RL with function approximation

MAL Seminar

Discrete RL

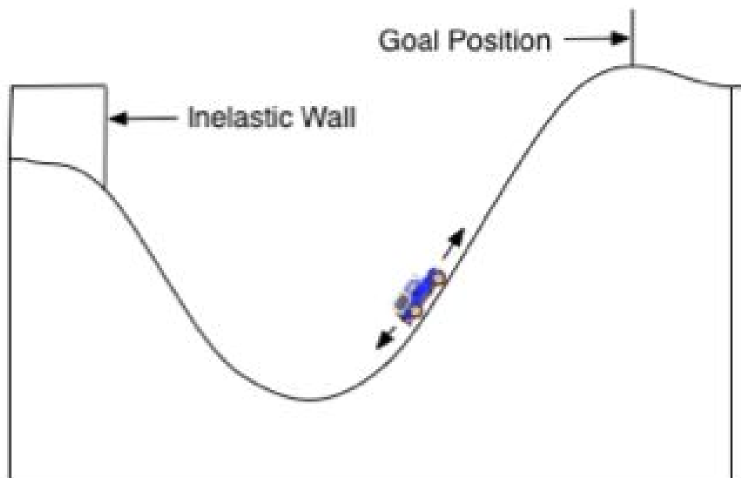
A = {a1,a2,...,an}

S = {s1,s2,...,sk}

Q(s1,a1)	...	Q(s1,an)
Q(s2,a1)	...	
	...	
Q(sk,a1)	...	Q(sk,an)

- Finite, discrete state and action spaces
- Q-function can be represented as a table
- Representation is exact
- Not possible for very large/infinite state or action spaces
- No generalization across states/actions

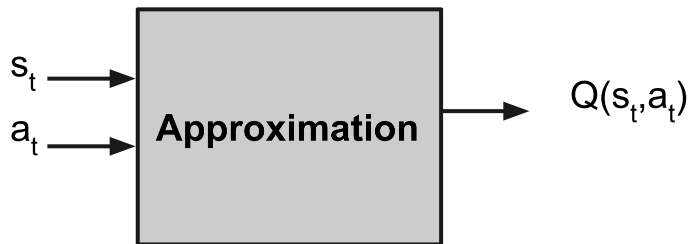
The Problem

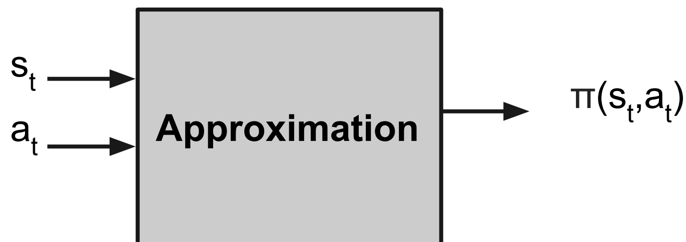


$$p \in [-1.2, 0.5]$$
$$v \in [-0.07, 0.07]$$

- 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/or **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

Batch RL

- Many function approximators (decision trees, neural networks) are more suited to **batch learning**
- Batch RL attempts to solve reinforcement learning problem using **offline transition data**
- No online control
- Separates the approximation and RL problems: train a sequence of approximators

Fitted Q-iteration

input: set of samples (s,a,r,s')
Q := Q_0 //initialize approximator

repeat
    Targets := r + $\max_{a'}$ Q(s',a')
    Q := 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)
  - Often difficult to get data with naive policies (random policies)

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** ϕ and learned **weights** θ:

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

or

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

the vector ϕ consists of **features** (or basis functions) that describe (s,a). θ is a vector of **weights** updated by RL (replace Q-values)

