FINE-TUNING LINEAR QUADRATIC REGULATOR: FROM REINFORCEMENT LEARNING'S PERSPECTIVE

Lihan Zha
Weiyang College
Tsinghua University

ABSTRACT

Linear quadratic regulator (LQR) often requires great manual efforts to tune the cost matrices in order to work properly. The paper proposes a framework on automatic fine-tuning LQR from the perspective of reinforcement learning. Through parameterizing the cost matrices with neural network, it is possible to do direct optimization on cost matrices with explicitly defined reward functions through policy gradient. The paper demonstrates the proposed method can find optimal cost matrices and outperforms naive policy gradient.

1 INTRODUCTION

Nearly all control methods require fine-tuning of controller's parameters. From PID to reinforcement learning, the controller's performance is closely related to the optimality of its parameters (Mason et al., 2014). For example, to balance a invert pendulum with LQR requires careful tuning of the scale between two cost matrices for quicker convergence (Zabihifar et al., 2020). For quadrotors, each single element in the cost matrices determines the success of control and requires very careful tuning (Martins et al., 2019).

However, this often requires great human efforts and experiences to find an appropriate set of parameters, and is usually done through naive trial-and-error. It is impossible to find a general methodology to automatically tune all kinds of controllers, given the different nature and characters of them. Yet for certain specific kinds of controller, for example LQR, there is also no general ways to fine-tuning its cost matrices(Anderson & Moore, 1990), which are crucial to the performance of it.

Prior works try to solve this problem through Bayesian optimization (Marco et al., 2016) and gradient approximation (Mason et al., 2014). However, both methods are time-consuming and not conceptually simple. Furthermore, Bayesian optimization requires Gaussian process to infer the cost function, which further restrict the expressiveness of LQR while suffering from training instability. Gradient approximation is not sample-efficient since it only leverages local data and does not take advantage of other available data. What's more, both methods do not exploit the physical meaning of cost matrices of LQR. Pan et al. adopts simultaneous perturbation stochastic approximation to fine-tune LQR to ameliorate the over-shooting problem. However, this method still suffers from instability and does not take the physical meaning of cost matrices into account.

In this work, in order to take into account the cost matrices' nature as weights of states and inputs, they are parameterized as diagonal matrices, with diagonal elements the outputs of a two-head neural network. With this parameterization, the cost matrices can be interpreted as the policy in reinforcement learning. Once the reward function is explicitly defined for downstream tasks, one can use policy gradient to find the optimal cost matrices given initial states. This method can generalize to all control tasks that requires parameters' fine-tuning and demands no extra manual tuning, so it can be embedded into the standard LQR framework.

2 Background

2.1 Linear Quadratic Regulator

Consider the discrete-time linear dynamic system,

$$x_{t+1} = Ax_t + Bu_t + \omega_t \tag{1}$$

where $x_t \in \mathbb{R}^n$ is the state; $u_t \in \mathbb{R}^m$ is the control input; $\omega_t \in \mathbb{R}^n$ is zero-mean noise; $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times m}$ are system matrices. The initial state $x_0 \in \mathbb{R}^n$ is sampled from state distribution $\mathcal{D}$. We assume that the system model has been linearized and neglect the linearization part here. From the view of dynamic programming, the goal of LQR is to find the optimal control to minimize the cost-to-go

$$C^*(Q,R) = \min_u \mathbb{E}_{x_0,\omega_t}\Big[\sum_t \big(x_t^TQx_t + u_t^TRu_t\big)\Big] \tag{2}$$

where $Q \succeq 0 \subseteq \mathbb{R}^{n\times n}$ and $R \succ 0 \subseteq \mathbb{R}^{m\times m}$ are symmetric positive semi-definite and positive definite cost matrices that can be manually set. Intuitively, the cost function is defined so as to ensure x and u both remain small for the purpose of control and saving energy. We assume (A, B) is controllable or stabilizable so that 2 has a unique solution

$$u^*(x) = -\big[B^TPB + R\big]^{-1}B^TPAx \tag{3}$$

where P can be uniquely determined through solving Discrete Algebratic Riccati Equation (DARE)

$$P = A^TPA + Q - A^TPB\big[B^TPB + R\big]^{-1}B^TPA \tag{4}$$

Since the optimal solution in 3 does not depend on noise ω_t, we can disregard ω_t in system dynamics 1, and will not affect our algorithm as well, we disregard ω_t in the rest of paper for simplicity.

