Reinforcement Learning

Function approximation

Mario Martin

CS-UPC

February 26, 2019
Recap

- **Algorithms:**
  - MonteCarlo methods for Policy Evaluation and Policy Learning
  - Temporal Differences methods for Policy Evaluation and Policy Learning
  - Extension of TD to n-steps back-ups and TD($\lambda$)
  - Sarsa and Expected Sarsa algorithms

- Differences between on-policy and off-policy learning
- Exploration vs. Exploitation
- Alpha parameter for learning
Problem

Incremental algorithms
  ▶ Prediction methods
    ★ Prediction Objective: Mean Squared Value Error, denoted VE:
    ★ Easy case: Linear methods and TD
    ★ Representation: RBFs, tile Coding, Coarse represent
    ★ Stochastic gradient
    ★ MC
    ★ TD
  ▶ Control methods: Sarsa Q-learning

batch methods
  ▶ Least Squares LSTD (LSPI)
  ▶ Kernel methods
  ▶ NNs and Deep Learning

Average reward
Goal of this lecture

- Methods we have seen so far work well when we have a *tabular* representation for each state, that is, when we represent value function with a lookup table.
- This is not reasonable on most cases:
  - In large state spaces: There are too many states and/or actions to store in memory (f.i. Backgammon: $10^{20}$ states, Go $10^{170}$ states)
  - and in continuous state spaces (f.i. robotic examples)
Goal of this lecture

- Methods we have seen so far work well when we have a *tabular* representation for each state, that is, when we represent value function with a lookup table.
- This is not reasonable on most cases:
  - In large state spaces: There are too many states and/or actions to store in memory (f.i. Backgammon: $10^{20}$ states, Go $10^{170}$ states)
  - and in continuous state spaces (f.i. robotic examples)
- In addition, we want to generalize from/to similar states to speed up learning. It is too slow to learn the value of each state individually.
Goal of this lecture

- We’ll see now methods to learn policies for large state spaces by using *function approximation to estimate value functions*:
  
  \[ V_\theta(s) \approx V^\pi(s) \]  
  
  \[ Q_\theta(s, a) \approx Q^\pi(s, a) \]  

- \( \theta \) is the set of parameters of the function approximation method (with size much lower than \(|S|\))

- Function approximation allow to generalize from seen states to unseen states and to save space.

- Now, instead of storing \( V \) values, we will update \( \theta \) parameters using MC or TD learning so they fulfill (1) or (2).
Which Function Approximation?

- There are many function approximators, e.g.
  - Artificial neural network
  - Decision tree
  - Nearest neighbor
  - Fourier/wavelet bases
  - Coarse coding

- In principle, any function approximator can be used. However, the choice may be affected by some properties of RL:
  - Experience is not i.i.d. – Agent’s action affect the subsequent data it receives
  - During control, value function $V(s)$ changes with the policy (non-stationary)
Incremental methods
Incremental methods allow to directly apply the control methods of MC, Q-learning and Sarsa, that is, back up is done using “on-line” sequence of data of the trial reported by the agent following the policy.

Most popular method in this setting is **gradient descent**, because it adapts to changes in the data (non-stationary condition).
Gradient Descent

- Let $L(\theta)$ be a differentiable function of parameter vector $\theta$, we want to minimize.
- Define the gradient of $L(\theta)$ to be:

$$\nabla_\theta L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial L(\theta)}{\partial \theta_n} \end{bmatrix}$$

- To find a local minimum of $L(\theta)$, gradient descent method adjust the parameter in the direction of negative gradient:

$$\Delta \theta = -\frac{1}{2} \alpha \nabla_\theta L(\theta)$$

where $\alpha$ is a stepsize parameter.
Gradient Descent

\[ \Delta \theta = -\frac{1}{2} \alpha \nabla_\theta L(\theta) \]
Minimizing Loss function of the approximation

