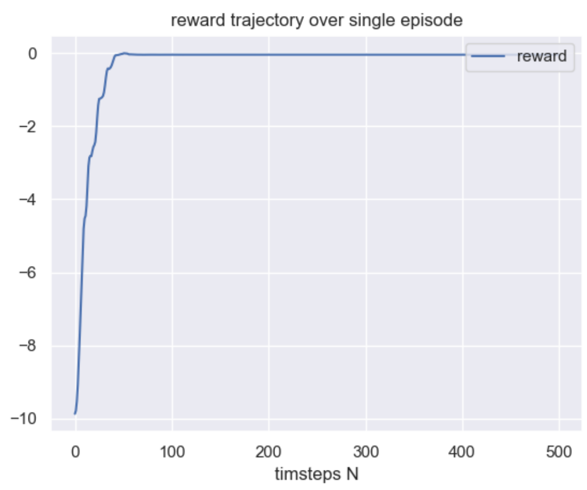
(b)



(b)



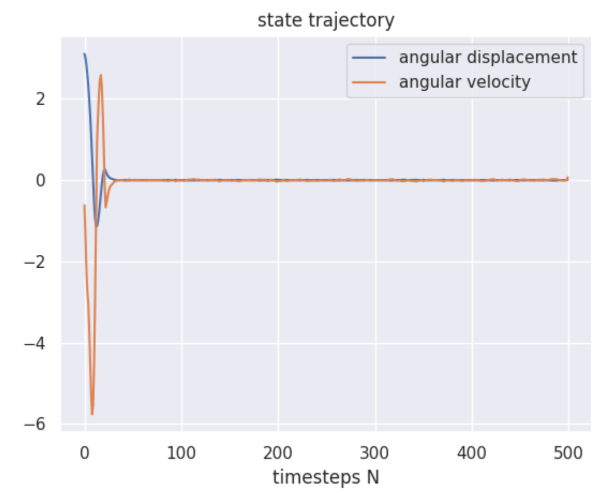
(c)



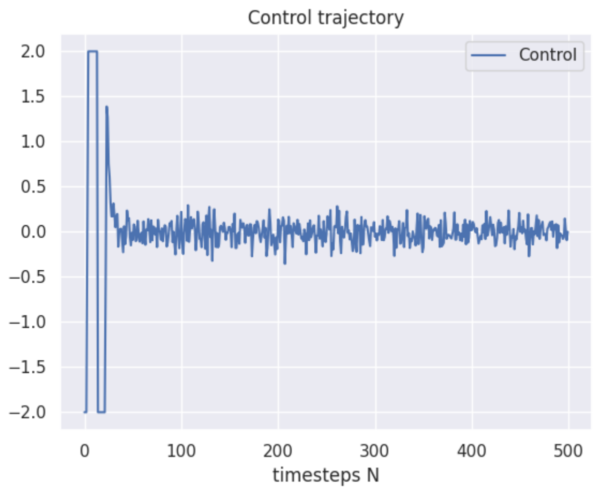
(c)

Fig. 6: RHC without additive noise

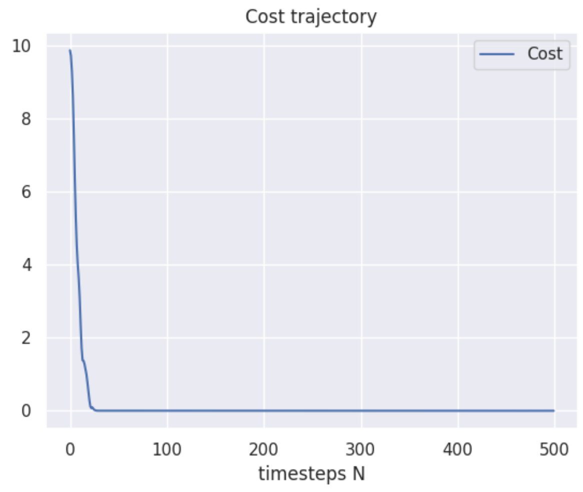
Fig. 7: Deep RL control without added noise



(a)

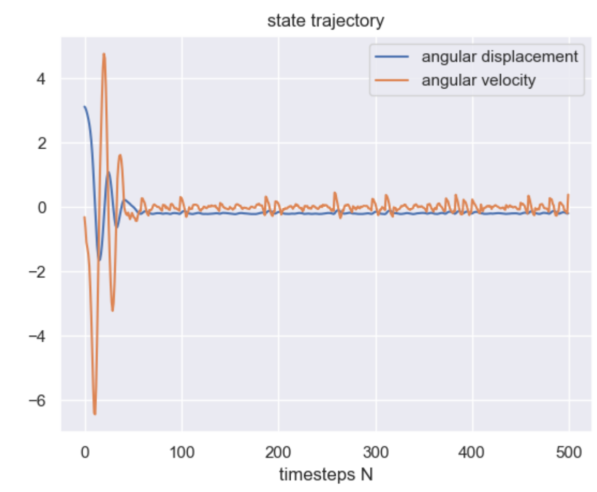


(b)

