RL with function approximation

MAL Seminar
The Problem

- Realistic learning problems tend to have very large / continuous state and/or action spaces
- Exact (tabular) RL methods need to store a value for each (s,a) pair
- Simple discretization may not be feasible
Function approximation

- **Function approximation** (neural networks, trees, kernel based, ...) is needed to represent value functions and policies.

- (Q-)Value function learning is now an (online) regression problem: using samples \((s,a,Q(s,a))\), train an approximator to predict \(Q(s,a)\) given \((s,a)\).

- 2 learning problems:
  - learn control from samples
  - minimize representation errors

Good ref: *Reinforcement Learning and Dynamic Programming Using Function Approximators* by Lucian Busoniu, Robert Babuska, Bart De Schutter, Damien Ernst. CRC Press, Automation and Control Engineering Series. 2010
Fitted Q-iteration

**input**: set of samples \((s,a,r,s')\)

\[
Q := Q_0 \quad //\text{initialize approximator}
\]

**repeat**

\[
\text{Targets} := r + \max_a Q(s',a')
\]

\[
Q := \text{Train(samples,Targets)}
\]

**until** finished

- Batch method: either completely offline or limited system interactions
- Expects training set of transitions as input
- Each step: single Q-learning backup over all data
- Retrains approximator on complete data (e.g. using backpropagation)
Fitted Q-iteration

● Pros:
  ○ Very sample efficient (less transitions needed than with learning online)
  ○ Can use powerful approximators & learning algorithms without online learning issues
  ○ Works well with neural networks, decision trees

● Cons:
  ○ No online control/learning
  ○ Training data must be representative (no exploration possible)
Online algorithms

- Online SARSA(λ), Q(λ), actor-critic can also be extended to use function approximation
- Online training can lead to issues with function approximation (divergence, unbalanced trees …)
- Successful examples exist (TD-gammon)
- Typically, linear function approximation is used.
Linear Function Approximation

Function is represented as linear combination of a feature vector $\phi$ and learned weights $\theta$:

$$Q_{\theta}(s, a) = \phi_{sa}^T \theta$$

or

$$Q_{\theta}(s, a) = \sum_{i} \phi_{sa}[i] \theta[i]$$

the vector $\phi$ consists of features (or basis functions) that describe $(s,a)$. $\theta$ is a vector of weights updated by RL (replace Q-values)
Online (linear) Gradient descent

\[
\begin{align*}
\text{theta} &:= 0 \\
\text{repeat:} \\
& \quad \text{get sample } (x, f(x)) \\
& \quad \text{err} := f(x) - \phi(x)^T \text{theta} \\
& \quad \text{g\_sq\_err} = -\text{err} \ast \phi(x) \\
& \quad \text{theta} = \text{theta} - \alpha \ast \text{g\_sq\_err} \\
\text{until} \ & \text{convergence}
\end{align*}
\]

- Simple algorithm to optimize parameters \( \theta \)
- Minimize squared error between target \( f(x) \) and estimated \( f_\theta(x) = \phi(x)^T \theta \)
- Each update of \( \theta \) take a step in direction of the gradient of the squared error
Example

- Target function:
  \[ f(x) = 1.5x^2 + 3x \]
- \( \phi(x) = [x^2; x] \)
- \( \theta \) initialized to [0;0]
- approximation:
  \[ f_\theta(x) = \phi(x)^T \theta \]
  \[ = \theta[0]x^2 + \theta[1]x \]
Tile Coding

- State space is covered with overlapping grids
- $\phi_s$ is binary vector with 1 value for each tile
- In each grid only the tile in which state $x$ falls is active
- $\phi_s$ is 1 for active tiles, 0 else
Tile Coding ctd.

\[ \phi: [0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0] \]

\[ \theta: [\theta_1, \theta_2, \theta_3, \ldots, \theta_{46}, \theta_{47}, \theta_{48}] \]

\[ Q = \phi^T \theta = \theta_5 + \theta_{25} + \theta_{42} \]
Tile Coding with Hashing

- Can drastically reduce memory requirements
- Typically only small part of state space needs high resolution approximation
- Maps noncontiguous, disjoint regions randomly spread throughout the state space to the same tile
- Often works well even with possible collisions
Tile Coding resolution

- Tile coding learns piecewise constant approximation
- Resolution:
  \[ r = \frac{\text{tile\_width}}{\#\text{tilings}} \]
- Resolution can be increased by adding tilings or reducing tile width
- More tilings give better generalization
Radial Basis Functions

\[ \phi_s(i) = \exp \left( -\frac{||s - c_i||^2}{2\sigma_i^2} \right). \]

- select number of centers (prototype states)
- 1 feature per center
- features indicate how similar input is to each prototype
- smooth approximation

\[ \phi: \begin{bmatrix} 0.3314, & 0.6455, & 0.0230 \end{bmatrix} \]
\[ \theta: \begin{bmatrix} \theta_1, & \theta_2, & \theta_3 \end{bmatrix} \]

\[ Q = \phi^T \theta = 0.3314\theta_0 + 0.6455\theta_1 + 0.023\theta_3 \]
RBF vs Tiling