Goal: Find parameter vector $\theta$ minimizing mean-squared error between approximate value function $V_\theta(s)$ and true value function $V^\pi(s)$

$$L(\theta) = \mathbb{E}_\pi [(V^\pi(s) - V_\theta(s))^2] = \sum_{s \in S} \mu^\pi(s) [V^\pi(s) - V_\theta(s)]^2$$

where $\mu^\pi(s)$ is the time spent in state $s$ while following $\pi$

- Gradient descent finds a local minimum:
  $$\Delta \theta = -\frac{1}{2} \alpha \nabla_\theta L(\theta)$$
  $$= \mathbb{E}_\pi [(V^\pi(s) - V_\theta(s)) \nabla_\theta V_\theta(s)]$$

- Stochastic gradient descent (SGD) samples the gradient
  $$\Delta \theta = \alpha (V^\pi(s) - V_\theta(s)) \nabla_\theta V_\theta(s)$$
Subsection 1

Linear approximation
Linear representation of the state

- Represent state by a feature vector:
  \[
  \phi(s) = \begin{bmatrix}
  \phi_1(s) \\
  \vdots \\
  \phi_n(s)
  \end{bmatrix}
  \]

- Represent value function by a linear combination of features:
  \[
  V_\theta(s) = \phi(s)^T \theta = \sum_{j=1}^{n} \phi_j(s) \theta_j
  \]
Linear representation of the state

- For example:
  - Distance of robot from landmarks
  - Trends in the stock market
  - Piece and pawn configurations in chess
Example: RoboCup soccer keepaway (Stone, Sutton & Kuhlmann, 2005)
Example: RoboCup soccer keepaway (Stone, Sutton & Kuhlmann, 2005)

State is encoded in 13 continuous variables:

- 11 distances among the players, ball, and the center of the field
- 2 angles to takers along passing lanes
Example: RoboCup soccer keepaway (Stone, Sutton & Kuhlmann, 2005)

- Full soccer state
- Sparse, coarse, tile coding
- Linear map $\theta$
- 13 continuous state variables
- Huge binary feature vector (about 400 1's and 40,000 0's)
- Action values
Table lookup is a special case of linear value function approximation. Using table lookup features:

$$\phi^{table}(S) = \begin{bmatrix} 1(S = s_1) \\ \vdots \\ 1(S = s_n) \end{bmatrix}$$

Parameter vector is exactly value of each individual state

$$V_\theta(S) = \begin{bmatrix} 1(S = s_1) \\ \vdots \\ 1(S = s_n) \end{bmatrix}^T \cdot \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$
Another obvious way of reducing the number of states is by grouping some of them using a grid.

Drawback is that all states in the cell are equal and you don’t learn “softly” from neighbor cells.

Better approach is Coarse Coding.

Coarse coding provides large feature vector $\phi(s)$ that "overlap"
Another obvious way of reducing the number of states is by grouping some of them using a grid.

Drawback is that all states in the cell are equal and you don’t learn “softly” from neighbor cells.

Better approach is *Coarse Coding*.

Coarse coding provides large feature vector $\phi(s)$ that ”overlap”

---

**Caution**

When using linear FA, we should ask ourselves if $V$ can be approximated by a linear function and what’s the error of this approximation.

Usually value functions are smooth (compared with reinforcement function). However, linear FA approximation error could be large, depending on the features selected.
Coarse coding using RBFs

- Each circle is a *Radial Basis Function* (center $c$ and a width $\sigma$) that represents a feature.

Value for each feature is:

$$
\phi_i(s) = e^{-\frac{\|x-c_i\|^2}{2\sigma^2}}
$$
Coarse coding using RBFs

Several possibilities:
- Number of centers (RBF’s)
- Width
- Different width for each variable of the state

a) Narrow generalization
b) Broad generalization
c) Asymmetric generalization
Coarse coding using Tiles

- RBFs return a real value for each feature. Tiles define a binary feature for each tile.
- Binary features means weighted sum easy to compute
- Number of features present at any one time is constant
- Easy to compute indexes of the features present

You can use irregular tilings or superposition of different tilings

Shape of tiles ⇒ Generalization

#Tilings ⇒ Resolution of final approximation
Going back to SGD

First nice property of SGD in linear F.A.

