Representations for Reinforcement Learning

Doina Precup
Reasoning & Learning Lab
McGill University

With thanks to Rich Sutton, Satinder Singh, Pierre-Luc Bacon, Jean Harb

MSR Cambridge AI Summer School, July 2017
Reinforcement learning

- Learning by trial-and-error
- Learning is driven by a (numerical) reward signal, which may be delayed
- Goal: maximize a cumulative measure of reward (e.g., discounted sum)
- Draws ideas from animal learning/psychology, control, operations research
A big success story: AlphaGo

The first AI Go player to defeat a human (9 dan) champion

ARTICLE

Mastering the game of Go with deep neural networks and tree search

MSR Cambridge AI Summer School, July 2017
Contrast: Supervised learning

- Training experience: a set of labeled examples of the form
  \[ \langle x_1, x_2, \ldots, x_n, y \rangle, \]
  where \( x_j \) are values for input variables and \( y \) is the output
- This implies the existence of a “teacher” who knows the right answers
- Goal: minimize the prediction error (loss) function
Contrast: Unsupervised learning

- Training experience: unlabelled data (e.g. gene level activity)
- What to learn: interesting associations in the data (often no single correct answer)
- E.g., clustering, dimensionality reduction
- Typical goal: produce a model that maximizes data likelihood
Reinforcement Learning Framework

- At every time step $t$, the agent perceives the state of the environment.
- Based on this perception, it chooses an action.
- The action causes the agent to receive a numerical reward.
- Prediction: Learn the expected cumulated future reward given the current state and current way of behaving.
- Control: Find a way of choosing actions, called a policy which maximizes the agent’s long-term expected return.
Prediction Example: Medical Time Series (Apex Project)

- The states are cardio-respiratory measurements
- Reward is the patient outcome at the end of the procedure (delayed)
- Policy is unknown (hospital practice)
Control Example: Atari Games (Mnih et al, 2015)

- The states are board positions in which the agent can move
- The actions are the possible joystick moves allowed by the game
- Reward is given by the points achieved in the game
Key Features of RL Control

- The learner is not told what actions to take, instead it finds out what to do by *trial-and-error search*
  
  Eg. Players trained by playing thousands of simulated games, with no expert input on what are good or bad moves

- The environment is *stochastic*

- The *reward may be delayed*, so the learner may need to sacrifice short-term gains for greater long-term gains
  
  Eg. Player might get reward only at the end of the game, and needs to assign credit to moves along the way

- The learner has to balance the need to *explore* its environment and the need to *exploit* its current knowledge
  
  Eg. One has to try new strategies but also to win games
Implementing reinforcement learning

- A policy $\pi : S \times A \rightarrow [0, 1]$ is a way of choosing actions
- The value of a state is the expected value of a long-term return (cumulative function of the rewards)
  - E.g. average reward per time step over a long horizon
  - E.g. Discounted return:
    \[
    V^*(s) = \max_{\pi} \mathbb{E}_\pi \left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s_t = s\right]
    \]
    where $\gamma \in [0, 1]$ is a discount factor (probability of the task finishing at each step, or inflation rate) and $\pi$ dictates the choices of action
- One can also condition on actions as well as states: $Q(s, a)$
- General approach: approximate the value of the current policy from data, then use these values to guide policy change
- If an action leads to an improved state of affairs, the tendency to pick it is strengthened (i.e., the action is reinforced)
The Curse of Dimensionality

- Values are governed by nice recursive equations:
  \[ V_{k+1}(s) \leftarrow \max_{a \in A} \left( r_a(s) + \gamma \sum_{s' \in S} P_a(s'|s)V_k(s') \right), \forall s \in S \]

- The number of states grows **exponentially** with the number of state variables (the dimensionality of the problem)
  E.g. in Go, there are \( 10^{170} \) states

- The **action set** may also be very large or continuous
  E.g. in Go, branching factor is \( \approx 100 \) actions

- The solution may require **chaining many steps** to find any information
  E.g. in Go games take \( \approx 200 \) actions
How to Handle RL Big Data

- *Approximate the iterations* (using sampling, cf. asynchronous dynamic programming, temporal-difference learning)
- *Generalize* the value function to unseen states using *function approximation*
- *Shape the time scale* and nature of the actions using *temporal abstraction*
Simplifying the iterations
Temporal-difference (TD) learning (Sutton, 1988)

• Instead of looping over all states as in a Bellman backup target:
  \[
  \left( r_a(s) + \gamma \sum_{s' \in S} P_a(s'|s)V_k(s') \right), \forall s \in S
  \]
  we will *sample transition* and use the samples