The diagram illustrates the comparison between RBF (Radial Basis Function) and Tiling methods in the context of state value features. The features are divided into three categories: feature 1, feature 2, and feature 3. The RBF method smoothly assigns values across these features, while Tiling discretely assigns values at specific points (0 and 1). The state value axis indicates the continuous and discrete representation of these methods.
State vs state-action

- For continuous action spaces, actions can be added as another input dimension in the approximator.
- Q-learning and sarsa still usually need discretize to select greedy action.
- For discrete action sets, separate $\theta$ can be learnt for every action.
The Q-function approximation can also be learned by minimizing the squared error with gradient descent:

$$E = \frac{1}{2}[Q(s, a) - Q_\theta(s, a)]^2$$

Gradient descent learning of values:

$$\theta := \theta - \alpha \frac{\delta E}{\delta \theta}$$

With linear function approximation this becomes:

$$\theta := \theta + \alpha [Q(s, a) - Q_\theta(s, a)] \phi_{sa}$$
During learning we estimate the true $Q(s,a)$ by an observed sample:

$$\theta := \theta + \alpha [Q(s, a) - Q_\theta(s, a)] \phi_{sa}$$

becomes (with bootstrapping):

$$\theta := \theta + \alpha [r + \gamma Q_\theta(s', a') - Q_\theta(s, a)] \phi_{sa}$$

or:

$$\theta := \theta + \alpha [r + \gamma \phi_{s'a'}^T \theta - \phi_{sa}^T \theta] \phi_{sa}$$

which we write as:

$$\theta := \theta + \alpha \delta \phi_{sa}$$
Several possible estimates can be used as target Q-function for gradient descent:

- full Monte Carlo return ($\lambda = 1$)
- bootstrapping 1 step backup ($\lambda = 0$)
- n-step return ($0 \leq \lambda \leq 1$)

Only the full return is an unbiased estimate that guarantees convergence to a local optimum, others will converge to some neighborhood of optimum.
Eligibility Traces

Traces can be used to implement **n-step returns** as in the tabular case. Trace values now indicate **activation** of **features** rather than state-action pairs:

\[
e := \gamma \lambda e
\]

\[
e := e + \phi_{sa}
\]

The update for \( \theta \) with traces becomes:

\[
\theta := \theta + \alpha \delta e
\]

where \( \delta \) is the again TD-error: \((r + \gamma V(s') - Q(s,a))\)
Linear Sarsa(\(\lambda\))

Initialize \(\tilde{\theta}\) arbitrarily
Repeat (for each episode):
\[ \tilde{e} = 0 \]
\[ s, a \leftarrow \text{initial state and action of episode} \]
\[ \mathcal{F}_a \leftarrow \text{set of features present in } s, a \]
Repeat (for each step of episode):
  For all \(i \in \mathcal{F}_a:\)
  \[ e(i) \leftarrow e(i) + 1 \] (accumulating traces)
  or \(e(i) \leftarrow 1\) (replacing traces)
Take action \(a\), observe reward, \(r\), and next state, \(s\)
\[ \delta \leftarrow r - \sum_{i \in \mathcal{F}_a} \theta(i) \]
With probability \(1 - \varepsilon:\)
  For all \(a \in \mathcal{A}(s):\)
  \[ \mathcal{F}_a \leftarrow \text{set of features present in } s, a \]
  \[ Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i) \]
  \[ a \leftarrow \arg \max_a Q_a \]
else
  \(a \leftarrow \text{a random action } \in \mathcal{A}(s)\)
\[ \mathcal{F}_a \leftarrow \text{set of features present in } s, a \]
\[ Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i) \]
\[ \delta \leftarrow \delta + \gamma Q_a \]
\[ \tilde{\theta} \leftarrow \tilde{\theta} + \alpha \delta \tilde{e} \]
\[ \tilde{e} \leftarrow \gamma \lambda \tilde{e} \]
until \(s\) is terminal

- n-step return to estimate \(Q(s,a)\)
- gradient(-like) update to learn parameters \(\theta\)
Linear $Q(\lambda)$

Initialize $\tilde{\theta}$ arbitrarily
Repeat (for each episode):

$\bar{e} \leftarrow 0$

$s, a \leftarrow$ initial state and action of episode
$\mathcal{F}_a \leftarrow$ set of features present in $s, a$

Repeat (for each step of episode):

For all $i \in \mathcal{F}_a$: $e(i) \leftarrow e(i) + 1$

Take action $a$, observe reward, $r$, and next state, $s$

$\delta \leftarrow r - \sum_{i \in \mathcal{F}_a} \theta(i)$

For all $a \in \mathcal{A}(s)$:

$\mathcal{F}_a \leftarrow$ set of features present in $s, a$

$Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$

$\delta \leftarrow \delta + \gamma \max_a Q_a$

$\tilde{\theta} \leftarrow \tilde{\theta} + \alpha \delta e$

With probability $1 - \varepsilon$:

For all $a \in \mathcal{A}(s)$:

$Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$

$a \leftarrow \arg \max_a Q_a$

$\bar{e} \leftarrow \gamma \lambda \bar{e}$

else

$a \leftarrow$ a random action $\in \mathcal{A}(s)$

$\bar{e} \leftarrow 0$

until $s$ is terminal

- Off-policy updates can lead to divergence of estimates
- Counter examples exist
- Better off-policy methods exist e.g. $GQ(\lambda)$