In the case of linear function approximation, objective function is \textit{quadratic}:

\[
L(\theta) = \mathbb{E}_{\pi} \left[ (V^\pi(s) - \phi(s)^T \theta)^2 \right]
\]

so SGD converges to global optimum:
Second nice property of SGD in linear F.A.

Gradient vector of value function is vector of feature values:

\[
\frac{\partial V_\theta(s)}{\partial \theta_i} = \sum_{j=1}^{n} \phi_j(s) \theta_j = \phi_i(s)
\]

So, update rule is particularly simple:

\[
\Delta \theta_i = \alpha (V^\pi(s) - V_\theta(s)) \phi_i(s)
\]
Subsection 2

Prediction algorithms for linear case
Have assumed true value function $V^\pi(s)$ is given in a supervised learning way

But in RL there is only rewards of trial, not examples

Solution: substitute $V^\pi(s)$ by a target that is an estimation of it. In practice,

- For MC, the target is the return $R_t$
- For TD(0), the target is the Bellman equation
MC prediction algorithm for the linear case

- The long-term-return of a trial $R_t$ is an unbiased, noisy sample of true value $V_\pi(s)$.
- Using $R_t$ as a target, we have linear Monte–Carlo policy evaluation
  \[
  \Delta \theta = \alpha (R_t - V_\theta(s)) \nabla_\theta V_\theta(s) \\
  = \alpha (R_t - V_\theta(s)) \phi(s)
  \]
- Monte–Carlo evaluation converges to optimum ($\theta$ with global minimum error)
- Moreover, MC, even when using non–linear value function approximation converges, but in this case to a local optimum
Monte Carlo policy evaluation

Given \( \pi \), the policy to be evaluated, initialize parameters \( \theta \) as appropriate (e.g., \( \theta = 0 \))

repeat
  Generate trial using \( \pi \)
  for each \( s_t \) in trial do
    \( R_t \leftarrow \) return following the first occurrence of \( s_t \)
    \( \theta \leftarrow \theta + \alpha (R_t - V_\theta(s_t)) \phi(s_t) \)  // Notice \( \theta \) and \( \phi \) are vectors
  end for
until true
Changes applying TD to the linear case:

1. Function approximation is now for the $Q$ value function:

$$Q^\pi(s, a) \approx Q_\theta(s, a) = \phi(s, a)^T \theta = \sum_{j=1}^{n} \phi_j(s, a) \theta_j$$

2. Loss function is also now for $Q$ value function:

$$L(\theta) = \mathbb{E}_\pi \left[ (Q^\pi(s, a) - \phi(s, a)^T \theta)^2 \right]$$
In TD(0) we use $Q$ of next state to estimate $Q$ on the current state using Bellman equations. So, in general,

$$
\Delta \theta_i = \alpha \left( Q^\pi(s, a) - Q_\theta(s, a) \right) \nabla_\theta Q_\theta(s, a) \\
= \alpha \left( r + \gamma Q_\theta(s', \pi(s')) - Q_\theta(s, a) \right) \nabla_\theta Q_\theta(s, a)
$$

And, in particular, for the linear case:

$$
\frac{\partial Q_\theta(s, a)}{\partial \theta_i} = \frac{\partial}{\partial \theta_i} \left( \sum_{j=1}^n \phi_j(s, a) \theta_j \right) \\
= \phi_i(s, a)
$$

and so,

$$
\Delta \theta_i = \alpha \left( r + \gamma Q_\theta(s', \pi(s')) - Q_\theta(s, a) \right) \phi_i(s, a)
$$
Caution!

- No same guarantees that MC had when bootstrapping estimate of \( Q(S_t, a) \) is used as the target
- Notice that TD targets are not independent of parameters. In TD(0):
  \[
  r + \gamma Q_\theta(s', \pi(s'))
  \]
depends of \( \theta \)
- Bootstrapping methods are not true gradient descent: they take into account the effect of changing \( \theta \) on the estimate, but ignore its effect on the target. They include only a part of the gradient and, accordingly, we call them semi-gradient methods.

