UNIT IV

Ashok Kumar Yadav

Apr 2018

1 Reinforcement Learning and Control

We now begin our study of reinforcement learning and adaptive control. In supervised learning, we saw algorithms that tried to make their outputs mimic the labels y given in the training set. In that setting, the labels gave an unambiguous "right answer" for each of the inputs x. In contrast, for many sequential decision making and control problems, it is very difficult to provide this type of explicit supervision to a learning algorithm. For example, if we have just built a four-legged robot and are trying to program it to walk, then initially we have no idea what the "correct" actions to take are to make it walk, and so do not know how to provide explicit supervision for a learning algorithm to try to mimic.

In the reinforcement learning framework, we will instead provide our algorithms only a reward function, which indicates to the learning agent when it is doing well, and when it is doing poorly. In the four-legged walking example, the reward function might give the robot positive rewards for moving forwards, and negative rewards for either moving backwards or falling over. It will then be the learning algorithm's job to figure out how to choose actions over time so as to obtain large rewards.

Reinforcement learning has been successful in applications as diverse as autonomous helicopter flight, robot legged locomotion, cell-phone network routing, marketing strategy selection, factory control, and efficient web page indexing. Our study of reinforcement learning will begin with a definition of the Markov decision processes (MDP), which provides the formalism in which RL problems are usually posed.

So here is the RL main idea: An agent must learn how to behave in the environment he is located at in order to optimize the long term reward received from the environment. Some points that make me like RL so much:

- RL framework basically describes a closed loop system. The agent (controller) performs actions (control signal), observes the environment state (system output) and receives a reward that has to be optimized.
- Not necessarily the agent must have previous knowledge about the environment. Those algorithms are known as model-free methods.

- The learning basically occur by trial-and-error. This mimics really well the learning process of a human.
- Also, being an optimization problem, it is possible to obtain nice convergence properties for many algorithms.

1.1 Why to use Reinforcement Learning

- A model of the environment is known, but an analytic solution is not available
- Difficult to solve
- Computation is offline
- Exact knowledge of dynamics is required
- Maximizes Performance
- Sustain Change for a long period of time

2 Markov Decision Processes (MDP)

A Markov decision process is a tuple $(S, A, \{P_{sa}\}, \gamma, R)$, where:

- S is a set of states.
- A is a set of actions.
- P_{sa} are the state transition probabilities. P_{sa} is a distribution over the state space for each state $s \in S$ and action $a \in A$.
- $\gamma \in [0, 1)$ is called the discount factor.
- $R : S \times A \mapsto \mathbb{R}$ is the reward function.

The dynamics of an MDP proceeds as follows:
We start in some state s_0, and get to choose some action $a_0 \in A$ to take in the MDP. As a result of our choice, the state of the MDP randomly transitions to some successor state s_1, drawn according to $s_1 \sim P_{s_0 a_0}$. Then, we get to pick another action a_1. As a result of this action, the state transitions again, now to some $s_2 \sim P_{s_1 a_1}$. We then pick a_2, and so on. Pictorially, we can represent this process as follows:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \dots$$

Upon visiting the sequence of states $s_0, s_1, \dots$ with actions $a_0, a_1, \dots$ our total payoff is given by

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \dots$$

Or, when we are writing rewards as a function of the states only, this becomes

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots$$

For most of our development, we will use the simpler state-rewards $R(s)$, though the generalization to state-action rewards $R(s, a)$ offers no special difficulties. Our goal in reinforcement learning is to choose actions over time so as to maximize the expected value of the total payoff:

$$E[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots]$$

Note that the reward at timestep t is discounted by a factor of γ^t. Thus, to make this expectation large, we would like to accrue positive rewards as soon as possible (and postpone negative rewards as long as possible). In economic applications where $R(.)$ is the amount of money made, γ also has a natural interpretation in terms of the interest rate.

3 Bellman equations

3.1 Reward and Return

As discussed above, RL agents learn to maximize cumulative future reward. The word used to describe cumulative future reward is return and is often denoted with R. We also use a subscript t to give the return from a certain time step. In mathematical notation, it looks like this:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + r_{t+4} + ... = \sum_{k=0}^{\infty} r_{t+k+1}$$

If we let this series go on to infinity, then we might end up with infinite return, which really doesn't make a lot of sense for our definition of the problem. Therefore, this equation only makes sense if we expect the series of rewards to end. Tasks that always terminate are called episodic. Card games are good examples of episodic problems. The episode starts by dealing cards to everyone, and inevitably comes to an end depending on the rules of the particular game. Then, another episode is started with the next round by dealing the cards again.