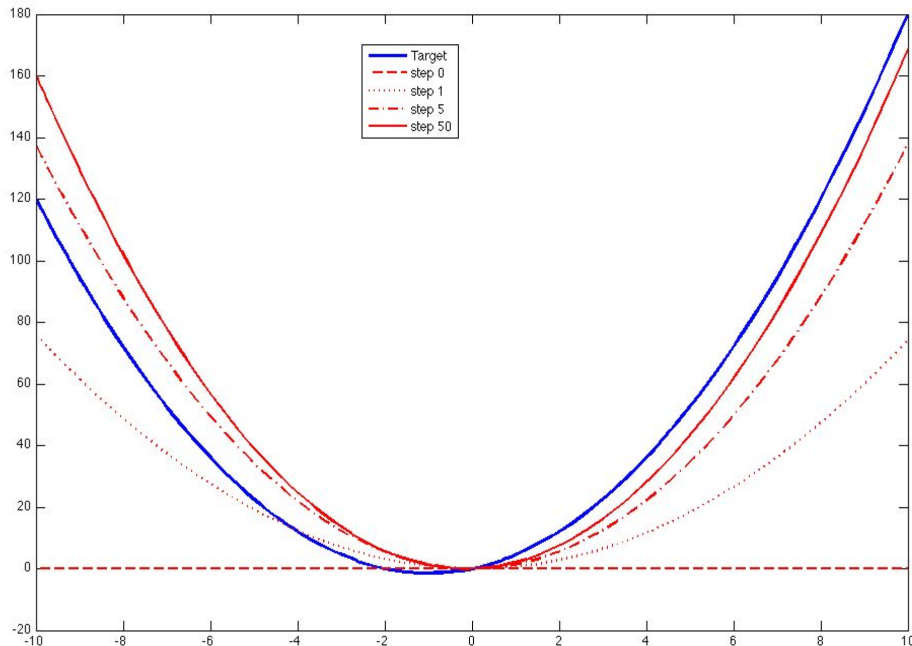
Gradient descent

Online (linear) Gradient descent

```
theta:= 0
repeat:
    get sample (x,f(x))
    err := f(x) - phi(x)^T*theta
    g_sq_err =  -err * phi(x)
    theta = theta - alpha * g_sq_err
until convergence
```

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

Example



- Target function:

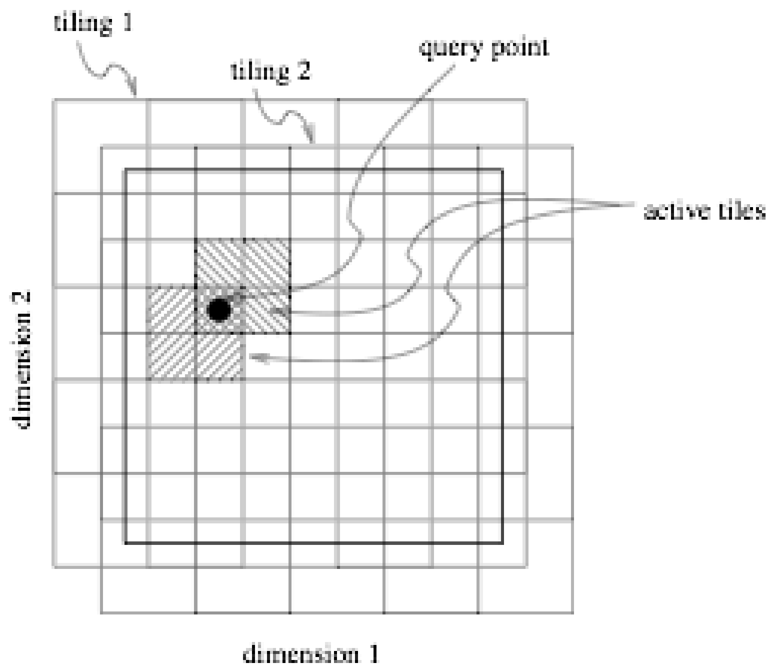
  $f(x) = 1.5x^2+3x$
- $\phi(x) = [x^2; x]$
- θ initialized to [0;0]
- approximation:

$$
\begin{aligned}
f_\theta(x) &= \phi(x)^T\theta \\
&= \theta[0]x^2 + \theta[1]x
\end{aligned}
$$

Basis Functions

- We need **feature representation** of states (and actions)
- We can **hand code** these representations for specific problems
- **generic** approaches exist that turn any continuous state into feature representations:
  - Fourier basis
  - Polynomial basis
  - **Tile coding**
  - **Radial Basis functions**
  - Kanerva coding
  - ….