However, it can be proved that linear TD(0) policy evaluation converges (close) to global optimum.
TD(0) prediction algorithm for the linear case

**TD(0) policy evaluation**

Given $\pi$, initialize parameters $\theta$ arbitrarily (e.g., $\theta = 0$)

**repeat**

$s \leftarrow$ initial state of episode

**repeat**

$a \leftarrow \pi(s)$
Take action $a$ and observe $s'$ and $r$

$\theta \leftarrow \theta + \alpha (r + \gamma Q_\theta(s', \pi(s')) - Q_\theta(s, a)) \phi(s_t)$

$s \leftarrow s'$

**until** $s$ is terminal

**until** convergence
Subsection 3

Control algorithms for the linear case
Like the Control methods we used in tabular learning algorithms, we will build algorithms that iterate the two following steps:

1. Policy evaluation - Follow a method for approximate policy evaluation
   \[ Q_\theta \approx Q^\pi \]
2. Policy improvement - do policy improvement of the policy

Depending on the Policy evaluation procedure used (MC, TD, etc.), we have a different method
Examples of Control using PI

Linear FA Monte Carlo

Initialize parameters $\theta$ as appropriate (e.g., $\theta = 0$)

repeat
  Generate trial using $\epsilon$-greedy policy derived from $Q_\theta$
  for each $s_t$ in trial do
    $R_t \leftarrow$ return following the first occurrence of $s_t$
    $\theta \leftarrow \theta + \alpha (R_t - Q_\theta(s_t, a_t)) \phi(s_t, a_t)$
  end for
until true
Examples of Control using PI

**Linear FA Monte Carlo**

Initialize parameters $\theta$ as appropriate (e.g., $\theta = 0$)

repeat

Generate trial using $\epsilon$-greedy policy derived from $Q_\theta$

for each $s_t$ in trial do

$R_t \leftarrow$ return following the first occurrence of $s_t$

$\theta \leftarrow \theta + \alpha (R_t - Q_\theta(s_t, a_t)) \phi(s_t, a_t)$

end for

until true

**Function $Q_\theta(s, a)$**

Given $\theta$, state $s$ and action $a$

return $\theta^T \phi(s, a)$
Select action following policy

**Function** $\pi_\theta(s)$

Given $\theta$ and state $s$

\[
\text{return } \arg \max_a \theta^T \phi(s, a)
\]

**Function implementing $\epsilon$-greedy**

Given $\theta$, $\epsilon \leq 1$ and state $s$

Select $p$ number from uniform distribution in range $[0, 1]$

\[
\text{if } p \leq \epsilon \text{ then}
\]

\[
a \leftarrow \text{Random action from } A
\]

\[
\text{else}
\]

\[
a \leftarrow \pi_\theta(s)
\]

\[
\text{end if}
\]

\[
\text{return } a
\]
Linear FA Q-learning

initialize parameters \( \theta \) arbitrarily (e.g. \( \theta = 0 \))

for each episode do

Choose initial state \( s \)

repeat

Choose \( a \) from \( s \) using policy \( \pi_\theta \) derived from \( Q_\theta \) (e.g., \( \epsilon \)-greedy)

Execute action \( a \), observe \( r, s' \)

\( \theta \leftarrow \theta + \alpha \left( r + \gamma Q_\theta(s', \pi_\theta(s')) - Q_\theta(s, a) \right) \phi(s, a) \)

\( s \leftarrow s' \)

until \( s \) is terminal

end for
Example of Control using PI

**Linear FA Sarsa: on-line learning**

initialize parameters $\theta$ arbitrarily (e.g. $\theta = 0$)

**for** each episode **do**

Choose initial state $s$

Choose $a$ from $s$ using policy derived from $Q_\theta$ (e.g., $\epsilon$-greedy)

**repeat**

Execute action $a$, observe $r$, $s'$

Choose $a'$ from $s'$ using policy derived from $Q_\theta$ (e.g., $\epsilon$-greedy)