More common than using future cumulative reward as return is using future cumulative discounted reward:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where $0 < \gamma < 1$. The two benefits of defining return this way is that the return is well defined for infinite series, and that it gives a greater weight to sooner rewards, meaning that we care more about imminent rewards and less about rewards we will receive further in the future. The smaller the value we select for γ the more true this is. This can be seen in the special cases where

we let γ equal 0 or 1. If γ is 1, we arrive back at our first equation where we care about all rewards equally, not matter how far into the future they are. On the other hand, when γ is 0 we care only about the immediate reward, and do not care about any reward after that. This would lead our algorithm to be extremely short-sighted. It would learn to take the action that is best for that moment, but won't take into account the effects that action will have on its future.

3.2 Policies

A policy, written $\pi(s, a)$, describes a way of acting. It is a function that takes in a state and an action and returns the probability of taking that action in that state. Therefore, for a given state, it must be true that $\sum_a \pi(s, a) = 1$. Our policy should describe how to act in each state. Our goal in reinforcement learning is to learn an optimal policy, π^*. An optimal policy is a policy which tells us how to act to maximize return in every state. There is one optimal action to take in each state. Sometimes this is written as $\pi^*(s) = a$, which is a mapping from states to optimal actions in those states.

3.3 Value function

To learn the optimal policy, we make use of value functions. There are two types of value functions that are used in reinforcement learning: the state value function, denoted $V(s)$, and the action value function, denoted $Q(s, a)$.

The state value function describes the value of a state when following a policy. It is the expected return when starting from state s acting according to our policy π:

$$V^\pi(s) = \mathbb{E}_\pi\big[R_t | s_t = s\big] \tag{1}$$

It is important to note that even for the same environment the value function changes depending on the policy. This is because the value of the state changes depending on how you act, since the way that you act in that particular state affects how much reward you expect to see. Also note the importance of the expectation. The reason we use an expectation is that there is some randomness in what happens after you arrive at a state. You may have a stochastic policy, which means we need to combine the results of all the different actions that we take. Also, the transition function can be stochastic, meaning that we may not end up in any state with 100% probability. Remember in the example above: when you select an action, the environment returns the next state. There may be multiple states it could return, even given one action. We will see more of this as we look at the Bellman equations. The expectation takes all of this randomness into account.

The other value function we will use is the action value function. The action value function tells us the value of taking an action in some state when following a certain policy. It is the expected return given the state and action under π:

$$Q^\pi(s, a) = \mathbb{E}_\pi\big[R_t | s_t = s, a_t = a\big] \tag{2}$$

The same notes for the state value function apply to the action value function. The expectation takes into account the randomness in future actions according to the policy, as well as the randomness of the returned state from the environment.

3.4 Bellman equations

Richard Bellman was an American applied mathematician who derived the following equations which allow us to start solving these MDPs. The Bellman equations are ubiquitous in RL and are necessary to understand how RL algorithms work. But before we get into the Bellman equations, we need a little more useful notation. We will define $\mathcal{P}$ and $\mathcal{R}$ as follows:

$$\mathcal{P}^a_{ss'} = Pr(s_{t+1} = s' | s_t = s, a_t = a)$$

$\mathcal{P}$ is the transition probability. If we start at state s and take action a we end up in state s' with probability $\mathcal{P}^a_{ss'}$.

$$\mathcal{R}^a_{ss'} = \mathbb{E}[r_{t+1} | s_t = s, s_{t+1} = s', a_t = a]$$

$\mathcal{R}^a_{ss'}$ is another way of writing the expected (or mean) reward that we receive when starting in state s, taking action a, and moving into state s'.