2.2 Reinforcement Learning

In reinforcement learning, the problem is formulated a Markov decision process (MDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \gamma, p)$ (Sutton & Barto, 2018), with state space $\mathcal{S} \subseteq \mathbb{R}^n$,action space $\mathcal{A} \subseteq \mathbb{R}^m$, reward function $\mathcal{R} \subseteq \mathbb{R}$, constant discount factor $\gamma \in \mathbb{R}$, and transition dynamics $p \subseteq \mathbb{R}^{(m+n)\times n}$.

There is an agent interacting with the environment using its policy π, often parameterized as π_θ, and its goal is to find an optimal policy in order to maximize the cumulative rewards. Specifically, at each time step t, the agent observes a state $s_t \in \mathcal{S}$, get action a_t from its policy $\pi_\theta(a_t|s_t)$, act to the environment, receive a reward r_t from the environment, and move into the next state s_{t+1}. This process repeats and formulates a trajectory $\tau = (s_0, a_0, r_0, s_1, ..., a_T, r_T)$. The cumulative reward is defined as the sum of discount rewards along the trajectory $J(\tau) = \sum_{t=1}^{T}\gamma^{t-1}r_t$. The goal of the agent is to find the optimal policy π^* to maximize the expectation of cumulative rewards

$$\pi^* = \arg\max_{\pi_\theta} \mathbb{E}_{\tau\sim p_{\pi_\theta}(\tau)} J(\tau) \triangleq \arg\max_\theta J(\theta) \tag{5}$$

2.3 Policy Gradient

Policy gradient (PG) is an algorithm of reinforcement learning (Sutton et al., 1999), which aims to solve the optimization problem in 5. It directly does gradient ascent on the policy π_θ.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p_\theta(\tau)}\Bigg[\Bigg(\sum_{t=1}^{T}\nabla_\theta \log_{\pi_\theta}(a_t|s_t)\Bigg)\Bigg(\sum_{t=1}^{T} r(s_t,a_t)\Bigg)\Bigg] \tag{6}$$

In practice, we approximate 6 with Monte-Carlo method

$$\nabla_\theta J(\theta) \approx \sum_i \Bigg(\sum_t \nabla_\theta \log_{\pi_\theta}(a_t^i|s_t^i)\Bigg)\Bigg(\sum_t r(s_t^i,a_t^i)\Bigg) \tag{7}$$

After obtaining the gradient $\nabla_\theta J(\theta)$, we can do gradient ascent on θ

$$\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta) \tag{8}$$

3 FINE-TUNING LQR

3.1 PROBLEM FORMULATION

We parameterize cost matrices in 2 with $\mathbf{Q}(\boldsymbol{\theta})$ and $\mathbf{R}(\boldsymbol{\phi})$ respectively, where $\boldsymbol{\theta} \in \mathbb{R}^n$ and $\boldsymbol{\phi} \in \mathbb{R}^m$. The dimensions of $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ are fixed because we assume $\mathbf{Q}$ and $\mathbf{R}$ are diagonal matrices with $\theta_i > 0, i = 1, ..., n$ and $\phi_j > 0, j = 1, ..., m$. This assumption will not limit the expressiveness of $\mathbf{Q}$ and $\mathbf{R}$ much, because through eigendecomposition we can always let $\mathbf{Q} = \mathbf{\Lambda}$ and regard the matrix U as a rotation of x which does not affect much

$$x^T Q x = x^T U^T \Lambda U x = (Ux)^T \Lambda (Ux) \tag{9}$$

Through this parameterization, the cost can be represented as

$$C(\boldsymbol{\theta}, \boldsymbol{\phi}, k) = \sum_i^n \theta_i x_i^2 + k \sum_j^m \phi_j u_j^2 \tag{10}$$

where $\sum_i \theta_i = 1$ and $\sum_j \phi_j = 1$. $k \in \mathbb{R}^+$ is a scale factor that determines the relative scale between $\mathbf{Q}$ and $\mathbf{R}$. k is also a parameter that requires to be fine-tuned.