$\theta \leftarrow \theta + \alpha (r + \gamma Q_\theta(s', a') - Q_\theta(s, a)) \phi(s)$

$s \leftarrow s'$, $a \leftarrow a'$

**until** $s$ is terminal

**end for**
Convergence of methods

- Experiments show it is desirable to bootstrap (TD in practice better than MC)
- But now we should consider convergence issues. When do incremental prediction algorithms converge?
  - When using bootstrapping (i.e., TD with < 1)?
  - When using linear function approximation?
  - When using off-policy learning?
  - When using non-linear approximation?
- Ideally, we would like algorithms that converge in all cases
We have examples of TD divergence even when exact solution is representable with linear function.

Fortunately, in practice, TD(0) works well... but we don't have guarantees.

Problem can be solved if we update parameters following an on-policy distribution (we have a proof of that). Good for Sarsa.

Unfortunately convergence guarantees on TD incremental methods only work for linear approximation.

Main cause is that TD does not follow true gradient.
Deadly triad

- The risk of divergence arises whenever we combine three things:

  **Function approximation:** Significantly generalizing from large numbers of examples.

  **Bootstrapping:** Learning value estimates from other value estimates, as in dynamic programming and temporal-difference learning.

  **Off-policy learning:** Learning about a policy from data not due to that policy, as in Q-learning, where we learn about the greedy policy from data with a necessarily more exploratory policy.

- Any two without the third is ok.
Convergence of incremental Gradient methods for prediction

<table>
<thead>
<tr>
<th>On/Off–Policy</th>
<th>Algorithm</th>
<th>Table Lookup</th>
<th>Linear</th>
<th>Non–Linear</th>
</tr>
</thead>
<tbody>
<tr>
<td>On–Policy</td>
<td>MC</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>TD(0)</td>
<td>OK</td>
<td>OK</td>
<td>KO</td>
</tr>
<tr>
<td></td>
<td>TD(λ)</td>
<td>OK</td>
<td>OK</td>
<td>KO</td>
</tr>
<tr>
<td>Off–Policy</td>
<td>MC</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>TD(0)</td>
<td>OK</td>
<td>KO</td>
<td>KO</td>
</tr>
<tr>
<td></td>
<td>TD(λ)</td>
<td>OK</td>
<td>KO</td>
<td>KO</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Table Lookup</td>
<td>Linear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------</td>
<td>--------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monte–Carlo Control</td>
<td>OK</td>
<td>OK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SARSA</td>
<td>OK</td>
<td>(OK)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q–learning</td>
<td>OK</td>
<td>KO</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(OK) = **chatters** around near–optimal value function
Conclusions and final notes about convergence

- Value-function approximation by stochastic gradient descent enables RL to be applied to arbitrarily large state spaces.
- Most algorithms just carry over the Targets from the tabular case.
- With bootstrapping (TD), we don’t get true gradient descent methods:
  - this complicates the analysis,
  - but the linear, on-policy case is still guaranteed convergent,
  - and learning is still much faster.
- For continuous state spaces, coarse/tile coding is a good strategy.
Conclusions and final notes about convergence

- Value-function approximation by stochastic gradient descent enables RL to be applied to arbitrarily large state spaces.
- Most algorithms just carry over the Targets from the tabular case.
- With bootstrapping (TD), we don’t get true gradient descent methods:
  - this complicates the analysis.
  - but the linear, on-policy case is still guaranteed convergent.
  - and learning is still much faster.
- For continuous state spaces, coarse/tile coding is a good strategy.
- Still some possible approaches: Gradient-TD (convergence in off-line linear FA) and Batch methods.
Mountain Car demonstration

- Q-learning with linear FA and Semi Gradient Methods.
  - Aggregating states
  - Tiling
Batch methods
Gradient descent is simple and appealing
  ▶ It is computationally efficient (one update per sample)
  ▶ ... But it is not sample efficient (does not take all profit from samples)

We can do better at the cost of more computational time
Batch Reinforcement Learning