Finally, with these in hand, we are ready to derive the Bellman equations. We will consider the Bellman equation for the state value function. Using the definition for return, we could rewrite Eq. 1 as follows:

$$V^\pi(s) = \mathbb{E}_\pi\Big[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... | s_t = s\Big] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\Big]$$

If we pull out the first reward from the sum, we can rewrite it like so:

$$V^\pi(s) = \mathbb{E}_\pi\Big[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s\Big]$$

The expectation here describes what we expect the return to be if we continue from state s following policy π. The expectation can be written explicitly by summing over all possible actions and all possible returned states. The next two equations can help us make the next step.

$$\mathbb{E}_\pi[r_{t+1} | s_t = s] = \sum_a \pi(s, a) \sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'}$$

$$\mathbb{E}_\pi\Big[\gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s\Big] = \sum_a \pi(s, a) \sum_{s'} \mathcal{P}^a_{ss'} \gamma \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_{t+1} = s'\Big]$$

By distributing the expectation between these two parts, we can then manipulate our equation into the form:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma \mathbb{E}_{\pi} \Big[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+2} | s_{t+1} = s' \Big] \right]$$

Now, note that Eq. 1 is in the same form as the end of this equation. We can therefore substitute it in, giving us

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \Big[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s') \Big] \tag{3}$$

The Bellman equation for the action value function can be derived in a similar way. Following much the same process as for when we derived the Bellman equation for the state value function, we get this series of equations, starting with Eq. 2

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi} \Big[r_{t+1} + \gamma r_{t+2} + \gamma^{2} r_{t+3} + ... | s_t = s, a_t = a \Big] = \mathbb{E}_{\pi} \Big[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} | s_t = s, a_t = a \Big]$$

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi} \Big[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^{k} r_{t+k+2} | s_t = s, a_t = a \Big]$$

$$Q^{\pi}(s,a) = \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma \mathbb{E}_{\pi} \Big[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+2} | s_{t+1} = s' \Big] \right]$$

$$Q^{\pi}(s,a) = \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma \sum_{a'} \mathbb{E}_{\pi} \Big[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+2} | s_{t+1} = s', a_{t+1} = a' \Big] \right]$$

$$Q^{\pi}(s,a) = \sum_{s'} \mathcal{P}^{a}_{ss'} \Big[\mathcal{R}^{a}_{ss'} + \gamma \sum_{a'} \pi(s',a') Q^{\pi}(s',a') \Big] \tag{4}$$

The importance of the Bellman equations is that they let us express values of states as values of other states. This means that if we know the value of s_{t+1}, we can very easily calculate the value of s_t. This opens a lot of doors for iterative approaches for calculating the value for each state, since if we know the value of the next state, we can know the value of the current state. The most important things to remember here are the numbered equations. Finally, with the Bellman equations in hand, we can start looking at how to calculate optimal policies and code our first reinforcement learning agent.

4 Value iteration and policy iteration

We now describe two efficient algorithms for solving finite-state MDPs. For now, we will consider only MDPs with finite state and action spaces ($|S| < \infty, |A| < \infty$). The first algorithm, value iteration, is as follows:

1. For each state s, initialize $V(s) := 0$.

2. Repeat until convergence

   (a) For every state, update $V(s) := R(s) + max_{a \in A} \gamma \sum_{s'} P_{sa}(s') V(s')$.

This algorithm can be thought of as repeatedly trying to update the estimated value function using Bellman Equations. There are two possible ways of performing the updates in the inner loop of the algorithm. In the first, we can first compute the new values for $V(s)$ for every state s, and then overwrite all the old values with the new values. This is called a synchronous update. In this case, the algorithm can be viewed as implementing a "Bellman backup operator" that takes a current estimate of the value function, and maps it to a new estimate. Alternatively, we can also perform asynchronous updates. Here, we would loop over the states (in some order), updating the values one at a time. Under either synchronous or asynchronous updates, it can be shown that value iteration will cause V to converge to V^*. Having found V^*, we can then use Equation to find the optimal policy. Apart from value iteration, there is a second standard algorithm for finding an optimal policy for an MDP. The policy iteration algorithm proceeds as follows:

1. Initialize π randomly.

2. Repeat until convergence

   (a) Let $V := V^\pi$.

   (b) For each state s, let $\pi(s) := argmax_{a \in A} \sum_{s'} P_{sa}(s') V(s')$.

