ADVANCED MACHINE LEARNING

Reinforcement Learning
Continuous
Drawbacks of standard RL

Curse of dimensionality:
Computational costs increase dramatically with number of states.

Even if the problem is discrete by nature e.g.
- Play Backgammon
- Make train schedule
- Determine slots for TV programs

The number of states may be huge (10^{20} for Backgammon) and may explode the memory of the system.
Drawbacks of standard RL

Markov World:

For most real-world problems, the state and/or actions are not discrete by nature.

- Robotics: control motion of a robot (motion and state of joints are continuous)
- Finances: values of stocks in continuous, actions (buy or not) may be discrete.

→ Discretizing is possible, but deciding on the granularity of the discretization is difficult and will impact the precision of the control.

→ Need a method to learn from continuous representations.
Drawbacks of standard RL

Model-based vs model free:

DP requires a model of the world (can be estimated through exploration).

- In discrete RL, one uses an iterative technique (TD / Sarsa) to obtain an estimate of the Value function or Q-value function → No longer optimal estimate

- In continuous RL, one extends this iterative approach to estimate the Value function, the Q-value function or the policy by exploiting function approximation techniques (similar to non-linear regression).

→ simplest techniques are gradient methods
**RL in continuous state and action spaces**

States $s_t$ and actions $a_t$, $t = 1, \ldots, T$, are continuous:

$$s_t \in \mathbb{R}^N, a_t \in \mathbb{R}^P$$

One can no longer swipe through all states and actions to determine the optimal policy.

Instead, one can either:

1) use function approximation to estimate the value function $V(s)$
2) use function approximation to estimate the state-action value function $Q(s, a)$
3) optimize a parameterized policy $\pi(a | s)$ (policy search)
Parame... by function approximation

Parametrize the value function:

\[ V(s; \alpha) \]

Open parameters

Parametrize the value function such that:

\[ V(s; \alpha) = \sum_{j=1}^{K} \alpha_j \phi_j(s) = \alpha^T \phi(s) \]

\( \phi_j(s) \) form a set of \( K \) basis functions (e.g. RBF functions).
These are set by the user and also called the features.

\( \alpha_j \in \mathbb{R} \) are the weights associated to each feature.
These are the unknown parameters.
Learning the value function

How to update the value function?

In Monte-Carlo, the target is the expected return
$$\Delta \alpha = \eta \left( R_t - V(s_t; \alpha) \right) \Delta \alpha V(s_t; \alpha), \quad \eta \in \mathbb{R}_+: \text{learning rate}$$

In TD, the target is $$r_{t+1} + \gamma V(s_{t+1}; \alpha)$$
$$\Delta \alpha = \eta \left( r_{t+1} + \gamma V(s_{t+1}; \alpha) - V(s_t; \alpha) \right) \Delta \alpha V(s_t; \alpha)$$

Do roll-outs, measure sets of rewards for each visited state $$\{s_t; r_t\}_{t=1}^T$$ and update the value function using the above equations.

One can use other techniques, e.g. ML techniques for non-linear regression, to get a better estimate of the parameters than simple gradient descent, see Deisenroth et al, Foundations & Trends in Robotics, 2011; Peters & Scahaal, Neural Networks, 2008 for surveys.
RL by function approximation: example

Value function:

\[ V(s; \alpha) = \sum_{j=1}^{K} \alpha_j \phi_j(s) \]

Example:

\( s \): cartesian state of robot's gripper

\( \phi_j(s) \): features in the state, e.g.
- distances to the holes
- distances to the walls

Choosing the features is not trivial and is key to success.
RL by function approximation: example

Value function:

\[ V(s; \alpha) = \sum_{j=1}^{K} \alpha_j \phi_j(s) \]

Example:

- \( s \): cartesian state of robot's gripper
- \( \phi_j(s) \): features in the state, e.g.
  - distances to the holes
  - distances to the walls

What would be a good set of weights for the robot to learn how to sink the ball to any of the holes?
What would the value function look like?
Learning the value function

Value function:

\[ V(s; \alpha) = \alpha_1 \phi_1(s) + \alpha_2 \phi_2(s) \]

\[ \phi_1(s) = 1 \text{ if hit a wall, 0 otherwise} \]

\[ \phi_2(s) = 1 \text{ if sunk into a hole, 0 otherwise} \]

Start with initial estimate:

\[ \alpha_1 = \alpha_2 = 0.5 \]

Perform one roll-out with greedy policy

Gather reward: -10 (hit a wall)

Update value function by gradient descent (TD):

\[ \Delta \alpha = (r_{t+1} + \gamma \alpha^T \phi(s_{t+1}) - \alpha^T \phi(s_t)) \Delta_\alpha \left( \alpha^T \phi(s_{t+1}) \right) \]

The update is influenced immediately by the active feature.
From value function to policy.

