Reinforcement Learning for Robotics: Hands-on Project

Zhaobo Xu and Mun Seng Phoon

I. INTRODUCTION

The environment for this project is a pole attached to an unactuated joint on a cart, which moves along an infinite track. The pole starts downright and the task is to swing the the pole up and hold without falling over by applying horizontal forces to the cart.

For this environment, the upward balanced pole position is defined as 0 radians, and the downward hanging position is π radians. The force action signal from the agent to the environment is from -10 to 10 Newton. The state variables are as shown in Table I.

TABLE I
STATE OF THE CART-POLE ENVIRONMENT

State s			
variable	Observation	Min	Max
x	Cart Position	-6	6
$\dot{x}$	Cart Velocity	-10	10
θ	Pole Angle	$-\pi$	π
$\dot{\theta}$	Pole Angular Velocity	-10	10

If the cart moves more than 6 units away from center, it will terminate the episode during the training.

The reward function used in this project is [1] :

$$r(s) = -(1 - \exp(-0.5(s - s_{end})T^{-1}(s - s_{end})^{\mathsf{T}})) \quad (1)$$

where

$$T^{-1} = \begin{bmatrix} 1 & l & 0 \\ l & l^2 & 0 \\ 0 & 0 & l^2 \end{bmatrix} \quad (2)$$

and the state used here is an alternation from the original:

$$\begin{aligned} s = \; & [x, \sin\theta, \cos\theta] \\ s_n = \; & [0, 0, 1] \end{aligned} \quad (3)$$

We implemented two method using optimal control and model-free reinforcement learning respectively.

II. ITERATIVE LINEAR QUADRATIC REGULATOR

Optimal control can be viewed as an alternative perspective of reinforcement learning. The idea is to find the optimal control path of a forced system by minimizing some cost functions. Iterative linear quadratic regulator(iLQR)[2] is a commonly used optimal control method for nonlinear systems.

A. Algorithm

The cost of each step is as follows:

$$l = \mathbf{x_t^{\mathsf{T}} Q x_t} + \mathbf{u_t^{\mathsf{T}} R u_t}; \quad l_{final} = \mathbf{x_N^{\mathsf{T}} Q_{final} x_N} \quad (4)$$

The goal is to minimize the summation of the above terms. This optimization can be done by dynamic programming. Define the minimal cost-to-go at time step t as:

$$\begin{aligned} V_t &= min_u \sum_{i=t}^{N-1} l(\mathbf{x_i}, \mathbf{u_i}) + l_{final} && (5) \\ &= min_u[l(\mathbf{x_t}, \mathbf{u_t}) + V_{t+1}(\mathbf{f}(\mathbf{x_t}, \mathbf{u_t}))] && (6) \end{aligned}$$

here $\mathbf{f}$ denotes the system dynamics. The above equation can clearly be solved from the final state to the initial. The algorithm consists of two parts: The backward pass solves for $\mathbf{u_t}^*$ for a given trajectory; the forward pass will apply the actions $\mathrm{u}_{0:N}$ and evaluate the cost.

In the backward pass, assume a perturbation around any point $(\mathbf{x}, \mathbf{u})$, the change in cost is denoted as function $Q(\delta\mathbf{x}, \delta\mathbf{u})$. The problem then changes to minimizing the Q function. Using Taylor expansion and first order condition we can solve for the estimated best policy change for each step $\delta\mathbf{u}_t^*$:

$$\begin{aligned} \mathbf{k} &= -Q_{\mathbf{uu}}^{-1} Q_{\mathbf{uu}} && (7) \\ \mathbf{K} &= -Q_{\mathbf{uu}}^{-1} Q_{\mathbf{ux}} && (8) \\ \delta\mathbf{u}_t^* &= \mathbf{k} + \mathbf{K}\delta\mathbf{x}_t && (9) \end{aligned}$$

here $Q_{\mathbf{uu}}$ and $Q_{\mathbf{xu}}$ are entries in Taylor expansion of the Q function(see Appendix)