And the corresponding minimum cost in 2 can be re-expressed as

$$C^*(\boldsymbol{\theta}, \boldsymbol{\phi}, k) = \min_{\boldsymbol{\theta}, \boldsymbol{\phi}, k} C(\boldsymbol{\theta}, \boldsymbol{\phi}) \tag{11}$$

When varying $\boldsymbol{\theta}$, $\boldsymbol{\phi}$ and k, we can obtain different P in 4, thus adopt different u^* in 3, which will affect the performance of the system.

One way to evaluate the performance of the system is $C^*\big(\boldsymbol{\theta}, \boldsymbol{\phi}, k\big)$ in 11. However, this is not a good choice. The physical meaning of it is: given $\mathbf{Q}$ and $\mathbf{R}$, the minimum costs the system can achieve. It is not necessarily the case that the lower the cost, the better the performance of the system. For example, if an element x_i of the state vector $\boldsymbol{x}$ is much larger than other elements, putting a large weight θ_i on this element x_i will results in larger $C^*(\boldsymbol{\theta}, \boldsymbol{\phi}, k)$, however will result in better system performance if element x_i is the very state element that we should punish. Thus, $C^*\big(\boldsymbol{\theta}, \boldsymbol{\phi}, k\big)$ is not a good metric to evaluate how $\mathbf{Q}(\boldsymbol{\theta})$ and $\mathbf{R}(\boldsymbol{\phi})$ influence the performance of the system.

So in this paper, we design another metric inspired by reinforcement learning. We can define a reward function r_t according to our requirements of the controller. For example, if we want to stabilize the system around a certain point $\boldsymbol{x}_c$, we can define $r_t = L^2(\boldsymbol{x}_t - \boldsymbol{x}_c)$. With this reward function, our goal is finding the optimal $\boldsymbol{\Theta} = \big[\boldsymbol{\theta}, \boldsymbol{\phi}, k\big]$

$$\boldsymbol{\Theta}^* = \arg\max_{\boldsymbol{\Theta}} \sum_t r_t \tag{12}$$

This can be further formulated as a reinforcement learning problem if we view $\pi_\Phi(\boldsymbol{\Theta}|\boldsymbol{x})$ as the policy parameterized by Φ, which can be optimized via PG algorithm in 8. Note that Φ is the parameters of the neural network that parameterize π, while $\boldsymbol{\Theta}$ is the output of the neural network. Plainly speaking, we want to optimize Φ so that we can find the optimal "action" $\boldsymbol{\Theta}_t$ under different "state" $\boldsymbol{x}_t$ based on all the reward signals r_j^i we have received so that the performance of the system is as well optimized.

3.2 ALGORITHM

The full algorithm for fine-tuning LQR is summerized in Algorithm 1.

To improve the system's performance, we obtain $\boldsymbol{\Theta}_t$ from policy π, and then update $\mathbf{Q}(\boldsymbol{\theta}_\mathbf{t}), \mathbf{R}(\boldsymbol{\phi}_\mathbf{t})$, with which we can re-calculate $\boldsymbol{u}^*$ with 2,3 and 4. With the new $\boldsymbol{u}^*$ as control inputs, we can interact the environment and receive corresponding rewards, based on which we can update π_Φ so that we can obtain better$\mathbf{Q}(\boldsymbol{\theta}), \mathbf{R}(\boldsymbol{\phi})$.

Note that re-calculating $\boldsymbol{u}^*$ at every time step is computationally heavy and not necessary, so we only carry out this step at an interval c. Though we do not explicitly use the system matrices A and B in Algorithm 1, they are required in calculating $LQR(\mathbf{Q}, \mathbf{R})$ in line 8. The LQR interval c, policy update interval Δ and learning rate α are all hyper-parameters that should be tuned according to different tasks at hand.