When the actions are simply the derivative of the state \( a = \dot{s} \) e.g. motion of robot in 2D space, the greedy policy can be derived by taking the gradient of the value function:

\[
\pi \left( a = \dot{s} \mid s \right) \sim \Delta V(s)
\]

\[ a_t = \dot{s}_t = \eta \Delta V(s_t), \quad \eta: \text{scaling factor} \]
From value function to policy.

When the actions are simply the derivative of the state \((a = \dot{s})\) e.g. motion of robot in 2D space, the greedy policy can be derived by taking the gradient of the value function:

\[
\pi(a = \dot{s} | s) \sim \Delta V(s)
\]

Not possible when the actions differ from the state space, e.g. underactuated robot.

**Actor-Critic:**

Critic: Update the parameters of the value function (or action-value function) using, e.g. TD learning.

Actor: Updates policy parameters in direction suggested by critic using gradient descent.
Robotics Applications of continuous RL

Teaching a two joints, three links robot leg to stand up

Robot state: $\theta_0$: pitch angle, $\theta_1$: hip joint angle, $\theta_2$: knee joint angle, $\theta_m$: the angle of the line from the center of mass to the center of the foot.

Robot actions: torques to actuate the two joints.

Morimoto and Doya, Robotics and Autonomous Systems , 2001
GOAL: The stand-up task is accomplished when the robot stands up and stays upright for more than $2(T + 1)$ seconds.

SUBGOALS: Reaching the final goal is a necessary but not sufficient condition of successful stand-up because the robot may fall down after passing through the final goal $\rightarrow$ need to define subgoals.
Robotics Applications of continuous RL

Decompose the task into an upper-level and lower-level sets of goals.

In upper level, perform coarse discretization of state-action space and use Q-learning.

Reward at upper level

\[ R(T) = R_{\text{main}} + R_{\text{sub}}. \]

\[ R_{\text{main}} = \begin{cases} 1, & \text{on success of stand-up,} \\ 0, & \text{on failure,} \end{cases} \]

\[ R_{\text{sub}} = \begin{cases} 1, & \text{final goal achieved,} \\ 0.25 \left( \frac{Y}{L} + 1 \right), & \text{sub-goal achieved,} \\ 0, & \text{on failure,} \end{cases} \]

Y: height of the head of the robot at a sub-goal posture and \( L \) is total length of the robot.

When the robot achieves a sub-goal, the learner gets a reward < 0.5. Full reward is obtained when all subgoals and main goal are both achieved.

Morimoto and Doya, Robotics and Autonomous Systems, 2001
Robotics Applications of continuous RL

The lower level learns how to achieve each subgoal:
The reward $r$ is the level of achievement of the subgoal. States and actions are continuous in time: $s(t), u(t)$

Model value function: $V(s; \alpha) = \alpha^T \phi(s)$

Model policy: $\pi(u|s; \beta)$

$\Rightarrow u(s) = f(\beta^T \phi(s) + \varepsilon)$,

$f$: truncated sigmoid function, $\varepsilon$: noise

Critic: Learn $\alpha$ through gradient descent on squared TD error.

Actor: Use TD error $e(t)$ to estimate $\beta$:

$\Delta \beta_j(t) \sim e(t)\phi_j, \quad j = 1...K$
Robotics Applications of continuous RL

• 750 trials in simulation + 170 on real robot
• Goal: to stand up
RL in continuous state and action spaces

States $s_t$ and actions $a_t$, $t = 1, \ldots, T$, are continuous:

$$s_t \in \mathbb{R}^N, a_t \in \mathbb{R}^P$$

One can no longer swipe through all states and actions to determine the optimal policy.

Instead, one can either:

1) use function approximation to estimate the value function $V(s)$
2) use function approximation to estimate the state-action value function $Q(s, a)$
3) optimize a parameterized policy $\pi(a | s)$ (policy search)
Parametrize the Q-value function

Approximate the state-value function:

\[ Q(s, a; \alpha) = \sum_{j=1}^{K} \alpha_j \phi_j(s, a) = \alpha^T \phi(s, a) \]

\( \phi(s, a) \) is a set of \( K \) basis function \( \phi_j(s, a) : \mathbb{R}^N \times \mathbb{R}^P \longrightarrow \mathbb{R} \). These are set by the user and also called the features.