Tile Coding



- Binary features
- State space is covered with **overlapping grids**
- ϕ_s is binary vector with 1 value for each tile
- In each grid only the tile in which state x falls is active
- ϕ_s is 1 for **active tiles**, 0 else

Tile Coding ctd.

Tiling 1 | Tiling 2 | Tiling 3

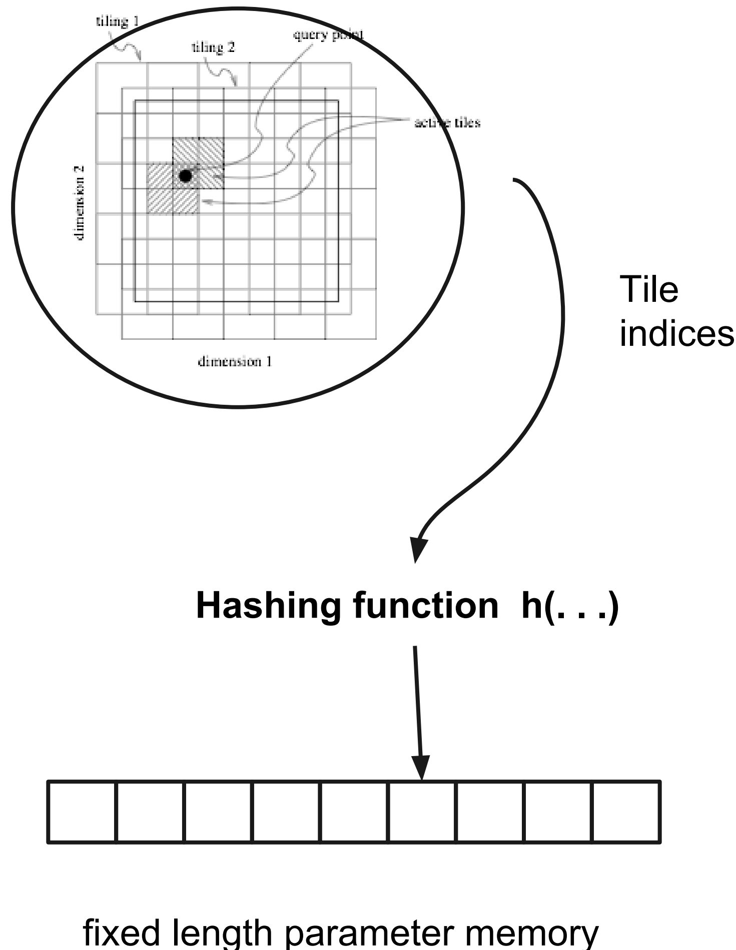
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

ϕ: [0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0]

θ: [$\theta_1, \theta_2, \theta_3,$ ... $\theta_{46}, \theta_{47}, \theta_{48}$]

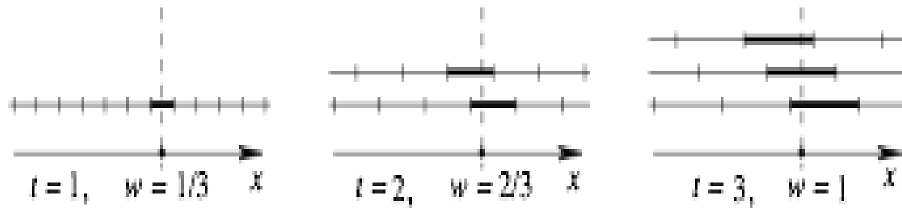
$$Q = \phi^T \theta = \theta_5 + \theta_{25} + \theta_{42}$$