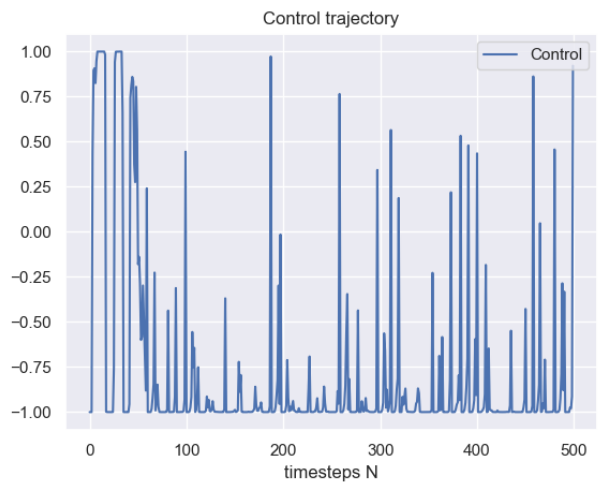


(c)

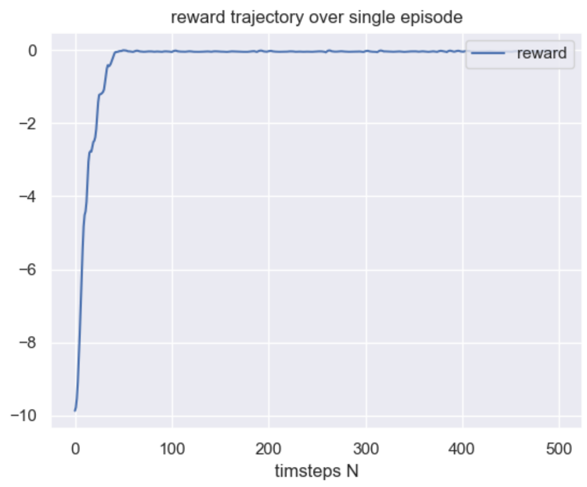
Fig. 8: RHC with additive noise



(a)



(b)



(c)

Fig. 9: Deep RL Control with additive noise

References

[1] Shubhendu Bhasin. Reinforcement learning and optimal control methods for uncertain nonlinear systems. 2011.

[2] Lorenz T Biegler. A survey on sensitivity-based nonlinear model predictive control. *IFAC Proceedings Volumes*, 46(32):499 – 510, 2013. 10th IFAC International Symposium on Dynamics and Control of Process Systems.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[4] Moritz Diehl. Lecture notes on numerical optimization(preliminary draft). 2017.

[5] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv e-prints*, page arXiv:1509.02971, September 2015.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1312.5602, December 2013.

[7] Sebastian Sager. Numerical methods for optimal control with binary control functions applied to a lotka-volterra type fishing problem. In Alberto Seeger, editor, *Recent Advances in Optimization*, pages 269–289, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[8] A. Schwartz and E. Polak. Consistent approximations for optimal control problems based on runge–kutta integration. *SIAM Journal on Control and Optimization*, 34(4):1235–1269, 1996.

[9] Jasem Tamimi and Pu Li. Nonlinear model predictive control using multiple shooting combined with collocation on finite elements. *IFAC Proceedings Volumes (IFAC-PapersOnline)*, 7, 07 2009.

VI. Appendix

A. Formulation

$$\begin{aligned}
&\min_{x_1 \dots x_N, u_0 \dots u_{N-1}} && L_N && \forall i \in 1,2,..N \\
&\text{subject to} && \begin{bmatrix} x_{i+1} - f_i(x_i, u_i) \\ x_0 - \bar{x_0} \\ E_0 \\ E_N \end{bmatrix} = 0 && x_i = [\theta_i, \omega_i]^T \quad \theta_i, \omega_i, u_i, \in \mathbb{R} \\
& && \begin{bmatrix} u_i + T_{max} \\ T_{max} - u_i \\ \omega_i + \omega_{max} \\ \omega_{max} - \omega_i \\ \theta_i + \theta_{max} \\ \theta_{max} - \theta_i \end{bmatrix} \geq 0 && \{T_{max} = 2\,\mathrm{Nm},\ \omega_{max} = 8\,\mathrm{s}^{-1},\ \theta_{max} = \pi\,\mathrm{Rad}\} \\
&\text{such that} && L_i = E_i(\theta_i, \omega_i, u) + \gamma * L_{i-1} && \{\gamma = 0.99\} \quad \forall i \in 1,2..N \\
& && E_i(\theta_i, \omega_i, u) = \theta_i^2 + 0.1 * \omega_i^2 + 0.001 * u_i^2 && \forall i \in 1..N-1 \\
& && x_{i+1} = f_i(x_i, u_i) && \forall i \in 0,1..N-1 \\
&\text{System Dynamics} && \dot{\theta} = \omega && \\
& && \dot{\omega} = -1.5 * \frac{g * sin(\theta)}{l} + 3 * \frac{u}{m * l^2} && \{g = 9.8\mathrm{ms}^{-2}, l = 1\mathrm{m}, m = 1\mathrm{Kg}\} \\
&\text{with} && f_i(x_i, u_i) \text{ denoting the explicit RK4 integrator} &&
\end{aligned} \tag{14}$$

B. DDPG Algorithm

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
   Initialize a random process $\mathcal{N}$ for action exploration
   Receive initial observation state $s_1$
   for t = 1, T do
      Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
      Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
      Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
      Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
      Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
      Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
      Update the actor policy using the sampled policy gradient:
         $$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$
      Update the target networks:
         $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
         $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
   end for
end for
```