• Estimated value at time $t$: $V(s_t)$
• Estimated value at time $t+1$: $r_{t+1} + \gamma V(s_{t+1})$
• *Temporal-difference error*:
  \[
  \delta = [r_{t+1} + \gamma V(s_{t+1})] - V(s_t)
  \]
  This is the *surprise* based on the new information at time step $t + 1$

• Main idea: use TD-error to drive the learning of the correct values
Representing Value Functions

• Instead of using vectors with one entry per state, suppose that $V$ is represented by some function approximator taking as input a description of the state, or feature vector $\phi_s$

• E.g. Fitted Value Iteration:
  Given $\langle s, a, s', r \rangle$ tuples and a current estimate $Q(s, a)$, form a data set of inputs $\phi_s$ and outputs $r + \gamma \max_{a'} Q(\phi_{s'}, a')$ and train a new approximation for $Q$

• We gain both in terms of space, and in terms of ability to generalize data to new situations

• Note that unlike in supervised learning, target values depend on the current approximator which causes interesting theoretical issues
What kind of function approximators?

- Linear (e.g. Sutton, 1998; Silver et al, 2010; Keller et al, 2006)
- Random projections (Fard et al, 2012)
- Nearest-neighbor
- Kernels (e.g. Barreto et al, 2012, 2013)
- Neural networks / deep architectures (e.g. Mnih et al, 2015)
- Randomized trees (e.g. Ernst et al, 2006)
- ...
Example: TD-Gammon (Tesauro, 1990-1995)

- Early predecessor of AlphaGo
- Learning from self-play, using TD-learning
- Became the best player in the world
- Discovered new ways of opening not used by people before
Example: AlphaGo (Silver et al, 2015-present)

- Perceptions: state of the board
- Actions: legal moves
- Reward: +1 or -1 at the end of the game
- Trained by playing games against itself
- Invented new ways of playing which seem superior
Policy Search

- Sometimes, the value function might be complex but the policy itself may be simple (Farahmand et al, 2015)
- Instead of relying on the value function, one can search through a space of parametrized policies $\pi_\theta$
- Outline:
  1. Initialize candidate policy
  2. Repeat
     - Estimate a new direction in which to move the parameters (using Monte Carlo, value-based methods etc)
     - Adjust the policy
**Actor-critic architecture**

- Clear optimization objective: average or discounted return
- Continual learning
- Handles both discrete and continuous states and actions
What is temporal abstraction?

• Consider an activity such as cooking dinner

  – High-level steps: choose a recipe, make a grocery list, get groceries, cook,...
  – Medium-level steps: get a pot, put ingredients in the pot, stir until smooth, check the recipe ...
  – Low-level steps: wrist and arm movement while driving the car, stirring, ...

• All have to be seamlessly integrated!
• Cf. macro actions in classical AI, controllers in robotics
Formalization of temporal abstraction

- Hierarchical abstract machines (Parr, 1998)
- MAXQ (Dietterich, 1998)
- Dynamic motion primitives (Schaal et al. 2004)
- Skills (Konidaris et al, 2009)
- Feudal RL (Dayan, 1994)
- Options (Sutton, Precup & Singh, 1999; Precup, 2000)
Options framework

- Suppose we have an MDP $\langle S, A, r, P, \gamma \rangle$
- An option $\omega$ consists of 3 components
  - An initiation set of states $I_\omega \subseteq S$ (aka precondition)
  - A policy $\pi_\omega : S \times A \rightarrow [0, 1]$
    $\pi_\omega(a|s)$ is the probability of taking $a$ in $s$ when following option $\omega$
  - A termination condition $\beta_\omega : S \rightarrow [0, 1]$
    $\beta_\omega(s)$ is the probability of terminating the option $\omega$ upon entering $s$
- Eg., robot navigation: if there is no obstacle in front ($I_\omega$), go forward ($\pi_\omega$) until you get too close to another object ($\beta_\omega$)

Cf. Sutton, Precup & Singh, 1999; Precup, 2000
Options as behavioral programs

- **Call-and-return execution**
  - Option is a subroutine which gets called by a policy over options $\pi_{\Omega}$
  - When called, $\omega$ is pushed onto the execution stack
  - During the option execution, the program looks at certain *variables (aka state)* and executes an *instruction (aka action)* until a termination condition is reached
  - The option can keep track of additional *local variables*, e.g., counting number of steps, saturation in certain features (e.g., Comanici, 2010)
  - *Options can invoke other options*

- **Interruption**
  - At each step, one can check if a better alternative has become available
  - If so, the option currently executing is *interrupted* (special form of concurrency)

- *The option identity is also a form of memory: what is the agent currently trying to achieve?* Cf. Shaul et al, 2014, Kulkarni et al, 2016
Option models