- Gradient descent is simple and appealing
  - It is computationally efficient (one update per sample)
  - ... But it is not sample efficient (does not take all profit from samples)
- We can do better at the cost of more computational time

- **Batch methods** seek to find the *best fitting value function* of given agent’s experience ("training data") in a supervised way.
Subsection 1

Least Square (LS) Prediction and Least Square Policy Iteration (LSPI)
Least Squares Prediction

- Given value function approximation method with parameters $\theta$
- And experience $\mathcal{D}$ consisting of (state,value) pairs:

$$\mathcal{D} = \{\langle s_1, V_1^\pi \rangle, \langle s_2, V_2^\pi \rangle, \ldots \langle s_T, V_T^\pi \rangle\}$$

- Find parameters $\theta$ that give the best fitting of value function $V_\theta(s)$

- *Least squares* algorithm find parameter vector $\theta$ minimizing sum-squared error between $V_\theta$ and target values $V^\pi$:

$$LS(\theta) = \sum_{t=1}^{T} (V_t^\pi - V_\theta(s_t))^2$$

$$= \mathbb{E}_\mathcal{D}[(V_t^\pi - V_\theta(s_t))^2]$$
SGD with Experience Replay

- Given experience consisting of (state, value) pairs,

\[ D = \{ \langle s_1, V_1^\pi \rangle, \langle s_2, V_2^\pi \rangle, \ldots \langle s_T, V_T^\pi \rangle \} \]

- Repeat
  1. Sample state, value from experience

\[ \langle s_i, V_i^\pi \rangle \]

  2. Apply stochastic gradient descent update

\[ \Delta \theta = \alpha (V^\pi(s_i) - V_\theta(s_i)) \nabla_\theta V_\theta(s) \]

- Converges to least squares solution:

\[ \theta^\pi = \arg \min_\theta LS(\theta) \]
• Experience replay finds least squares solution
• But it may take many iterations
• Using linear value function approximation $V_\theta(s) = \phi(s)^T \theta$
• We can solve the least squares solution directly
Linear Least Squares Prediction

- At minimum of LS(w), the expected update must be zero

\[
\mathbb{E}_D[\Delta \theta] = 0
\]

\[
\alpha \sum_{t=1}^{T} \phi(s_t)(V_t^\pi - \phi(s_t)^T \theta) = 0
\]

\[
\sum_{t=1}^{T} \phi(s_t)V_t^\pi = \sum_{t=1}^{T} \phi(s_t)\phi(s_t)^T \theta
\]

\[
\left( \sum_{t=1}^{T} \phi(s_t)\phi(s_t)^T \right)^{-1} \sum_{t=1}^{T} \phi(s_t)V_t^\pi = \theta
\]

- For N features, direct solution time is \(O(N^3)\)
- Extensible to \(Q\) value function and pairs \((s, a)\)
Linear Least Squares Prediction

- We do not know true values $V_t^\pi$
- As always, substitute in equation $V_t^\pi$ for estimation from samples:
  - **LSMC** Least Squares Monte-Carlo uses return:
    $$V_t^\pi \approx R_t$$
  - **LSTD** Least Squares Temporal-Difference uses TD target
    $$V_t^\pi \approx r_t + \gamma V_\theta(s_{t+1})$$
- In each case solve directly for fixed point of MC / TD
### Convergence of LS for prediction

<table>
<thead>
<tr>
<th>On/Off-Policy</th>
<th>Algorithm</th>
<th>Table Lookup</th>
<th>Linear</th>
<th>Non-Linear</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>On-Policy</strong></td>
<td>MC</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td></td>
<td>LSMC</td>
<td>✔</td>
<td>✔</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>TD</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>LSTD</td>
<td>✔</td>
<td>✔</td>
<td>–</td>
</tr>
<tr>
<td><strong>Off-Policy</strong></td>
<td>MC</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td></td>
<td>LSMC</td>
<td>✔</td>
<td>✔</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>TD</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>LSTD</td>
<td>✔</td>
<td>✔</td>
<td>–</td>
</tr>
</tbody>
</table>
Linear Least Squares Policy Iteration (LSPI)

- How to turn the Least Square prediction algorithm into a Control algorithm?
- LSPI Algorithm - two iterated steps
  - **Policy evaluation** Policy evaluation by least squares Q-learning
  - **Policy improvement** Greedy policy improvement
- As in Q-learning, we now use $Q$ value function to get rid of transition probabilities
For policy evaluation, we want to efficiently use all experience.