The forward pass will apply these actions as:

$$\begin{aligned} \hat{\mathbf{x}}_0 &= \mathbf{x}_0 && (10) \\ \hat{\mathbf{u}}_t &= \mathbf{u}_t + \alpha\delta\mathbf{u}_t^* && (11) \\ \hat{\mathbf{x}}_{t+1} &= \mathbf{f}(\hat{\mathbf{x}}_t, \hat{\mathbf{u}}_t) && (12) \end{aligned}$$

α here is for line search. after getting the trajectory, we will evaluate the total cost. This will be the criteria of convergence and α is required to guarantee the cost going down rather than bumping around.

The above trajectory will then be used for next backward pass, until the system trajectory converged. Since Q function is optimal, no perturbation can be applied, the final trajectory is thus desired system behavior.

Algorithm 1 iLQR

```
Input: horizon N, cost J, system dynamics f
initialize trajectory($x, u$); V$_x(N+1), V_{xx}(N+1)$,
 $V_u(N+1)$
 while not converged do
    for t = N, N-1 ... 0 do
        derive $Q_x(t)$, $Q_{xx}(t)$ ...using $V$, $f$ and $(x(t), u(t))$
        derive $k(t)$ and $K(t)$ using $Q$
        derive $V_x(t)$, $V_{xx}(t)$ using $k$ and $K$
    end
    while not optimal do
        for t = 0 ... N do
            calculate new trajectory as (10)-(12)
        end
        do line search
    end
    if converged then
        return current trajectory
    end
    update trajectory
end
```

B. Results

Normally, the iLQR will converge very fast. The training episodes used is around 200.

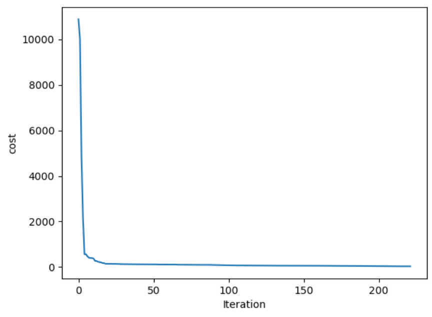

Fig. 1. cost change in each episode

Result relies on following perspectives:

1) Cost Matrix: The matrices **Q** and **R** in the cost function can be viewed as penalty terms. This project will use eq.2 as the penalty for state. The penalty for action is not given and thus set to be a small value.

These matrices will strongly affect the final result. For example, if penalty for actions is large, the agent will tend to be more conservative.

2) Horizon: In optimal control, the agent wouldn't try to finish the task as quickly as possible but rather try to reach the target state exactly at the final step. Control horizon defines such final step. If the horizon is too small, there will be not enough time to apply any possible controls. If the horizon is too large, the system will do useless actions.

3) Regularization: Chances are, that the hessian of Q function is not positive definite and the FOC is hence not complete. A general idea is to add regularization terms to the calculation of hessian. We adopted the method in [3]. The full steps are included in the appendix. Without the regularization, the result will most likely diverge.

III. Deep Q-Learning Network (DQN)

The deep Q-network (DQN) algorithm is a model-free, online, off-policy reinforcement learning method. A DQN agent is a value-based reinforcement learning agent based on Q-learning [4].

A. Algorithm

In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions (state-action values) is generated as the output. Each output element represents the expected cumulative long-term reward for taking the corresponding discrete action from the state indicated by the observation inputs.

We use the two important ingredients of the DQN algorithm: target network and experience replay [5]. Instead of using one neural network for learning, we use two: policy network Q and target network $\hat{Q}$. The target network has the same network architecture as the policy network but with frozen parameters. The parameters are copied from the policy network to the target network for every C iterations. This leads to more stable training because it keeps the target function fixed and reduces correlations between targets and Q values.