• Option model has two parts:
  1. Expected reward $r_\omega(s)$: the expected return during $\omega$’s execution from $s$
     – Needed because it is used to update the agent’s internal representations
  2. Transition model $P_\omega(s'|s)$: a sub-probability distribution over next states (reflecting the discount factor $\gamma$ and the option duration) given that $\omega$ executes from $s$
     – $P$ specifies where the agent will end up after the option/program execution and when termination will happen

• Models are predictions about the future, conditioned on the option being executed
Option models provide semantics

- Programming languages: preconditions (initiation set) and postconditions
- Models of options represent (probabilistic) post-conditions
- Models that are compositional, can be used to reason about the policy over options
- Sequencing

\[
\begin{align*}
    r_{\omega_1 \omega_2} &= r_{\omega_1} + P_{\omega_1} r_{\omega_2} \\
    P_{\omega_1 \omega_2} &= P_{\omega_1} P_{\omega_2}
\end{align*}
\]

- Stochastic choice: can take expectations of reward and transition models
- These are sufficient conditions to allow Bellman equations to hold
- Silver & Ciosek (2012): re-write model in one matrix, compose models to construct programs
  Eg. good generalization in Towers of Hanoi
MDP + Options = Semi-Markov Decision Process

- Introducing options in an MDP induces a related semi-MDP
- Hence *all planning and learning algorithms* from classical MDPs transfer directly to options (Cf. Sutton, Precup & Singh, 1999; Precup, 2000)
- But planning and learning with options can be much faster!
Illustration: Navigation

with cell-to-cell primitive actions

\begin{align*}
V(\text{goal}) &= 1 \\
\end{align*}

Iteration #0  \\
Iteration #1  \\
Iteration #2

with room-to-room options

\begin{align*}
V(\text{goal}) &= 1 \\
\end{align*}

Iteration #0  \\
Iteration #1  \\
Iteration #2
Illustration: Random landmarks

- **Generate a lot of options, then worry about which are useful!**
- Large set of *landmarks*, i.e. states in the environment, chosen at random ([Mann, Mannor & Precup, 2015])
- Rough planner which can get to a landmark from its vicinity, by solving a *deterministic relaxation* of the MDP

Landmark-based approximate value iteration gets a good solution much faster!
The anatomy of the reward option model

- Primitive action model: \( r_a(s) = \mathbb{E}[r_t | s_t = s, a_t = a] \)
- Option model:

\[
r_\omega(s) = \mathbb{E}[r_t + \gamma r_{t+1} + \ldots | s_t = s, \omega_t = \omega]
\]

- This expectation indicates a Markov-style property, as it depends only on the identity of the state and the option, not on the time step
- Notice the model is basically a value function so we can write Bellman equations for the model:

\[
r_\omega(s) = \sum_a \pi_\omega(a | s)[r_a(s) + \sum_{s'} \gamma(1 - \beta_\omega(s'))r_\omega(s')]
\]

- This means that we can use RL methods to learn the models of options!
- Very similar equations hold for the transition model
Intra-option algorithms

- Learning about one option at a time is very inefficient
- In fact, we may not want to execute options at all!
- Instead, learn about all options consistent with the behaviour
- In some sense, a form of attention
- E.g. action-value function, tabular case

On single-step transition \( \langle s, a, r, s' \rangle \), for all \( \omega \) that could have been executing in \( s \) and taken \( a \):

\[
Q_\Omega(s, \omega) = Q_\Omega(s, \omega) + \alpha[r_a(s) + \gamma(1 - \beta_\omega(s'))Q_\Omega(s', \omega) + \\
+ \gamma\beta_\omega(s') \sum_{s', \omega'} \max \omega' Q_\Omega(s', \omega') - Q_\Omega(s, \omega)]
\]

Red: continuation. Blue: termination

- In general function approximation, importance sampling will need to be used (several papers on this)
Frontier: Option Discovery

- Options can be given by a system designer
- If subgoals / secondary reward structure is given, the option policy can be obtained, by solving a smaller planning or learning problem (cf. Precup, 2000)
- What is a good set of subgoals / options?
- This is a representation discovery problem
- Studied a lot over the last 15 years
- Bottleneck states and change point detection currently the most successful methods
Goals of our current work

- Explicitly state an *optimization objective* and then solve it to find a set of options
- Handle both *discrete and continuous* set of state and actions
- Learning options should be *continual* (avoid combinatorially-flavored computations)
- Options should provide *improvement within one task* (or at least not cause slow-down...)

MSR Cambridge AI Summer School, July 2017
Actor-critic architecture

- Clear optimization objective: average or discounted return
- Continual learning
- Handles both discrete and continuous states and actions
Option-critic architecture (Bacon et al, 2017)

- Parameterize internal policies and termination conditions
- Policy over options is computed by a separate process (planning, RL, ...)

Figure 1: The option-critic architecture consists of a set of options.