\( \alpha_j \in \mathbb{R} \) are the weights associated to each feature. These are the unknown parameters.
Update on Bellman residual error

Recall (see slides on discrete RL) the update step on Q-learning:
\[ Q(s_t, a_t; \alpha) = r_t + \gamma Q(s_{t+1}, a_{t+1}; \alpha) \]

The Bellman residual error is given by:
\[ L(\alpha) = \sum_t \| Q(s_t, a_t; \alpha) - r_t - \gamma Q(s_{t+1}, a_{t+1}; \alpha) \|^2 \]

Use the Bellman residual error to determine the parameters of the Q-value function iteratively, see next slide.
Update on Bellman residual error

Perform a roll-out for $T$ time steps using policy $\pi(a \mid s; \alpha)$ (greedy search on arg max $Q(s, a; \alpha)$ using current estimate of $Q(s, a; \alpha)$).

Collect samples $\{s_{1:T+1}, a_{1:T}, r_{1:T} = r(s_{2:T+1})\}$.

Determine how good an initial choice of parameter $\alpha$ is by comparing the predicted reward and the actual reward $R = \{r_t\}_{t=1}^T$.

$$L(\alpha) = \left\| R - (\phi - \gamma \phi')^T \alpha \right\|,$$

Bellman Residual Error

$\phi = \{\phi(s_t, a_t)\}_{t=1}^{T}, \phi' = \{\phi(s_{t+1}, \pi(s_t, a_t))\}_{t=1}^{T}, \gamma \in [0,1]$ discount factor

Solution is found by least-square on the objective function.

Example learning to ride a bike

Goal: learn to balance and to ride a bicycle to a target position located 1 km away from the starting location

Continuous state: six-dimensional real-valued vector
- angle of the handlebar,
- vertical angle of the bicycle
- angle of the bicycle to the goal
  \[ S = (\theta, \dot{\theta}, \omega, \dot{\omega}, \psi) \]

5 discrete actions: torque applied to the handlebar (discretized to \{-2, 0, +2\}) and displacement of the rider (discretized to \{-0.02, 0, +0.02\}).

Model of the world (simulated): uniform noise on each action. Model of the bike’s dynamics.

Example learning to ride a bike

State:
\[ S = (\theta, \dot{\theta}, \omega, \dot{\omega}, \psi) \]

Q-value function parametrized with 20 basis functions:
\[ Q(s,a) = \sum_{i=1}^{20} \alpha_i \phi(s,a) \]

Features: combination of state and its 1st and second derivatives
with \( \phi(s) = \{1, \theta, \dot{\theta}, \theta^2, \dot{\theta}^2, \omega \theta, \omega \theta^2, \omega^2 \theta, \omega, \omega^2, \dot{\omega}, \omega \dot{\omega}, \psi, \psi^2, \psi \theta, \psi, \psi^2, \psi \theta\} \)

and uniform distribution on discrete values of \( a \).

Iterate between using the policy to generae roll-outs by taking greedy approach on current estimate of \( Q \) and update \( Q \) using the Bellman error.
Example learning to ride a bike

Collect training samples using random policy. Each episode lasts 20 steps. Learned policy evaluated 100 times to estimate the probability of success.

The policy after the first iteration balances the bicycle, but fails to ride to the goal.

The policy after the second iteration is heading towards the goal, but fails to balance.

All policies thereafter balance and ride the bicycle to the goal.

Crash still happens because of noise in the model.
Example learning to ride a bike

Successful policies are found after a few thousand training episodes.

With 5000 training episodes (60000 samples) the probability of success is about 95% and the expected number of balancing steps is about 70000 steps.

RL in continuous state and action spaces

States $s_t$ and actions $a_t$, $t = 1,...,T$, are continuous:

$$s_t \in \mathbb{R}^N, a_t \in \mathbb{R}^P$$

One can no longer swipe through all states and actions to determine the optimal policy.

Instead, one can either:

1) use function approximation to estimate the value function $V(s)$
2) use function approximation to estimate the state-action value function $Q(s, a)$
3) optimize a parameterized policy $\pi(a | s)$ (policy search)
Policy Gradients

An alternative is to parametrize the stochastic policy:
\[ \pi(a \mid s) \] is approximated by drawing from the distribution \( p(a \mid s; \alpha) \).

Actor in state \( s_t \), choose an action \( a_t \) following a stochastic policy: \( \pi(a_t \mid s_t) \) drawing from the distribution \( p(a \mid s_t; \alpha) \).
Policy learning by Weighted Exploration with the Returns (PoWER)

Starting from a deterministic estimate of the policy:

$$\pi(a | s; \alpha) = \sum_{j=1}^{K} \alpha_j \phi_j(a | s) = \alpha^T \phi(a | s)$$

where \(\phi_j(a | s)\) are again a set of \(K\) known basis functions and \(\alpha_j\) the weights for each basis function.