For the experience replay, observed transitions are stored in the memory buffer to be reused later. Instead of updating from the last transition, the network is updated from a batch of randomly sampled transitions from the same experience replay. This can stabilize and improve the DQN training procedure [5].

```
Initialize network $Q$
Initialize target network $\hat{Q}$
Initialize experience replay memory $D$
Initialize the Agent to interact with the Environment
while not converged do
    /* Sample phase
    $\epsilon \leftarrow$ setting new epsilon with $\epsilon$-decay
    Choose an action $a$ from state $s$ using policy $\epsilon$-greedy($Q$)
    Agent takes action $a$, observe reward $r$, and next state $s'$
    Store transition $(s, a, r, s', done)$ in the experience replay memory $D$

    if enough experiences in $D$ then
        /* Learn phase
        Sample a random minibatch of $N$ transitions from $D$
        for every transition $(s_i, a_i, r_i, s'_i, done_i)$ in minibatch do
            if $done_i$ then
                $y_i = r_i$
            else
                $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
            end
        end
        Calculate the loss $\mathcal{L}$ =1/N $\sum_{i=0}^{N-1}(Q(s_i, a_i) - y_i)^2$
        Update $Q$ using the SGD algorithm by minimizing the loss $\mathcal{L}$
        Every $C$ steps, copy weights from $Q$ to $\hat{Q}$
    end
end
```

Fig. 2. Deep Q-Learning Network (DQN) algorithm

Figure 2 shows the DQN algorithm used in the project. In addition to that, we implemented a few improvements in deep Q-learning following the extensions summarized in the Rainbow paper [6].

1) Double deep Q-learning: Traditional deep Q-learning tends to overestimate the reward, which leads to unstable training and lower quality policy. We use double deep Q-learning network (DDQN) to overcome the overestimation bias by selecting actions according to the policy network, but quantifying the return for this action using the target network. The idea behind double DQN is to minimize the problem of overestimating Q values, caused by max operators in the target values [7]. We change the equation of Q target y_i to:

$$Q(s,a) = r(s,a) + \gamma \hat{Q}\left(s', \operatorname{argmax}_a Q\left(s', a\right)\right) \tag{13}$$

2) Soft update of target network: Instead of updating weights of the target network after a certain number of episodes (hard update), we incrementally update the target network in smaller increments. At every τ step, we copy the weights parameters from the DQN policy network to update the target network.

$$\theta_{\hat{Q}} = \tau\theta_Q + (1-\tau)\theta_Q \tag{14}$$

3) Dueling DQN: Dueling architecture splits the DQN network into two separate streams, one for estimating the state-value and the other for estimating state-dependent action advantages [8]. After the two streams, the last module of the network combines the state-value and advantage outputs.

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a)\right) \tag{15}$$

B. Results

We use a simple feed forward network with one hidden layer. The dimension of the output layer is determined by the number of actions we have in the environment. In this project, we use discrete action space in DQN, 10 action spaces which represents the 10 force actions (-10, -8, ..., +8, +10). Two networks are created: a target network to evaluate the Q value and a policy network to choose the best action. We run DDQN on the Cartpole environment, with a sliding replay memory of 10000 memories, using a linear-annealed epsilon-greedy technique from 1.0 to 0.1 over 1500 episodes. Each episode consists of 500 steps but it will terminate when the cart reaches the edge of the screen.

The results in Figure 3 is the total accumulated reward if we have reached the end of each episode. There is a trade-off between number of actions taken in an episode and the total reward, as the reward function of CartPole environment outputs only negative values. If no penalty is set to limit the agent, the agent will try to reach the end of the episode as fast as possible by moving towards the episode termination criteria. In this unlikely case, the agent converged to the episodes with less actions taken as those yield the less negative total rewards. Therefore, a penalty (negative reward) is given to the terminated episodes.