Figure 2: Layout of the four-rooms domain and value function.
Formulation

- The option-value function of a policy over options $\pi_\Omega$ is given by

$$Q_{\pi_\Omega}(s, \omega) = \sum_a \pi_\omega (a|s) Q_U (s, \omega, a)$$

where

$$Q_U (s, \omega, a) = r_a(s) + \gamma \sum_{s'} P_a(s'|s) U(\omega, s')$$

- The last quantity is the utility from $s'$ onwards, \textit{given that we arrive in $s'$ using $\omega$}

$$U(\omega, s') = (1 - \beta_\omega (s')) Q_{\pi_\Omega} (s', \omega) + \beta_\omega (s') V_{\pi_\Omega} (s')$$

- We parameterize the internal policies by $\theta$, as $\pi_{\omega, \theta}$, and the termination conditions by $\nu$, as $\beta_{\omega, \nu}$

- \textit{Note that $\theta$ and $\nu$ can be shared over the options!}
Main result: Gradient updates

- Suppose we want to optimize the expected return: $\mathbb{E}\{Q_{\pi\Omega}(s, \omega)\}$
- The gradient wrt the internal policy parameters $\theta$ is given by:
  $$\mathbb{E}\left\{ \frac{\partial \log \pi_{\omega,\theta}(a|s)}{\partial \theta} Q_U(s, \omega, a) \right\}$$
  This has the usual interpretation: *take better primitives more often* inside the option
- The gradient wrt the termination parameters $\nu$ is given by:
  $$\mathbb{E}\left\{ -\frac{\partial \beta_{\omega,\nu}(s')}{\partial \nu} A_{\pi\Omega}(s', \omega) \right\}$$
  where $A_{\pi\Omega} = Q_{\pi\Omega} - V_{\pi\Omega}$ is the advantage function
  This means that we want to *lengthen options that have a large advantage*
Results: Options transfer

- 4-rooms domain, tabular representations, value functions learned by Sarsa
- Learning in the first task no slower than using primitives
- Learning once the goal is moved faster with the options
Results: Nonlinear function approximation

- Atari simulator, DQN to learn value function over options, actor as above

- Performance matching or better than DQN
Results: Learned options are intuitive

- In rooms environment, terminations are more likely near hallways (although there are no pseudo-rewards provided)

- In Seaquest, separate options are learned to go up and down
What are beneficial options

• Successful simultaneous learning of terminations and option policies
• But, as expected, *options shrink over time* unless a margin is required for the advantage
  
  Cf. time-regularized options, Mann et al, (2014)
• Intuitively, using longer options increase the speed of learning and planning (but may lead to a worse result in call-and-return execution)
• What is the right tool to formalize this intuition?
A proposal: Deliberation cost

- Assumption: *executing a policy is cheap, deciding what to do is expensive*
  - Many choices may need to be evaluated (branching factor over actions)
  - In planning, many next states may need to be considered (branching factor over states)
  - Evaluating the function approximator might be expensive (e.g. if it is a deep net)

- Deliberation is also expensive in animals:
  - Energy consumption (to engage higher-level brain function)
  - Missed opportunity cost: thinking too long means action is delayed
Problem formulation

- Let $c(s, \omega)$ be the immediate cost of deliberating to choose $\omega$ in $s$
- In the call-and-return model, it is easy to see that we have a value function that expresses total deliberation cost given by the following Bellman equation:

$$Q_c(s, \omega) = -c(s, \omega) + \sum_{s'} P_\omega(s' | s) \sum_{\omega'} \pi_\Omega(\omega' | s') Q_c(s', \omega')$$

- We can obtain $Q_c$ using learning, value iteration etc
- **New objective: maximize reward with reasonable effort**

$$\max_\Omega \mathbb{E} \left[ Q_\Omega(s, \omega) + \xi Q_c(s, \omega) \right]$$

- $\xi \geq 0$ controls the trade-off between value and computation effort ($\xi = 0$ means optimizing original reward)
• Emphasizing deliberation cost, shifts the policy towards using options
• Number of iterations of planning is smaller for higher deliberation cost penalties
• When options are learned in one task and then used to plan in a different task, options obtained with deliberation costs are more robust
Conclusions

• Reinforcement learning is useful for temporal prediction under uncertainty as well as stochastic control
• *Good representations exist to re-shape the state and action space to handle larger problems, and increase efficiency*
• Temporal abstraction methods developed in reinforcement learning provide syntax and semantics of behavioral programs
• Option-critic allows using policy gradient ideas for *continual learning of temporal abstractions*, but there are lots of things to do:
  – More empirical work in option construction
  – Tighter integration with Neural Turing Machines and similar models
  – Improved reward shaping, eg see new Ms Pacman results from van Seijen et al, Maluuba/Microsoft
  – Other execution models