Thus, the inner-loop repeatedly computes the value function for the current policy, and then updates the policy using the current value function. (The policy π found in step (b) is also called the policy that is greedy with respect to V). Note that step (a) can be done via solving Bellman's equations as described earlier, which in the case of a fixed policy, is just a set of $|S|$ linear equations in $|S|$ variables. After at most a finite number of iterations of this algorithm, V will converge to V^*, and π will converge to π^*.

Both value iteration and policy iteration are standard algorithms for solving MDPs, and there isn't currently universal agreement over which algorithm is better. For small MDPs, policy iteration is often very fast and converges with very few iterations. However, for MDPs with large state spaces, solving for V^π explicitly would involve solving a large system of linear equations, and could be difficult. In these problems, value iteration may be preferred. For this reason, in practice value iteration seems to be used more often than policy iteration.

5 Linear quadratic regulator (LQR)

This model is widely used in robotics, and a common technique in many problems is to reduce the formulation to this framework. First, let's describe the model's assumptions. We place ourselves in the continuous setting, with

$$\mathcal{S} = \mathbb{R}^n, \ \mathcal{A} = \mathbb{R}^d$$

and we'll assume linear transitions (with noise)

$$s_{t+1} = A_t s_t + B_t a_t + w_t$$

where $A_r \in R^{n \times n}, B_t \in R^{n \times d}$ are matrices and $w_t \sim \mathcal{N}(0, \sum_t)$ is some gaussian noise (with zero mean). As we'll show in the following paragraphs, it turns out that the noise, as long as it has zero mean, does not impact the optimal policy. We'll also assume quadratic rewards

$$R^{(t)}(s_t, a_t) = -s_t^T U_t s_t - a_t^T W_t a_t$$

where $U_t \in R^{n \times n}, W_t \in R^{d \times d}$ are positive definite matrices (meaning that the reward is always negative).
The 2 steps of the LQR algorithm

1. Suppose that we don't know the matrices $A, B, \sum$. To estimate them, we can follow the ideas outlined in the Value Approximation section of the RL notes. First, collect transitions from an arbitrary policy. Then, use linear regression to find $\underset{A,B}{argmin} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \parallel s_{t+1}^{(i)} - \left(A s_t^{(i)} + B a_t(i) \right) \parallel^2$. Finally, use a technique seen in Gaussian Discriminant Analysis to learn $\sum$.

2. Assuming that the parameters of our model are known (given or estimated with step 1), we can derive the optimal policy using dynamic programming. In other words, given

$$\begin{cases} s_{t+1} & = A_t s_t + B_t a_t + w_t \ A_t, B_t, U_t, W_t, \sum_t \text{ known} \\ R^{(t)}(s_t, a_t) & = -s_t^T U_t s_t - a_t^T W_t a_t \end{cases}$$

   we want to compute V_t^*. we can apply dynamic programming, which yields

1. **Initialization step**
   For the last time step T,

$$\begin{aligned} V_T^* &= \max_{a_T \in \mathcal{A}} R_T(s_T, a_T) \\ &= \max_{a_T \in \mathcal{A}} - s_T^T U_T s_T - a_T^T W_t a_t \\ &= -s_T^T U_T s_T \text{ (maximized for } a_T = 0) \end{aligned}$$

2. **Recurrence step**
   Let $t < T$. Suppose we know V_{t+1}^*.

6 Linear Quadratic Gaussian (LQG)

In control theory, the linear–quadratic–Gaussian (LQG) control problem is one of the most fundamental optimal control problems. It concerns linear systems

driven by additive white Gaussian noise. The problem is to determine an output feedback law that is optimal in the sense of minimizing the expected value of a quadratic cost criterion. Output measurements are assumed to be corrupted by Gaussian noise and the initial state, likewise, is assumed to be a Gaussian random vector. Under these assumptions an optimal control scheme within the class of linear control laws can be derived by a completion-of-squares argument. This control law which is known as the LQG controller, is unique and it is simply a combination of a Kalman filter (a linear–quadratic state estimator (LQE)) together with a linear–quadratic regulator (LQR). The separation principle states that the state estimator and the state feedback can be designed independently. LQG control applies to both linear time-invariant systems as well as linear time-varying systems, and constitutes a linear dynamic feedback control law that is easily computed and implemented: the LQG controller itself is a dynamic system like the system it controls. Both systems have the same state dimension.