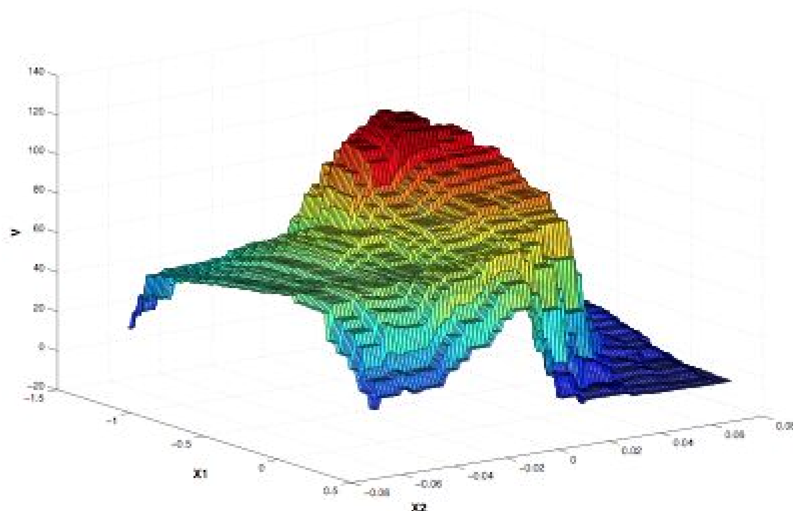
Tile Coding with Hashing



- 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





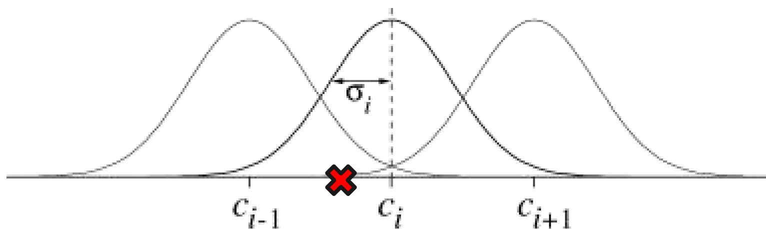
- Tile coding learns **piecewise constant** approximation
- Resolution:

$$r = \frac{tile_width}{\#tilings}$$

- Resolution can be increased by **adding tilings** or **reducing tile width**
- More tilings give better **generalization**

Radial Basis Functions

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



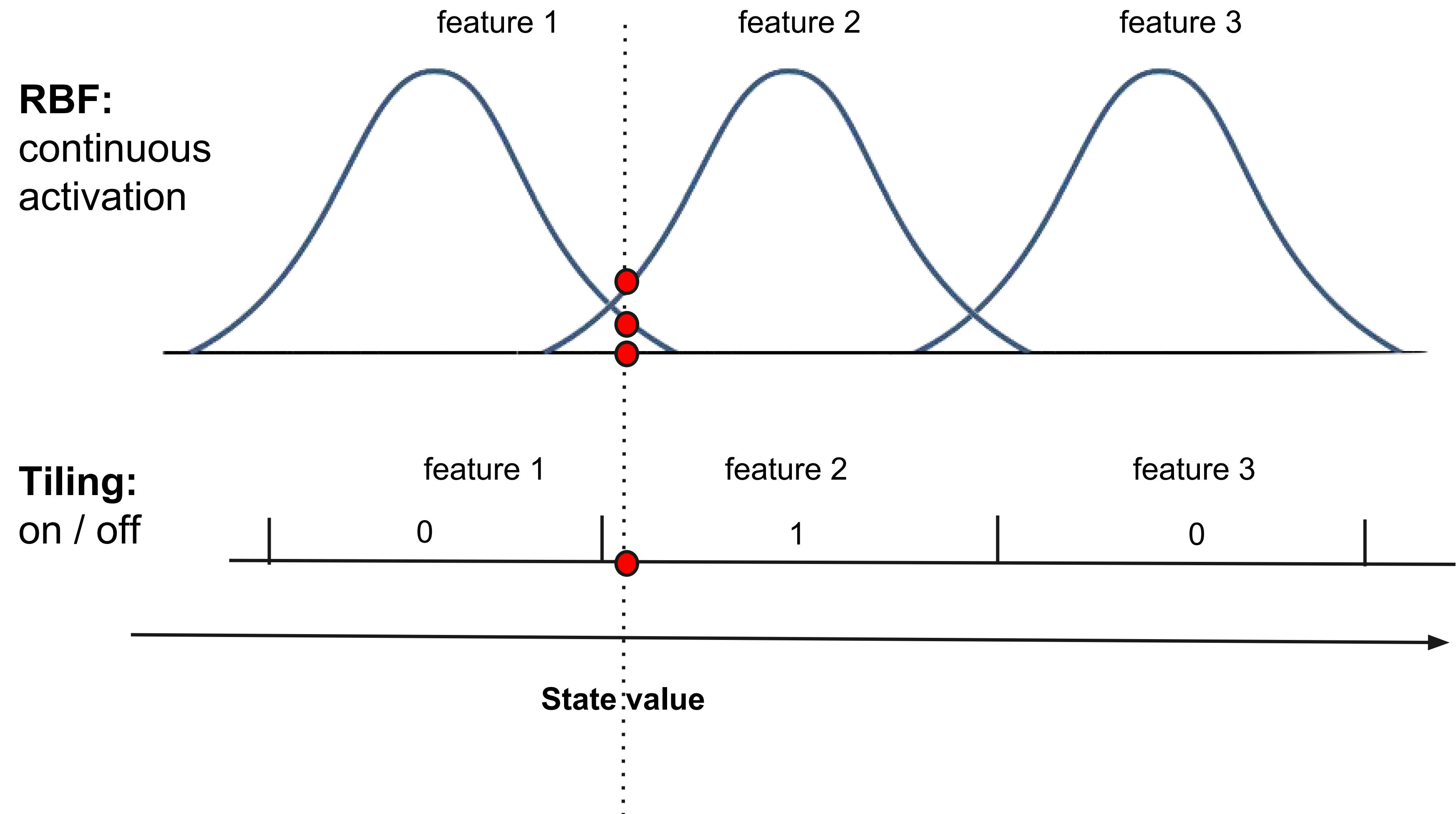
ϕ: [0.3314, 0.6455, 0.0230]