To ensure the scale of $\mathbf{Q}(\boldsymbol{\theta})$ and $\mathbf{R}(\boldsymbol{\phi})$ is correct, the final layer of π_ϕ is a $Softmax$ layer, so that $\sum_j \theta_i = 1$ and $\sum_j \phi_j = 1$.

Algorithm 1: Fine-tuning LQR

```
Input: System matrices $A, B$, initial state distribution $\boldsymbol{x}_0 \sim \mathcal{D}$, initial policy $\pi_\Phi$, LQR interval
       $c$, environment $env$, policy update interval $\Delta$, trajectory number $N \leftarrow 0$, trajectory
       buffer $\mathcal{B} \leftarrow \emptyset$, learning rate $\alpha$
Output: Optimal policy $\pi^*$
1  while not done do
2      Sample $\boldsymbol{x}_0 \sim D$
3      $\tau \leftarrow ()$
4      for $t = 1, ..., T$ do
5          if $t \,\%\, c == 0$ then
6              $\boldsymbol{\Theta}_t \sim \pi_\Phi(\cdot|\boldsymbol{x}_{t-1})$
7              $\mathbf{Q}, \mathbf{R} \leftarrow \mathbf{Q}(\boldsymbol{\Theta}_\mathbf{t}), \mathbf{R}(\boldsymbol{\Theta}_\mathbf{t})$
8              $\boldsymbol{u}^* \leftarrow LQR(\mathbf{Q}, \mathbf{R})$
9          $\boldsymbol{x}_t, r_t \leftarrow env.STEP\left(\boldsymbol{u}^*(\boldsymbol{x}_{t-1})\right)$
10         $\tau.append(\boldsymbol{x}_{t-1}, \boldsymbol{\Theta}_t, r_t, \boldsymbol{x}_t)$
11     $N \leftarrow N + 1$
12     $\mathcal{B} \leftarrow \mathcal{B} \cup \tau$
13     if $N \,\%\, \Delta == 0$ then
14         Sample $\{\tau_i\}$ from $\mathcal{B}$
15         $\nabla_\Phi J(\Phi) \approx \sum_i \left(\sum_t \nabla_\Phi \log_{\pi_\Phi}(\boldsymbol{\Theta}_t^i|s_t^i)\right)\left(\sum_t r(s_t^i, \boldsymbol{\Theta}_t^i)\right)$
16         $\Phi \leftarrow \Phi + \alpha \nabla_\Phi J(\Phi)$
```

4 Experiments

In this part, we try to answer 2 questions:

- Can our method really fine-tune LQR?
- Is our method better than direct reinforcement learning?

We evaluate our algorithm on a 3D quadrotor (Sabatino, 2015). The model and corresponding variables depicting the dynamics are shown in Fig. 1. The state vector of the system is

$$\boldsymbol{x} = [\phi\ \theta\ \varphi\ p\ q\ r\ u\ v\ w\ x\ y\ z]^T \in \mathbb{R}^{12} \tag{13}$$

where $[x\ y\ z\ \phi\ \theta\ \varphi]^T$ is the vector containing the linear and angular position of the quadrotor in the earth frame, and $[u\ v\ w\ p\ q\ r]^T$ is the vector containing the linear and angular velocities in the body frame. The control input of the system is

$$\boldsymbol{u} = [f_t\ \tau_x\ \tau_y\ \tau_z]^T \in \mathbb{R}^4 \tag{14}$$

where f_t is the total thrust generated by rotors and $[\tau_x\ \tau_y\ \tau_z]^T$ are the control torques generated by differences in the rotor speeds. Details of the full dynamics and linearization of the model is in Appendix A.1. The linearized model is:

$$\dot{\boldsymbol{x}} = A\boldsymbol{x} + B\boldsymbol{u} \tag{15}$$

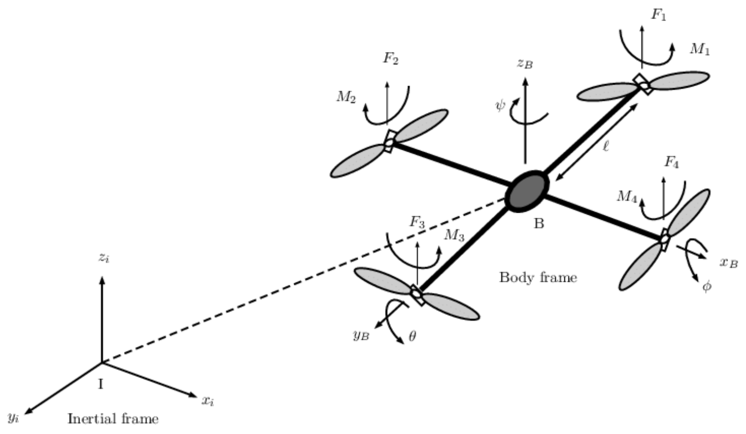

Figure 1: Model of 3D Quadrotor

where the system matrices A and B are

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -g & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ g & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{16}$$

$$B = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \frac{1}{I_x} & 0 & 0 \\ 0 & 0 & \frac{1}{I_y} & 0 \\ 0 & 0 & 0 & \frac{1}{I_z} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{m} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{17}$$

The corresponding parameters are listed in Table 1 (Belkheiri et al., 2012).

Table 1: Quadrotor Parameters

Parameter	**Value**
m	$1kg$
g	$9.81m/s^2$
I_x	$8.1 \times 10^{-3} kg \cdot m^2$
I_y	$8.1 \times 10^{-3} kg \cdot m^2$
I_z	$14.2 \times 10^{-3} kg \cdot m^2$

First, we try to evaluate whether our method work. We compare our method with naive LQR. The task is to maintain the balance of the quadrotor after an initial disturbance that pushes the quadrotor to 1 meters away in a random direction. Fig. 2 shows how is the quadrotor's deviation from rotor evolves with time. It can be seen that our method always converge faster than naive LQR, indicating that our method is able to find better cost matrices to improve the performance of the system.

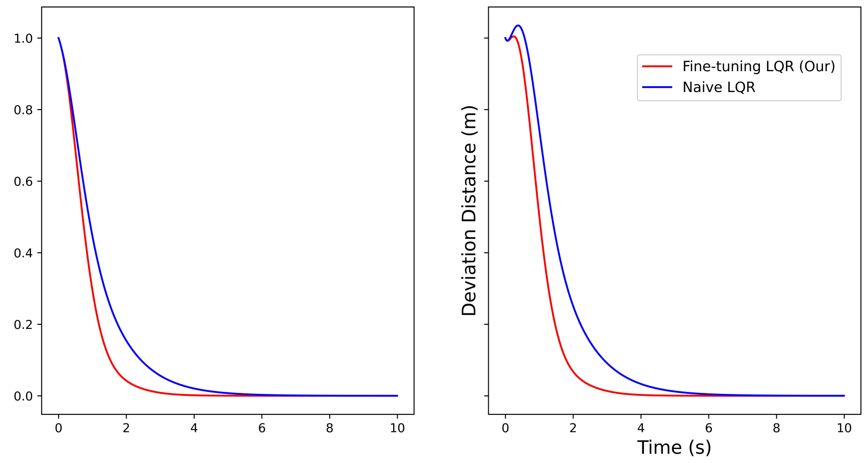

Figure 2: Quadrotor's deviation from the origin under random initial perturbations. Above figure shows how the deviation evolves with time under two different perturbation orientations.

Next, we compare our algorithm with standard LQR and vanilla PG (Peters & Schaal, 2006) in a trajectory-tracking task. The implementation of vanilla PG is similar to our method. The difference is that instead of learning $\pi_\phi(\boldsymbol{\Theta}|\boldsymbol{x})$, it directly learns $\pi_\phi(\boldsymbol{u}|\boldsymbol{x})$. Vanilla PG is a model-free method and does not take the dynamics of the system into account. The hyper-parameters of the algorithm and structure of neural network are listed in Appendix A.2. The result is shown in Fig. 3. It is proved that our method outperforms vanilla PG and can handle more complex tasks.