A deeper statement of the separation principle is that the LQG controller is still optimal in a wider class of possibly nonlinear controllers. That is, utilizing a nonlinear control scheme will not improve the expected value of the cost functional. This version of the separation principle is a special case of the separation principle of stochastic control which states that even when the process and output noise sources are possibly non-Gaussian martingales, as long as the system dynamics are linear, the optimal control separates into an optimal state estimator (which may no longer be a Kalman filter) and an LQR regulator. In the classical LQG setting, implementation of the LQG controller may be problematic when the dimension of the system state is large. The reduced-order LQG problem (fixed-order LQG problem) overcomes this by fixing a priori the number of states of the LQG controller. This problem is more difficult to solve because it is no longer separable. Also, the solution is no longer unique. Despite these facts numerical algorithms are available to solve the associated optimal projection equations which constitute necessary and sufficient conditions for a locally optimal reduced-order LQG controller.

LQG optimality does not automatically ensure good robustness properties. The robust stability of the closed loop system must be checked separately after the LQG controller has been designed. To promote robustness some of the system parameters may be assumed stochastic instead of deterministic. The associated more difficult control problem leads to a similar optimal controller of which only the controller parameters are different. Finally, the LQG controller is also used to control perturbed non-linear systems

6.1 Description of the problem

Continuous time
Consider the continuous-time linear dynamic system

$$\dot{\mathbf{x}}(t) = A(t)\mathbf{x}(t) + B(t)\mathbf{u}(t) + \mathbf{v}(t)$$

,

$$\mathbf{y}(t) = C(t)\mathbf{x}(t) + \mathbf{w}(t),$$

where $\mathbf{x}$ represents the vector of state variables of the system, $\mathbf{u}$ the vector of control inputs and $\mathbf{y}$ the vector of measured outputs available for feedback. Both additive white Gaussian system noise $\mathbf{v}(t)$ and additive white Gaussian measurement noise $\mathbf{w}(t)$ affect the system. Given this system the objective is to find the control input history $\mathbf{u}(t)$ which at every time $\mathbf{t}$ may depend linearly only on the past measurements $\mathbf{y}(t'), 0 \leq t' < t$ such that the following cost function is minimized:

$$J = \mathbb{E}\left[\mathbf{x}^{\mathrm{T}}(T)F\mathbf{x}(T) + \int_0^T \mathbf{x}^{\mathrm{T}}(t)Q(t)\mathbf{x}(t) + \mathbf{u}^{\mathrm{T}}(t)R(t)\mathbf{u}(t)\,dt\right]$$

$$F \geq 0, \quad Q(t) \geq 0, \quad R(t) > 0,$$

where $\mathbb{E}$ denotes the expected value. The final time (horizon) $\mathbf{T}$ may be either finite or infinite. If the horizon tends to infinity the first term $\mathbf{x}^{\mathrm{T}}(T)F\mathbf{x}(T)$ of the cost function becomes negligible and irrelevant to the problem. Also to keep the costs finite the cost function has to be taken to be J/T.

The LQG controller that solves the LQG control problem is specified by the following equations:

$$\dot{\hat{\mathbf{x}}}(t) = A(t)\hat{\mathbf{x}}(t) + B(t)\mathbf{u}(t) + L(t)\left(\mathbf{y}(t) - C(t)\hat{\mathbf{x}}(t)\right), \quad \hat{\mathbf{x}}(0) = \mathbb{E}\left[\mathbf{x}(0)\right],$$

$$\mathbf{u}(t) = -K(t)\hat{\mathbf{x}}(t).$$

The matrix $L(t)$ is called the Kalman gain of the associated Kalman filter represented by the first equation. At each time t this filter generates estimates $\hat{\mathbf{x}}(t)$ of the state $\mathbf{x}(t)$ using the past measurements and inputs. The Kalman gain $L(t)$ is computed from the matrices $A(t), C(t)$, the two intensity matrices $V(t), W(t)$ associated to the white Gaussian noises $\mathbf{v}(t)$ and $\mathbf{w}(t)$ and finally $\mathbb{E}\left[\mathbf{x}(0)\mathbf{x}^{\mathrm{T}}(0)\right]$. These five matrices determine the Kalman gain through the following associated matrix Riccati differential equation:

$$\dot{P}(t) = A(t)P(t) + P(t)A^{\mathrm{T}}(t) - P(t)C^{\mathrm{T}}(t)W^{-1}(t)C(t)P(t) + V(t),$$

$$P(0) = \mathbb{E}\left[\mathbf{x}(0)\mathbf{x}^{\mathrm{T}}(0)\right].$$

Given the solution $P(t), 0 \leq t \leq T$ the Kalman gain equals

$$L(t) = P(t)C^{\mathrm{T}}(t)W^{-1}(t).$$

The matrix $K(t)$ is called the feedback gain matrix. This matrix is determined by the matrices $A(t), B(t), Q(t), R(t)$ and F through the following associated matrix Riccati differential equation:

$$-\dot{S}(t) = A^{\mathrm{T}}(t)S(t) + S(t)A(t) - S(t)B(t)R^{-1}(t)B^{\mathrm{T}}(t)S(t) + Q(t),$$

$$S(T) = F.$$

Given the solution $S(t), 0 \leq t \leq T$ the feedback gain equals

$$K(t) = R^{-1}(t)B^{\mathrm{T}}(t)S(t).$$

Observe the similarity of the two matrix Riccati differential equations, the first one running forward in time, the second one running backward in time. This similarity is called duality. The first matrix Riccati differential equation solves the linear–quadratic estimation problem (LQE). The second matrix Riccati differential equation solves the linear–quadratic regulator problem (LQR). These problems are dual and together they solve the linear–quadratic–Gaussian control problem (LQG). So the LQG problem separates into the LQE and LQR problem that can be solved independently. Therefore, the LQG problem is called separable.

When $A(t), B(t), C(t), Q(t), R(t)$ and the noise intensity matrices $V(t), W(t)W(t)$ do not depend on t and when T tends to infinity the LQG controller becomes a time-invariant dynamic system. In that case both matrix Riccati differential equations may be replaced by the two associated algebraic Riccati equations.

Discrete time

Since the discrete-time LQG control problem is similar to the one in continuous-time, the description below focuses on the mathematical equations. The discrete-time linear system equations are

$$\mathbf{x}_{i+1} = A_i\mathbf{x}_i + B_i\mathbf{u}_i + \mathbf{v}_i,$$

$$\mathbf{y}_i = C_i\mathbf{x}_i + \mathbf{w}_i.$$

Here i represents the discrete time index and $\mathbf{v}_i, \mathbf{w}_i$ represent discrete-time Gaussian white noise processes with covariance matrices V_i, W_i respectively. The quadratic cost function to be minimized is

$$J = \mathbb{E}\left[\mathbf{x}_N^{\mathrm{T}} F \mathbf{x}_N + \sum_{i=0}^{N-1} (\mathbf{x}_i^{\mathrm{T}} Q_i \mathbf{x}_i + \mathbf{u}_i^{\mathrm{T}} R_i \mathbf{u}_i)\right],$$

$$F \geq 0, Q_i \geq 0, R_i > 0.$$

The discrete-time LQG controller is

$$\hat{\mathbf{x}}_{i+1} = A_i\hat{\mathbf{x}}_i + B_i\mathbf{u}_i + L_{i+1}\left(\mathbf{y}_{i+1} - C_{i+1}\left\{A_i\hat{\mathbf{x}}_i + B_i u_i\right\}\right), \qquad \hat{\mathbf{x}}_0 = \mathbb{E}[\mathbf{x}_0],$$

$$\mathbf{u}_i = -K_i\hat{\mathbf{x}}_i.$$

The Kalman gain equals

$$L_i = P_i C_i^{\mathrm{T}} (C_i P_i C_i^{\mathrm{T}} + W_i)^{-1},$$

where P_i is determined by the following matrix Riccati difference equation that runs forward in time:

$$P_{i+1} = A_i\left(P_i - P_i C_i^{\mathrm{T}}\left(C_i P_i C_i^{\mathrm{T}} + W_i\right)^{-1} C_i P_i\right) A_i^{\mathrm{T}} + V_i, \qquad P_0 = \mathbb{E}[(\mathbf{x}_0 - \hat{\mathbf{x}}_0)(\mathbf{x}_0 - \hat{\mathbf{x}}_0)^{\mathrm{T}}].$$