For control, we also want to improve the policy.

This experience is generated from many policies.

So to evaluate $Q^\pi$ we must learn off-policy.

We use the same idea as Q-learning:

- Use experience generated by old policy $S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{old}$
- Consider alternative successor action $A' = \pi_{new}(S_{t+1})$
- Update $Q_\theta(S_t, A_t)$ towards value of alternative action $R_{t+1} + \gamma Q_\theta(S_{t+1}, A')$.
Linear Least Squares Policy Iteration (LSPI)

- LSTDQ algorithm: solve for total update = zero

\[
\alpha \sum_{t=1}^{T} \phi(s_t, a_t) (V_t^\pi - \phi(s_t, a_t)^T \theta) = 0
\]

\[
\alpha \sum_{t=1}^{T} \phi(s_t, a_t) (r_{t+1} + \gamma \phi(s_{t+1}, \pi(s_{t+1}))^T \theta - \phi(s_t, a_t)^T \theta) = 0
\]

\[
\left( \sum_{t=1}^{T} \phi(s_t, a_t)(\phi(s_t, a_t) - \gamma \phi(s_{t+1}, \pi(s_{t+1}))^T \right)^{-1} \sum_{t=1}^{T} \phi(s_t, a_t) r_{t+1} = \theta
\]
Linear Least Squares Policy Iteration (LSPI)

- LSPI-TD uses LSTDQ for policy evaluation
- It *repeatedly re-evaluates experience* $\mathcal{D}$ *with different policies*
- Obtain $\mathcal{D}$ with any probabilistic policy (e.g. random policy)

### LSPI-TD algorithm

<table>
<thead>
<tr>
<th>Given $\mathcal{D}$, initialize $\pi'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat</td>
</tr>
<tr>
<td>$\pi \leftarrow \pi'$</td>
</tr>
<tr>
<td>$\theta \leftarrow \text{LSTDQ}(\mathcal{D}, \pi)$</td>
</tr>
<tr>
<td>for each $s \in S$ do</td>
</tr>
<tr>
<td>$\pi'(s) = \arg \max_{a \in \mathcal{A}} \phi(s, a)^T \theta$</td>
</tr>
<tr>
<td>\quad // i.e. $\arg \max_{a \in \mathcal{A}} Q_\theta(s, a)$</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>until $\pi \approx \pi'$</td>
</tr>
<tr>
<td>return $\pi'$</td>
</tr>
</tbody>
</table>
## Convergence of LSPI

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Table Lookup</th>
<th>Linear</th>
<th>Non-Linear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monte-Carlo Control</td>
<td>✓</td>
<td>(✓)</td>
<td>X</td>
</tr>
<tr>
<td>Sarsa</td>
<td>✓</td>
<td>(✓)</td>
<td>X</td>
</tr>
<tr>
<td>Q-learning</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LSPI</td>
<td>✓</td>
<td>(✓)</td>
<td>-</td>
</tr>
</tbody>
</table>
Fitted Q-learning: Non-linear approximation
### Q-learning with FA