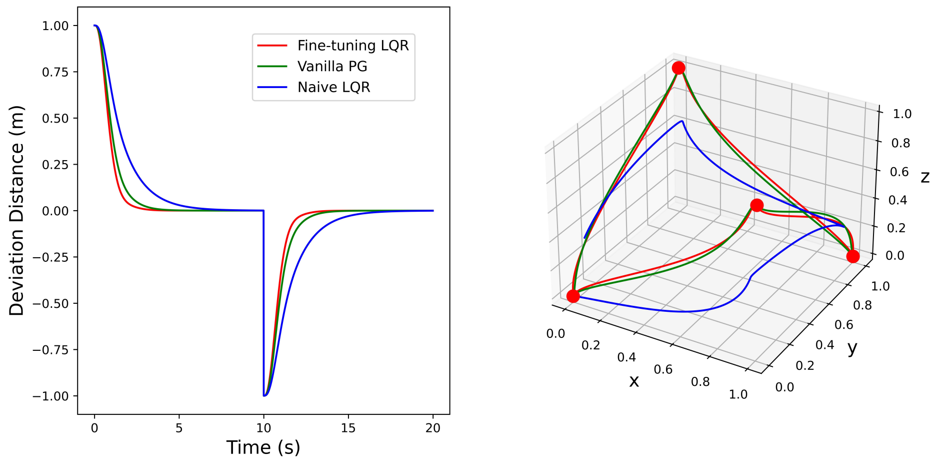

Figure 3: Left: Quadrotor's deviation from origin. Right: Trajectories generated by quadrotor. The red point indicates the reference points of the ideal trajectory. It can be seen that fine-tuning LQR gives the best result, while naive LQR fail in this task.

For naive LQR, as the linear approximation of the model becomes very inaccurate as the drone moves far away from the origin, it is doomed to fail in a tracking task. It is possible to use iterative LQR to update the approximation of the model from time to time so as to achieve better performance than naive LQR, but we don't consider it in this work due to time limit. It would be interested to investigate whether our method can outperform iterative LQR in future work.

The training curve of our method in both tasks is also visualized in Figure 4 for completeness. It can be seen that our method will converge after sufficient training, which indicates that it succeed in finding an optimal policy. Since tracking is more complex than merely stabilizing around the origin, it requires more time to train and has higher variance during training. The reason that the reward of tracking task may drop below 0 sometimes is that we define an additional negative termination reward term $r_f = -0.1$ here to discourage the policy from moving away from the target points.