The feedback gain matrix equals

$$K_i = (B_i^{\mathrm{T}} S_{i+1} B_i + R_i)^{-1} B_i^{\mathrm{T}} S_{i+1} A_i$$

where S_i is determined by the following matrix Riccati difference equation that runs backward in time:

$$S_i = A_i^{\mathrm{T}} \left(S_{i+1} - S_{i+1} B_i \left(B_i^{\mathrm{T}} S_{i+1} B_i + R_i \right)^{-1} B_i^{\mathrm{T}} S_{i+1} \right) A_i + Q_i, \quad S_N = F.$$

If all the matrices in the problem formulation are time-invariant and if the horizon N tends to infinity the discrete-time LQG controller becomes time-invariant. In that case the matrix Riccati difference equations may be replaced by their associated discrete-time algebraic Riccati equations. These determine the time-invariant linear–quadratic estimator and the time-invariant linear–quadratic regulator in discrete-time. To keep the costs finite instead of J one has to consider J/N in this case.

7 Q-learning

Now, lets consider the case where the agent does not know apriori what are the effects of its actions on the environment (state transition and reward models are not known). The agent only knows what are the set of possible states and actions, and can observe the environment current state. In this case, the agent has to actively learn through the experience of interactions with the environment. There are two categories of learning algorithms:

model-based learning:
In model-based learning, the agent will interact to the environment and from the history of its interactions, the agent will try to approximate the environment state transition and reward models. Afterwards, given the models it learnt, the agent can use value-iteration or policy-iteration to find an optimal policy.

model-free learning:
In model-free learning, the agent will not try to learn explicit models of the environment state transition and reward functions. However, it directly derives an optimal policy from the interactions with the environment.

Q-Learning is an example of model-free learning algorithm. It does not assume that agent knows anything about the state-transition and reward models. However, the agent will discover what are the good and bad actions by trial and error.

The basic idea of Q-Learning is to approximate the state-action pairs Q-function from the samples of $Q(s, a)$ that we observe during interaction with the environment. This approach is known as Time-Difference Learning.

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha Q_{obs}(s,a)$$
$$\text{where } Q_{obs}(s,a) = r(s,a) + \gamma \max_{a'} Q(s', a')$$

where α is the learning rate. The $Q(s,a)$ table is initialized randomly. Then the agent starts to interact with the environment, and upon each interaction the agent will observe the reward of its action $r(s,a)$ and the state transition (new state s'). The agent compute the observed Q-value $Q_{obs}(s,a)$ and then use the above equation to update its own estimate of $Q(s,a)$.

An important question is how does the agent select actions during learning. Should the agent trust the learnt values of $Q(s,a)$ enough to select actions based on it ? or try other actions hoping this may give it a better reward. This is known as the exploration vs exploitation dilemma.

A simple approach is known as the ϵ-greedy approach where at each step. With small probability ϵ, the agent will pick a random action (explore) or with probability $(1-\epsilon)$ the agent will select an action according to the current estimate of Q-values. ϵ value can be decreased overtime as the agent becomes more confident with its estimate of Q-values.

8 Value function approximation

We now describe an alternative method for finding policies in continuous state MDPs, in which we approximate V^* directly, without resorting to discretization. This approach, caled value function approximation, has been successfully applied to many RL problems.

9 Policy search

An alternative method is to search directly in (some subset of) the policy space, in which case the problem becomes a case of stochastic optimization. The two approaches available are gradient-based and gradient-free methods. Gradient-based methods (policy gradient methods) start with a mapping from a finite-dimensional (parameter) space to the space of policies: given the parameter vector θ, let π denote the policy associated to θ. Defining the performance function by

$$\rho(\theta) = \rho^{\pi_\theta}$$

under mild conditions this function will be differentiable as a function of the parameter vector θ. If the gradient of ρ was known, one could use gradient ascent. Since an analytic expression for the gradient is not available, only a noisy estimate is available. Such an estimate can be constructed in many ways, giving rise to algorithms such as Williams' Reinforce method (which is known as the likelihood ratio method in the simulation-based optimization literature). Policy search methods have been used in the robotics context. Many policy search methods may get stuck in local optima (as they are based on local search).

A large class of methods avoids relying on gradient information. These include simulated annealing, cross-entropy search or methods of evolutionary computation. Many gradient-free methods can achieve (in theory and in the limit) a global optimum.

