8
Approximate Dynamic Programming

Approximate Dynamic Programming (ADP), also sometimes referred to as *neuro-dynamic programming*, attempts to overcome the limitations of value iteration in large state spaces where some *generalization* between states and actions is required due to computational and sample complexity limits. Further, all the algorithms we have discussed thus far require a strong *access model* to reconstruct the optimal policy from the value function and to compute the optimal value function at all.

Access Models

Implicitly up until now, we've largely assumed that we have complete, white box access to a full description of the system dynamics. For different reinforcement learning problems, there may be different levels of system access. For the Tetris problem we often assign as homework for this class, we can create the exact same state over and over again while learning (or testing our algorithms). For robotic systems, we have much less access – we can never create *exactly* the same state again. It's worth reviewing here some notions of access model to the system, as the techniques we can apply (and which will be most effective) are largely governed by the access model that is available to us for the development of a good policy.

1. Full Probabilistic Description

   In this model, we have access to the cost function and the transition function for every action. A downside of having this kind of model is that it can become so large as to be computationally intractable for any nontrivial problem. It is also information-theoretically hard to identify this type of model from data.

2. Deterministic Generative Model

   In this case, we have a function that maps $f(x, a) \rightarrow x'$, deterministically. Deterministic can mean that we have access to the random seed in a computer program, so we can recreate the same system including the randomness. Therefore, we can assume this kind of model for the upcoming Tetris assignment.

3. Generative Model

   In this model, we have programmatic access. We can put the system into any state we want.

4. Reset Model

   In this model, we can execute a policy or roll-out dynamics any time we want, and we can always reset to some known state or distribution over states. This is a good model for a robot in the lab that can be reset to stable configurations.

5. Trace Model

   This is the model that best describes the real world. Samuel Butler said "Life is like playing a violin solo in public and learning the instrument as one goes on"; the trace model captures the inability to "reset" in the real world.

There are a few general strategies one can pursue for applying approximation techniques.

Approximate the Algorithm. The most straightforward approach is to take the algorithms we've developed thus far *Policy Iteration* and *Value Iteration*, and replace the steps where we would update a tabular representation of the value function with a set of sampled (*state-action-next state*) and a supervised-learning *function approximator*.

This approach is an incredibly tempting way to pursue hard RL problems: we simply plug in a regression estimator and run existing, known-to-be-convergent algorithms. In a sense, we can see the tremendously successful Differential Dynamic Programming approach as of this form: we are finding quadratic approximations and running the existing value-iteration approach.

We find below that while at times successful in practice, there are many sources of *instability* in these algorithms that result in often extremely poor performance. We analyze informally the two main sources of error: the *bootstrapping* that happens in dynamic programming mixes poorly with generalization across states, and even more significantly, the change of policy induced by the max operation produces a *change in distribution* (affects which state-actions matter most) that dramatically amplifies any errors in the function approximation process. We discuss some strategies for remediating these.

Approximate the Bellman Equation. The next broad set of strategies is to treat the Bellman equation itself as a fixed point equation and optimize to find a fixed point. These techniques, known as *Bellman Residual Techniques* are dramatically more stable and have a richer theory.[1] [2] Practically, the performance is often (but not always!) worse then methods based on the "approximate the dynamic programming" strategy above, and it suffers as well from the *change of distribution* problem.

Approximate the Policy Alone. We cover a final approach that eschews the bootstrapping inherent in dynamic programming and instead caches policies and evaluates with rollouts. This is the approach broadly taken by methods like *Policy Search by Dynamic Programming* [2] and *Conservative Policy Iteration*[3]. [4]

[1] L. C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *International Conference on Machine Learning*, 1995; and Wen Sun, Geoffrey J Gordon, Byron Boots, and J Bagnell. Dual policy iteration. In *Advances in Neural Information Processing Systems*, 2018

[2] J. A. Bagnell and J. Schneider. Covariant policy search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2003

[3] S. Kakade and J. Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the 19th International Conference on Machine Learning (ICML)*, 2002

[4] Methods like the Natural Policy approach that we discuss later are closely connected.

Action-Value Functions

In this lecture, we consider the finite horizon case with horizon T. The *quality function*, Q-function, or *action-value function* is defined as,

$$Q^*(x,a,t) = c(x,a) + \text{total value received if optimal thereafter,}$$
$$Q^\pi(x,a,t) = c(x,a) + \text{total value received if following policy } \pi \text{ thereafter.}$$

These can be restated in terms of the Q-function itself as

$$Q^*(x,a,t) = c(x,a) + \gamma \mathbb{E}_{p(x'|x,a)}[\min_{a'} Q^*(x',a',t+1)]$$
$$Q^\pi(x,a,t) = c(x,a) + \gamma \mathbb{E}_{p(x'|x,a)}[Q^\pi(x',\pi(x'),t+1)]$$

Note that unlike infinite horizon case where a single value function/action-value function is defined, there are T value functions/action value functions for the finite horizon case, one for each time step.

Once we have the action-value functions, the value function V^* and the optimal policy π^* are easily computed as

$$V^*(x,t) = \min_{a \in \mathbb{A}} Q^*(x,a,t)$$
$$\pi^*(x,t) = \operatorname*{argmin}_{a \in \mathbb{A}} Q^*(x,a,t)$$

We can compare the above equation to how we compute the optimal policy from the optimal value function,

$$\pi^*(x,t) = \operatorname*{argmin}_{a \in \mathbb{A}} \; c(x,a) + \gamma \mathbb{E}_{p(x'|x,a)}[V^*(x',t+1)]$$

Pros and Cons of Action-Value Functions

Pros

1. Computing the optimal policy from Q^* is easier compared to extracting the optimal policy from V^* since it only involves an argmax and does not require evaluating the expectation and thus the transition model.
2. Given Q^*, we do not need a transition model to compute the optimal policy.