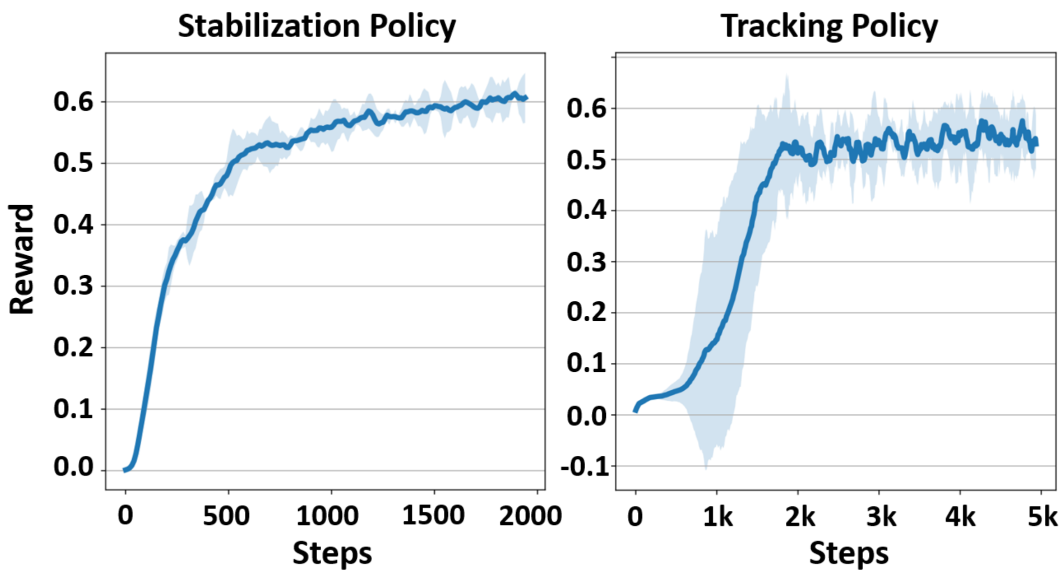

Figure 4: The training curve of both tasks. For method that leverage neural networks, randomness can affect the performance a lot, so all training curves are obtained through averaging over the results of 5 random seeds. The blue shadow part represents the variance across different seeds. "Steps" refers to the number of training iterations, and the absolute scale of "Reward" does not matter, as we don't explicitly constrain the meaning of it. We are only interested in the pattern of the curve, as it represents how quick our algorithm can converge.

5 Conclusions

In this work, we propose a novel method to fine-tune LQR with reinforcement learning. Compared with existing methods, our method is more stable and easier to implement. We experimentally prove that our method outperforms vanilla PG and naive LQR in a moderately complex dynamics system. Based on the observation that there's no general solution for tuning LQR cost matrices, our method try to introduce a highly non-linear function approximator to catch the complex nature of the problem. We believe that our work also shed light on a novel way to combine reinforcement learning with LQR, which is an exciting field that remains to be explored.

References

Brian D. O. Anderson and John B. Moore. *Optimal Control: Linear Quadratic Methods*. Prentice-Hall, Inc., USA, 1990. ISBN 0136385605.

M. Belkheiri, A. Rabhi, A. El Hajjaji, and C. Pegard. Different linearization control techniques for a quadrotor system. In *CCCA12*, pp. 1–6, 2012. doi: 10.1109/CCCA.2012.6417914.

Alonso Marco, Philipp Hennig, Jeannette Bohg, Stefan Schaal, and Sebastian Trimpe. Automatic LQR tuning based on gaussian process global optimization. *CoRR*, abs/1605.01950, 2016. URL `http://arxiv.org/abs/1605.01950`.

Luís Martins, Carlos Cardeira, and Paulo Oliveira. Linear quadratic regulator for trajectory tracking of a quadrotor. *IFAC-PapersOnLine*, 52(12):176–181, 2019. ISSN 2405-8963. doi: https://doi.org/10.1016/j.ifacol.2019.11.195. URL `https://www.sciencedirect.com/science/article/pii/S2405896319311450`. 21st IFAC Symposium on Automatic Control in Aerospace ACA 2019.

Sean Mason, Ludovic Righetti, and Stefan Schaal. Full dynamics lqr control of a humanoid robot: An experimental study on balancing and squatting. In *2014 IEEE-RAS International Conference on Humanoid Robots*, pp. 374–379, 2014. doi: 10.1109/HUMANOIDS.2014.7041387.

Kaiwen Pan, Yang Chen, Zhengxi Wang, Huaiyu Wu, and Lei Cheng. Quadrotor control based on self-tuning lqr. In *2018 37th Chinese Control Conference (CCC)*, pp. 9974–9979, 2018. doi: 10.23919/ChiCC.2018.8483978.

Jan Peters and Stefan Schaal. Policy gradient methods for robotics. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2219–2225, 2006. doi: 10.1109/IROS.2006.282564.

Francesco Sabatino. Quadrotor control: modeling, nonlinearcontrol design, and simulation. 2015.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249.

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pp. 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

Seyed Hassan Zabihifar, Arkady Semenovich Yushchenko, and Hamed Navvabi. Robust control based on adaptive neural network for rotary inverted pendulum with oscillation compensation. *Neural Comput. Appl.*, 32(18):14667–14679, sep 2020. ISSN 0941-0643. doi: 10.1007/s00521-020-04821-x. URL `https://doi.org/10.1007/s00521-020-04821-x`.

