Function Approximators for Reinforcement Learning

Keith L. Downing

The Norwegian University of Science and Technology (NTNU)
Trondheim, Norway
keithd@idi.ntnu.no

October 14, 2018
Large RL Search Spaces:

State space = cross product of all variable states.

Robotics

1. Sensor readings
2. Location, orientation, velocity, acceleration
3. Battery level

Texas Hold ’Em Poker

1. A player’s hole cards, of which there are \( \binom{52}{2} \) possibilities.
2. Number of raises in the current betting round.
3. Number of players still active in the current betting round.
4. Money in the pot (preferably discretized into \( k \) different bins)
5. The (discretized and binned) strength rating of the player’s hand - based on her hole cards plus the mutual cards (flop, turn, river).
6. Number of active players known to bluff frequently
Explosion of possible states $\rightarrow$ tables are impossible to build.

- Neural net = a function approximator (*funcapp*) from states to evaluations (critic) or states to actions (actor).

- The *funcapp* must handle every possible state as input.

- **Key generality assumption:** For the vast majority of cases, similar states map to similar values.
Learning with Tables -vs- Funcapps

- Using tables for state values in TD(0), update the table entry $V(s)$ for state $s$ via:

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$$

where $\delta = r + \gamma V(s') - V(s)$; episode = $s \rightarrow s' \rightarrow s'' \ldots$

- $\delta$ = error in the current estimate of $V(s)$.
- Clearly, this update ONLY affects $V(s)$.
- With a Function Approximator, there is no entry just for $V(s)$. There are model parameters ($\theta$), where $|\theta| << |S|$.
- Based on $\delta$, how can we modify $\theta$ to get a better approximation of $V(s)$...but without ruining our estimates for $V(s') \forall s' \neq s$?
- If the funcapp is an NN, how do we modify the weights when given $\delta$ to reduce our overall error ($\forall s \in S$)?
- This has become a supervised learning problem, with each training case = features $(s)$ + target $(r + \gamma V(s'))$. 

Keith L. Downing

Function Approximators for Reinforcement Learning
Stochastic Gradient Descent (SGD) in RL Funcapps

**Loss Function** \( L(\theta) \)

\[
L(\theta) = \frac{1}{2} \sum_{s \in MB} (V^*(s) - V_\theta(s))^2
\]

where

- \( MB = \) a minibatch of states
- \( V^*(s) = \) actual value of \( s = \) the target
- \( V_\theta(s) = \) estimate of \( V^*(s) = \) output of NN with params \( \theta \)

Using a minibatch of 1 case (\( s \)), \( \forall w \in \theta \), compute:

\[
\nabla_w L(\theta) = \frac{\partial L(\theta)}{\partial w} = \left[ V^*(s) - V_\theta(s) \right] \nabla_w (-V_\theta(s))
\]

The weight update (where \( \alpha = \) learning rate):

\[
\Delta w = -\alpha \nabla_w L(\theta) = \alpha \left[ V^*(s) - V_\theta(s) \right] \nabla_w V_\theta(s)
\]
What is the target, $V^*(s)$?

Assume $s = s_t = \text{state visited at step } t \text{ of an episode}$. 

- **1-step TD:** $\text{TD}(0) \; \{ \gamma = \text{discount} \}$
  
  $$V^*(s_t) = r_{t+1} + \gamma V_\theta(s_{t+1})$$

- **N-Step TD**
  
  $$V^*(s_t) = G_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_\theta(S_{t+n})$$

- **Monte Carlo** \{ $T =$ final step of the episode \}
  
  $$V^*(s_t) = G_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-t-1} r_T$$

For TD methods, bootstrapping (red terms) makes $V^*(S_t)$ a function of $\theta$, since $V_\theta(S_{t+n})$ is a function of $\theta$. Hence, $\frac{\partial L(\theta)}{\partial w}$ is more complex; this is **Semi-Gradient Descent**... but it still works pretty well.
NN = funcapp $V_\theta$ that maps states to evaluations.

1. Compute the symbolic gradients: $\forall w : \nabla_w V_\theta$

2. Do one episode of RL search.
   Along the way (TD) or at episode end (MC), collect cases: $(s_t, V^*(s_t))$.

3. $\forall (s, V^*(s)) \in$ cases: \{Train $V_\theta$ on each case\}
   - $\forall w \in \theta$:
     - Instantiate the symbolic gradients with $s$ and $V_\theta(s)$, yielding numeric gradients: $\forall w : \nabla_w V_\theta(s)$
     - Update $w$:
       \[
       \Delta w \leftarrow \alpha [V^*(s) - V_\theta(s)] \nabla_w V_\theta(s)
       \]

4. Go to 2

- Gradients such as $\nabla_w V_\theta$ are standard backprop calculations.
- For 1-step TD, $\Delta w \leftarrow \alpha \delta \nabla_w V_\theta(s)$
Semi-Gradient Descent for TD($\lambda$)

- $V^*(s_t)$ is the same as for 1-step TD, regardless of $\lambda$.

$$V^*(s_t) = r_{t+1} + \gamma V_\theta(s_{t+1})$$

- TD Error is then:

$$\delta = V^*(s_t) - V_\theta(s_t) = r_{t+1} + \gamma V_\theta(s_{t+1}) - V_\theta(s_t)$$

- Each weight, $w_j$, has an eligibility $e_j$, updated via:

$$e_j \leftarrow \gamma \lambda e_j + \nabla w_j V_\theta(s_t)$$

where $\gamma =$ discount, $\lambda =$ eligibility decay factor.

- Weight updates (with $\alpha =$ learning rate) are:

$$w_j \leftarrow w_j + \alpha \delta e_j$$
Eligibility Traces in Backprop Nets

Eligibilities decay on each step and are incremented when $\nabla_{w_j} V_\theta(s_t) \neq 0$

$$e_j \leftarrow \gamma \lambda e_j + \nabla_{w_j} V_\theta(s_t)$$

$$w_j \leftarrow w_j + \alpha \delta e_j$$

$$dV(S) / dw = d(Sum) / dw \times d(B)/d(Sum) \times \ldots$$

$$= A \times d(B)/d(Sum) \times \ldots$$

$dV(S) / dw$ is only significant when $\text{abs}(A)$ is non-zero

$\Rightarrow$ $w$ does not always contribute to $V(S)$

$\Rightarrow$ $w$'s eligibility trace is not always incremented.
TD-Gammon (Tesauro, 1995)

$S_t = \text{state of the board}$

$S_{t+1}^{1} a_1 S_{t+1}^{2} a_2 S_{t+1}^{k} a_k$

$V(S_t)$

$\delta_t = r^* + V(S_{t+1}^*) - V(S_t)$

Learn

Do move $a^*$

Keith L. Downing  Function Approximators for Reinforcement Learning
General Observations

ANN as Value function

Many search spaces are too large to yield accurate $V(s)$ entries via standard RL exploration. ANNs are an adaptive approximator of $V(s)$ that allow systems to bootstrap to effective policies.

RL to Supervised Learning via TD

1. In TD, each move leads to a comparison of expectations to slightly more informed expectations, and any difference between the two is a surprise that leads to learning.

2. If an ANN (or any system) can predict future states (based on current states and actions), then

3. when the future arrives, reality serves as the correct answer, which, when compared to the prediction, provides an error term, which can guide learning.

4. This can be done at every timestep.

5. So prediction + comparison to reality turns RL into supervised learning.