Adding some noise and making the policy explicitly time-dependent

$$\pi(a_t | s_t, t; \alpha) = (\alpha^T + \epsilon_t) \phi(a | s_t, t), \quad \epsilon_t \sim N(0, \Sigma): \text{ noise for exploration}$$

⇒ This leads to a stochastic policy:

$$\pi(a_t | s_t, t; \alpha) \sim N(\phi(a | s_t, t)^T \alpha, \phi(a | s_t, t)^T \Sigma \phi(a | s_t, t))$$

Kobler and Peters, Machine Learning, 2011
Policy learning by Weighted Exploration with the Returns (PoWER)

At each roll-out (episode), the agent gathers tuples of rewards and associated states and actions: \( \tau = \{a_{1:T}, s_{1:T+1}, r_{1:T}\} \).

Compute unbiased estimate of the Q-function:

\[
Q_{\pi}(s, a, t; \alpha) = \sum_{t=1}^{T} r(s_{t+1}, s_t, a_t, t)
\]

Update at each time step the parameters \( \alpha \):

\[
\Delta \alpha_t = \left( \sum_{t=1}^{T} W(s, t) Q_{\pi}(s, a, t; \alpha) \right)^{-1} \left( \sum_{t=1}^{T} W(s, t) \varepsilon_t Q_{\pi}(s, a, t; \alpha) \right)
\]

\[
W(s, t) = \phi(a | s, t) \phi(a | s, t)^T \left( \phi(a | s, t)^T \Sigma \phi(a | s, t) \right)^{-1}
\]

until convergence, i.e. \( \Delta \alpha_t \) smaller than a threshold.
Teaching a robot to play the ball-in-a-cup task

State: joint angle + velocities of robot + Cartesian coordinates of the ball

Action: joint space accelerations

\[ \pi(\ddot{\theta} | \theta, \dot{\theta}, x) \sim \alpha^T \phi(\theta, \dot{\theta}, x), \] with 31 basis functions per DOF

Kobler and Peters, Machine Learning, 2011
Policy learning by Weighted Exploration with the Returns (PoWER)

Fig. 10: This figure illustrates how the reward is calculated. The plane represents the level of the upper rim of the cup. For a successful rollout the ball has to be moved above the cup first and is then flying in a downward direction into the opening of the cup. The reward is calculated as the distance $d$ of the center of the cup and the center of the ball on the plane at the moment the ball is passing the plane in a downward direction. If the ball is flying directly into the center of the cup, the distance is 0 and through the transformation $\exp(-d^2)$ yields the highest possible reward of 1. The further the ball passes the plane from the cup, the larger the distance and thus the smaller the resulting reward.

$$r(t) = \begin{cases} \exp\left(-\alpha (x_c - x_b)^2 - \alpha (y_c - y_b)^2\right) & \text{if } t = t_c, \\ 0 & \text{otherwise,} \end{cases}$$

where we denote the moment when the ball passes the rim of the cup with a downward direction by $t_c$, the cup position by $[x_c, y_c, z_c] \in \mathbb{R}^3$, the ball position by $[x_b, y_b, z_b] \in \mathbb{R}^3$, and a scaling parameter by $\alpha = 100$ (see also Figure 10). The algorithm is robust to
Extensions of RL framework

1) A major difficulty in RL is to determine the reward

Inverse Reinforcement Learning (closely related to Inverse Optimal Control for Robotics problems) was proposed as framework to estimate what the reward could be, see:


2) Reinforcement learning always assume to make good observations of the system (positive reward). This is often impractical (no expert to teach the robot).

Approaches to learn from bad examples only, from failure (Donut), see

Summary

• RL was coined first for discrete state and action spaces.

• A RL problem is entirely determined by its states, actions, rewards, state transitions probabilities, probability of reward, state-action transitions (policy). It assumes that all probabilities are first order Markov.

• When the world is known, an optimal solution for the policy can be found using Dynamic Programming (DP).

• Otherwise, iterative techniques are used to approximate the optimal solution (Monte-Carlo, TD, SARSA). This requires to generate roll-outs to span the state-action space. When the policy is used right away during training, one must balance exploration with exploitation.

• Continuous RL problem extend the discrete RL framework. They reuse the notion of Value function, Q-value function and Policy but treat these as continuous functions and approximate their value by gradient descent.