**Q-learning with FA**

<table>
<thead>
<tr>
<th>Q-learning with FA</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize parameters $\theta$ arbitrarily (e.g. $\theta = 0$)</td>
</tr>
<tr>
<td>for each episode do</td>
</tr>
<tr>
<td>Choose initial state $s$</td>
</tr>
<tr>
<td>repeat</td>
</tr>
<tr>
<td>Choose $a$ from $s$ using policy $\pi_\theta$ derived from $Q_\theta$ (e.g., $\epsilon$-greedy)</td>
</tr>
<tr>
<td>Execute action $a$, observe $r, s'$</td>
</tr>
<tr>
<td>$Q_\theta(s, a) \leftarrow Q_\theta(s, a) + \alpha (r + \gamma Q_\theta(s', \pi_\theta(s')) - Q_\theta(s, a) ) \nabla_\theta Q_\theta(s, a))$</td>
</tr>
<tr>
<td>$s \leftarrow s'$</td>
</tr>
<tr>
<td>until $s$ is terminal</td>
</tr>
<tr>
<td>end for</td>
</tr>
</tbody>
</table>
Problems with incremental Q-learning with FA

**Essence of off-policy learning.**

repeat
Choose \( a \), execute it and observe \( r \) and \( s' \) \((s, a, r, s')\) using any probabilistic policy
\[
Q_\theta(s, a) \leftarrow Q_\theta(s, a) + \alpha \left( r + \gamma Q_\theta(s', \pi_\theta(s')) - Q_\theta(s, a) \right) \nabla_\theta Q_\theta(s, a)
\]
\( s \leftarrow s' \)
until \( s \) is terminal

Several problems with incremental off-policy TD learning

- SGD does converge because gradient does not follow true gradient
- Target value is always changing and SGD does not converge
- Data is not even close to iid (it is strongly correlated) so another problem for SGD convergence

How to solve all these problems?
Let’s generalize the method:

Generalization of off-policy learning.

\[
\text{repeat} \\
\text{Choose from } D = \{\langle s, a, r, s' \rangle\} \text{ } N \text{ samples randomly. } D \text{ has been} \\
\text{obtained using any probabilistic policy} \\
\text{for } K \text{ times do} \\
\text{for each sample } i \text{ in } D \text{ do} \\
\quad y_i \leftarrow r + \gamma Q_\theta(s'_i, \max_a Q_\theta(s'_i a)) \\
\text{end for} \\
\theta \leftarrow \arg \min_\theta \sum (Q_\theta(s_i, a_i) - y_i)^2 \hspace{1cm} \text{// Any ML regression method} \\
\text{end for} \\
\text{until convergence}
\]
Generalization of off-policy learning

Notice several differences:

1. Sample a set of $N$ examples instead of only 1
2. Don’t use 1-step of gradient descent but compute exact solution (regression problem)
3. Repeat K times the Policy iteration method with the selected examples
Generalizarion of off-policy learning

Notice several differences:

1. Sample a set of \( N \) examples instead of only 1
2. Don’t use 1-step of gradient descent but compute exact solution (regression problem)
3. Repeat K times the Policy iteration method with the selected examples

Each difference improves convergence

1. Samples obtained randomly reduce correlation between them and stabilize Q value function for the regressor learner
2. Computation of exact solution avoid the true gradient problem
3. Repeat K times the Policy iteration method with the selected examples
Fitted Q-learning

- Implements fitted value iteration
- Given a dataset of experience tuple D, solve a sequence of regression problems
  - At iteration $i$, build an approximation $Q_i$ over a dataset obtained by $(TQ_{i-1})$
- Allows to use a large class of regression methods, e.g.
  - Kernel averaging
  - Regression trees
  - Fuzzy regression
- With other regression methods it may diverge
- In practice, good results also with neural networks
Fitted Q-learning

Given $\mathcal{D}$ of size $T$ with examples $(s_t, a_t, r_{t+1}, s_{t+1})$, and regression algorithm, set $N$ to zero and $Q_N(s, a) = 0$ for all $a$ and $s$

repeat

$N \leftarrow N + 1$

Build training set $TS = \{(s_t, a_t), r_{t+1} + \gamma \max_a Q_N(s_{t+1}, a)\}_{t=1}^T$

$Q_{N+1} \leftarrow \text{regression algorithm on } TS$

until $Q_N \approx Q_{N+1}$ or $N > \text{limit}$

return $\pi$ based on greedy evaluation of $Q_N$

- Works specially well for forward Neural Networks as regressors (Neural Fitted Q-learning)