θ: [θ_1,θ_2,θ_3]

$$Q = \phi^T \theta = 0.3314\theta_0 + 0.6455\theta_1 + 0.023\theta_3$$

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

RBF vs Tiling



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 θ** can be learnt for every action

Approximating the Q-function

The Q-function approximation can be learnt 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:

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

Approximating the Q-function (2)

During learning we estimate the true Q(s,a) using 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:

$$\theta := \theta + \alpha \delta \phi_{sa}$$

Estimate Q(s,a)

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:

$$\begin{aligned} e &:= \gamma\lambda e \\ e &:= e + \phi_{sa} \end{aligned}$$

The update for θ with traces becomes:

$$\theta := \theta + \alpha\delta e$$

where δ is the again TD-error: (r + γ V(s') - Q(s,a))

Gradient issues

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

target value

- Using bootstrapping means we use the parameters we're learning in our learning target
- This is not a true gradient anymore
- Can cause stability problems, especially with off-policy updates
- Recent research: true gradient methods

Linear Sarsa(λ)

Initialize $\vec{\theta}$ arbitrarily
Repeat (for each episode):
  $\vec{e} = \vec{0}$
  $s, a \leftarrow$ 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 $i \in \mathcal{F}_a$:
      $e(i) \leftarrow e(i) + 1$ (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 $a \in \mathcal{A}(s)$:
        $\mathcal{F}_a \leftarrow$ 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
      $a \leftarrow$ a random action $\in \mathcal{A}(s)$
      $\mathcal{F}_a \leftarrow$ set of features present in s, a
      $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
    $\delta \leftarrow \delta + \gamma Q_a$
    $\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}$
    $\vec{e} \leftarrow \gamma \lambda \vec{e}$
  until s is terminal

- n-step return to estimate Q(s,a)
- gradient(-like) update to learn parameters θ

Linear Q(λ)

Initialize $\vec{\theta}$ arbitrarily
Repeat (for each episode):
  $\vec{e} = \vec{0}$
  $s, a \leftarrow$ 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
      $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
    $\delta \leftarrow \delta + \gamma \max_a Q_a$
    $\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}$
    With probability $1 - \varepsilon$:
      For all $a \in \mathcal{A}(s)$:
        $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
      $a \leftarrow \arg\max_a Q_a$
      $\vec{e} \leftarrow \gamma \lambda \vec{e}$
    else
      $a \leftarrow$ a random action $\in \mathcal{A}(s)$
      $\vec{e} \leftarrow 0$
  until s is terminal

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