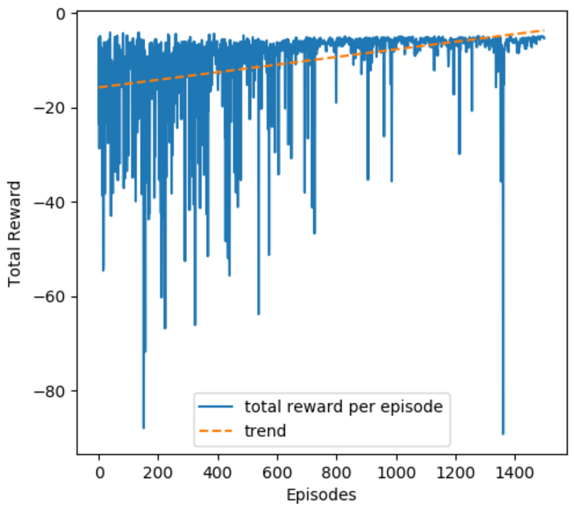

Fig. 3. Total rewards per episode in training of DQN agent. (Hyperparameters: discount factor = 0.99, minibatch size = 256, learning rate = 0.001, target network update rate = 100, replay memory buffer size = 10000)

DQN and DDQN (both with experience replay) are implemented to solve the cart-pole environment. Both of these deep learning approaches are able to learn and solve the problem sometimes, but not always. The learning of neural network is not stable and it is heavily influenced by the hyper-parameters e.g. batch size, learning rate, number of neurons in the hidden layer, and exploration rate. The same combination of hyperparameters perform differently in different random seeds.

IV. Comparison and Discussion

It is inspiring to view the different RL approaches in this perspective:

1) Optimal control: searching the optimal in state action space, guided by system dynamics.
2) Model-based approaches: searching the optimal in state action space, guided by a ”learned” or ”estimated” system dynamics
3) Model-free approaches: searching the optimal in state action space without guidance, using strategies and heuristics.

Normally, more information we have about the environment, faster the agent finds the correct path. Thus, optimal control methods usually converge fast, by taking advantage of the system dynamics. This is more of a ”guided” method to search the state action space. Most of the results of these methods, like DDP or iLQR are in closed forms. This offers more interpretability than NN based methods. The down side of optimal control is also obvious: the modelling of the system dynamics, including computing gradients is very complicated. This becomes even impossible for very complex systems. Therefore, the application field would be quite narrow for these approaches.

Compared to the first approach, it takes relatively longer time to train the DQN model by experimenting with the hyper-parameters to even just beat a random walk. After hours of debugging, the instability problems in DQN are connected to hyper-parameters like exploration technique, learning rate, target network update and size of the experience replay memory. In some iterations, the DQN agent will get stuck in local minimum which pushes the cart off of the environment before we reach the optimal reward. It is necessary to run comparisons across multiple random seeds.

In the second approach, DQN uses a discrete action space compared to continuous action space in iLQR. We break the output (state action values) of the universal function approximator up into multiple segments, and treat each one as a discrete action. Continuous analog of DQN is an actor-critic architecture [9], but it will become an on-policy method. Another alternative for deep Q-learning for continuous action spaces is to use Deep Deterministic Policy Gradient (DDPG) or deterministic policy gradient algorithm [10] that gives a distribution over actions as output to sample.

V. CONCLUSION

Both approaches finished the task but with some essential differences.

The iLQR method converges very fast and gives the optimal policy in closed form. However, this method acquires the system dynamics and thus not suitable for complex environments. This method needs regularization during the calculation of hessians.

Deep Q-learning can cope with complex environments, but usually encounter problems of convergence and training time. The implementation of the experience replay and the target network have improved the performance of a deep Q-learning agent in the CartPole environment. Further improvements to the implemented DQN algorithm that could be done are Prioritized Experience Replay (PER) [11], deep recurrent Q-network (DRQN) [12] and noisy networks for exploration [13].