Policy search methods may converge slowly given noisy data. For example, this happens in episodic problems when the trajectories are long and the variance of the returns is large. Value-function based methods that rely on temporal differences might help in this case. In recent years, actor–critic methods have been proposed and performed well on various problems

10 Partially observable Markov decision process (POMDP)

A partially observable Markov decision process (POMDP) is a generalization of a Markov decision process (MDP). A POMDP models an agent decision process in which it is assumed that the system dynamics are determined by an MDP, but the agent cannot directly observe the underlying state. Instead, it must maintain a probability distribution over the set of possible states, based on a set of observations and observation probabilities, and the underlying MDP.

The POMDP framework is general enough to model a variety of real-world sequential decision processes. Applications include robot navigation problems, machine maintenance, and planning under uncertainty in general. The general framework of Markov decision processes with incomplete information was described by Karl Johan Åström in 1965 in the case of a discrete state space, and it was further studied in the operations research community where the acronym POMDP was coined. It was later adapted for problems in artificial intelligence and automated planning by Leslie P. Kaelbling and Michael L. Littman.

An exact solution to a POMDP yields the optimal action for each possible belief over the world states. The optimal action maximizes (or minimizes) the expected reward (or cost) of the agent over a possibly infinite horizon. The sequence of optimal actions is known as the optimal policy of the agent for interacting with its environment.

Definition

A discrete-time POMDP models the relationship between an agent and its environment. Formally, a POMDP is a 7-tuple $(S, A, T, R, \Omega, O, \gamma)$, where

- S is a set of states,
- A is a set of actions,
- T is a set of conditional transition probabilities between states,
- $R : S \times A \to \mathbb{R}$ is the reward function.
- Ω is a set of observations,
- O is a set of conditional observation probabilities, and
- $\gamma \in [0, 1]$ is the discount factor.

At each time period, the environment is in some state $s \in S$. The agent takes an action $a \in A$, which causes the environment to transition to state s' with probability $T(s' \mid s, a)$. At the same time, the agent receives an observation $o \in \Omega$ which depends on the new state of the environment, s', and on the just taken action, a, with probability $O(o \mid s', a)$. Finally, the agent receives a reward r equal to $R(s, a)$. Then the process repeats. The goal is for the agent to choose actions at each time step that maximize its expected future discounted reward: $E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$, where r_t is the reward earned at time t. The discount factor γ determines how much immediate rewards are favoured over more distant rewards. When $\gamma = 0$ the agent only cares about which action will yield the largest expected immediate reward; when $\gamma = 1$ the agent cares about maximizing the expected sum of future rewards.

Because the agent does not directly observe the environment's state, the agent must make decisions under uncertainty of the true environment state. However, by interacting with the environment and receiving observations, the agent may update its belief in the true state by updating the probability distribution of the current state. A consequence of this property is that the optimal behaviour may often include (information gathering) actions that are taken purely because they improve the agent's estimate of the current state, thereby allowing it to make better decisions in the future.

It is instructive to compare the above definition with the definition of a Markov decision process. An MDP does not include the observation set, because the agent always knows with certainty the environment's current state. Alternatively, an MDP can be reformulated as a POMDP by setting the observation set to be equal to the set of states and defining the observation conditional probabilities to deterministically select the observation that corresponds to the true state.

After having taken the action a and observing o, an agent needs to update its belief in the state the environment may (or not) be in. Since the state is Markovian (by assumption), maintaining a belief over the states solely requires knowledge of the previous belief state, the action taken, and the current observation. The operation is denoted $b' = \tau(b, a, o)$. Below we describe how this belief update is computed.

After reaching s', the agent observes $o \in \Omega$ with probability $O(o \mid s', a)$. Let b be a probability distribution over the state space S. $b(s)$ denotes the probability that the environment is in state s. Given $b(s)$, then after taking action a and observing o,

$$b'(s') = \eta O(o \mid s', a) \sum_{s \in S} T(s' \mid s, a) b(s)$$

where $\eta = 1/\Pr(o \mid b, a)$ is a normalizing constant with $\Pr(o \mid b, a) = \sum_{s' \in S} O(o \mid s', a) \sum_{s \in S} T(s' \mid s, a) b(s)$.