A Appendix

A.1 Model dynamics and linearization of 3D quadrotor

All the variables have been defined in Section 4. The dynamics of the quadrotor is:

$$\begin{cases} \dot{\phi} = p + r[c(\phi)t(\theta)] + q[s(\phi)t(\theta)] \\ \dot{\theta} = q[c(\phi)] - r[s(\phi)] \\ \dot{\varphi} = r\frac{c(\phi)}{c(\theta)} + q\frac{s(\phi)}{c(\theta)} \\ \dot{p} = \frac{I_y - I_z}{I_x} rq + \frac{\tau_x + \tau_{wx}}{I_x} \\ \dot{q} = \frac{I_z - I_x}{I_y} pr + \frac{\tau_y + \tau_{wy}}{I_y} \\ \dot{r} = \frac{I_x - I_y}{I_z} pq + \frac{\tau_z + \tau_{wz}}{I_z} \\ \dot{u} = rv - qw - g[s(\theta)] + \frac{f_{wx}}{m} \\ \dot{v} = pw - ru - g[s(\phi)c(\theta)] + \frac{f_{wy}}{m} \\ \dot{w} = qu - pv - g[c(\theta)c(\phi)] + \frac{f_{wz} - f_t}{m} \\ \dot{x} = w[s(\phi)s(\varphi) + c(\phi)c(\varphi)s(\theta)] - v[c(\phi)s(\varphi) - c(\varphi)s(\phi)s(\theta)] + u[c(\varphi)c(\theta)] \\ \dot{y} = v[c(\phi)c(\varphi) + s(\phi)s(\varphi)s(\theta)] - w[s(\phi)c(\varphi) - s(\varphi)c(\phi)s(\theta)] + u[c(\theta)s(\varphi)] \\ \dot{z} = w[c(\phi)c(\theta) - u[s_\theta] + v[c(\theta)s(\phi)] \end{cases} \tag{18}$$

where $s(\cdot)$, $c(\cdot)$ and $t(\cdot)$ are short-hands for $\sin(\cdot)$, $\cos(\cdot)$ and $\tan(\cdot)$. In order to perform the linearization, the equilibrium point is chosen as:

$$\bar{\boldsymbol{x}} = [0\;0\;0\;0\;0\;0\;0\;0\;0\;\bar{x}\;\bar{y}\;\bar{z}]^T \tag{19}$$

And the corresponding control inputs is

$$\bar{\boldsymbol{u}} = [mg\;0\;0\;0]^T \tag{20}$$

We can set $A = \frac{\partial f(\boldsymbol{x},\boldsymbol{u})}{\partial \boldsymbol{x}} | \boldsymbol{x} = \bar{\boldsymbol{x}}, \boldsymbol{u} = \bar{\boldsymbol{u}}$ and $B = \frac{\partial f(\boldsymbol{x},\boldsymbol{u})}{\partial \boldsymbol{x}} | \boldsymbol{x} = \bar{\boldsymbol{x}}, \boldsymbol{u} = \bar{\boldsymbol{u}}$ to get the linearized system in 16 and 17.

A.2 Hyperparameters and Model Structure

For fine-tuning LQR, the policy is parameterized by a three-layer MLP. The input layer's dimension is 12 (the dimension of state vector), the hidden layer's dimension is 64, and the output layer is a three-headed layer, with one head outputting $\boldsymbol{\theta}$ with dimension 12 (the dimension of state vector), one head outputting $\boldsymbol{\phi}$ with dimension 4 (the dimension of control input), and one head outputting k with dimension 1. The former two heads are passed through a $Softmax$ layer respectively before final output to re-scale them to 1. The activation function is $ReLU$.

For vanilla PG, the policy is also parameterized by a three-layer MLP, with input layer dimension 12, hidden layer dimension 64, and output layer dimension 4. The activation function is $ReLU$.

Both fine-tuning LQR and vanilla PG are optimized with $Adam$ optimizer with learning rate $\alpha = 0.001$. For fine-tuning LQR, the initial state distribution $\mathcal{D}$ is chosen to be a uniform distribution defined on all the valid states, the trajectory length $T = 100$, LQR interval $c = 20$ and policy update interval $\Delta = 5$.