APPENDIX

The Taylor expansion of the perturbation function:

$$Q(\delta\mathbf{x},\delta\mathbf{u}) = \begin{bmatrix} 1 \\ \delta\mathbf{x} \\ \delta\mathbf{u}) \end{bmatrix}^{\mathsf{T}} \begin{bmatrix} 0 & Q_{\mathbf{x}}^{\mathsf{T}} & Q_{\mathbf{u}}^{\mathsf{T}} \\ Q_{\mathbf{x}} & Q_{\mathbf{xx}} & Q_{\mathbf{xu}} \\ Q_{\mathbf{u}} & Q_{\mathbf{ux}} & Q_{\mathbf{vv}} \end{bmatrix} \begin{bmatrix} 1 \\ \delta\mathbf{x} \\ \delta\mathbf{u}) \end{bmatrix} \tag{16}$$

where(with regularization):

$$Q_{\mathbf{x}} = l_x + \mathbf{f}_{\mathbf{x}}^{\mathsf{T}} V_{\mathbf{x}} \tag{17}$$
$$Q_{\mathbf{u}} = l_u + \mathbf{f}_{\mathbf{u}}^{\mathsf{T}} V_{\mathbf{x}} \tag{18}$$
$$Q_{\mathbf{xx}} = l_{xx} + \mathbf{f}_{\mathbf{x}}^{\mathsf{T}} V_{\mathbf{xx}} \mathbf{f}_{\mathbf{x}} \tag{19}$$
$$Q_{\mathbf{ux}} = l_{ux} + \mathbf{f}_{\mathbf{u}}^{\mathsf{T}} (V_{\mathbf{xx}} + \mu\mathbf{I}) \mathbf{f}_{\mathbf{u}} \tag{20}$$
$$Q_{\mathbf{uu}} = l_{uu} + \mathbf{f}_{\mathbf{u}}^{\mathsf{T}} (V_{\mathbf{xx}} + \mu\mathbf{I}) \mathbf{f}_{\mathbf{u}} \tag{21}$$

in each time step, the function V is as follows:

$$V_{\mathbf{x}} = Q_{\mathbf{x}} + \mathsf{K}^{\mathbf{T}} Q_{\mathbf{uu}} \mathbf{k} + \mathsf{K}^{\mathbf{T}} Q_{\mathbf{u}} Q_{\mathbf{ux}}^{\mathbf{T}} \mathbf{k} \tag{22}$$
$$V_{\mathbf{xx}} = Q_{\mathbf{xx}} + \mathsf{K}^{\mathbf{T}} Q_{\mathbf{uu}} \mathbf{k} + \mathsf{K}^{\mathbf{T}} Q_{\mathbf{ux}} Q_{\mathbf{ux}}^{\mathbf{T}} \mathbf{k} \tag{23}$$

regularization:

$$\Delta_{i+1} = min(1.0, \Delta_i/2 \tag{24}$$
$$\mu_{i+1} = \Delta_{i+1} \cdot \mu_i \tag{25}$$

REFERENCES

[1] M. P. Deisenroth, *Efficient Reinforcement Learning using Gaussian Processes*, ser. Karlsruhe Series on Intelligent Sensor-Actuator-Systems, U. D. Hanebeck, Ed. KIT Scientific Publishing, 2010, vol. 9, ISBN: 978-3-86644-569-7.

[2] Z. Cheng, J. Ma, X. Zhang, F. L. Lewis, and T. H. Lee, "Neural network ilqr: A new reinforcement learning architecture," 2020.

[3] Y. Tassa, T. Erez, and E. Todorov, "Synthesis and stabilization of complex behaviors through online trajectory optimization," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 4906–4913.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb 2015. [Online]. Available: https://doi.org/10.1038/nature14236

[6] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," 2017.

[7] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.

[8] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," 2016.

[9] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12. MIT Press, 2000.

[10] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, E. P. Xing and T. Jebara, Eds., vol. 32, no. 1. Bejing, China: PMLR, 22–24 Jun 2014, pp. 387–395. [Online]. Available: http://proceedings.mlr.press/v32/silver14.html

[11] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016.

[12] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," 2017.

[13] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, "Noisy networks for exploration," 2019.