Cons

1. Action-value functions take up more memory compared to value functions (|States| x |Actions|, as opposed to |States|).[5]

[5] Note, however, that if we use a value function instead of Q-function, we may need another |States| x |Actions| table to store the transition probability in order to find the optimal policy if the transition model is not given analytically.

Fitted Q-Iteration

We can now describe Fitted Q-Iteration, an approximate dynamic programming algorithm that learns approximate action-value functions from a batch

of samples. Once the data is collected the Q-function is approximated without any interaction with the system.

Algorithm `FittedQIteration`($\{x_i, a_i, c_i, x'_i\}_{i=1}^n$)
  $Q(x, a, T) \leftarrow 0,\ \forall x \in \mathbb{X}, a \in \mathbb{A}$
  forall $t \in [T-1, T-2, \ldots, 0]$ **do**
    $D^+ \leftarrow \emptyset$
    forall $i \in 1, \ldots, n$ **do**
      input $\leftarrow \{x_i, a_i\}$
      target $\leftarrow c_i + \gamma \min_{a'} Q(x'_i, a', t+1)$
      $D^+ \leftarrow D^+ \cup \{\text{input}, \text{target}\}$
    end
    $Q(\cdot, \cdot, t) \leftarrow$ LEARN(D^+)
  end
  return $Q(\cdot, \cdot, 0 : T-1)$

Algorithm 10: Fitted Q-iteration with finite horizon.

The algorithm takes as input a dataset D which contains samples of the form {state, action, associated cost, next state}. In practice, this is obtained by augmenting expert demonstration data with random exploration samples. As in value iteration, the algorithm updates the Q functions by iterating backwards from the horizon $T-1$. Essentially, for each time step t, we create a training dataset D^+ by using the learned Q function learned for time step $t+1$. This dataset is fed into a function approximator LEARN, which could be any of your favorite machine learning algorithms (linear regression, neural nets, Gaussian processes, etc), to approximate the Q function from the training dataset. We could also start with an initial guess for $Q(\cdot, T)$. [6]

[6] The version presented here assumes the dynamics and cost functions are the same at each time-step.

Note that the above fitted Q-iteration algorithm can be easily modified to work for infinite horizon case. In fact, the infinite horizon version is simpler, because we can choose to maintain a single Q function. Hence, for each iteration, we can just collect a batch of samples, and update the Q function.

Algorithm `FittedQIteration`($\{x_i, a_i, c_i, x'_i\}_{i=1}^n$)
  $Q(x, a) \leftarrow 0,\ \forall x \in \mathbb{X}, a \in \mathbb{A}$
  while *not converged* **do**
    $D^+ \leftarrow \emptyset$
    forall $i \in 1, \ldots, n$ **do**
      input $\leftarrow \{x_i, a_i\}$
      target $\leftarrow c_i + \gamma \min_{a'} Q(x'_i, a')$
      $D^+ \leftarrow D^+ \cup \{\text{input}, \text{target}\}$
    end
    $Q \leftarrow$ UPDATE(Q, D^+)
  end
  return Q

Algorithm 11: Fitted Q-iteration with infinite horizon.

There are a few of things that we need to be aware of when using fitted Q-iteration in practice:

- In a goal-directed problem, we need to make sure that our samples include goal states in order to get meaningful iterations.
- Often it makes sense to run the algorithm on features of the state-action pair (x, a), not the raw state-action pair itself.
- Fitted Q-iteration can be run repeatedly, augmenting the data set with

new samples from the resulting policies.

- For goal-directed problems, the goal states x_i are nailed down to 0 Q-value (target = c_i), and bad or infeasible states are provided a large constant target value c^-. The former ensures that the Q-values do not drift up over time, and the latter prevents the Q-value for bad states from blowing up to ∞.

Example

An example of work that uses Fitted Q-Iteration was the paper "Learning to Drive a Real Car in 20 Minutes". Link below:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.3532&rep=rep1&type=pdf
Highlights:

- Learned a model from scratch for driving a car along a GPS guided course, minimizing cross-track-error (distance of vehicle to one side of a straight line between waypoints).
- The only data for learning came from actual driving.
- 3 layer neural net for regression.
- Action: steering discretized into 5 angle choices.

8.1 Challenges when using Fitted Q-Iteration

Unfortunately, while in the tabular case (maintaining a value for each state-action pair) the Q-function function converges [7] as the number of iterations of value-iteration (or policy iteration) increases to ∞, this does not generically hold under function approximation. The value function might converge, diverge, oscillate, or behave chaotically. Perhaps worse, meaningful bounds on the resulting performance of a policy learned using value function approximation can be either hard to obtain or vacuous.

[7] Under suitable assumptions discussed earlier.

Fitted Q-iteration and Fitted Value Iteration (a similar algorithm as fitted Q-iteration but approximates the value function), especially the infinite horizon version, is prone to bootstrapping issues in the sense that sometimes it does not converge. Since these methods rely on approximating the value function inductively, errors in approximation are propagated, and, even worse, amplified as the algorithm encourages actions that lead to states with sub-optimal values.

The key reason behind this is the minimization operation performed when generating the target value used for the action value function. The minimization operation pushes the desired policy to visit states where the value function approximation is less than the true value of that state. This typically happens in areas of state spaces which are relatively bad and have very few training samples. From a learning theory perspective, this can be viewed as a violation of the i.i.d assumption on training and test samples.

The following examples from [3] [Boyan and Moore, 1995] demonstrate this problem [8].

[8] All figures from Boyan et. al

Example: 2D gridworld

Figure 8.1.1 shows the 2D grid world example, which has a linear true value function J^*.

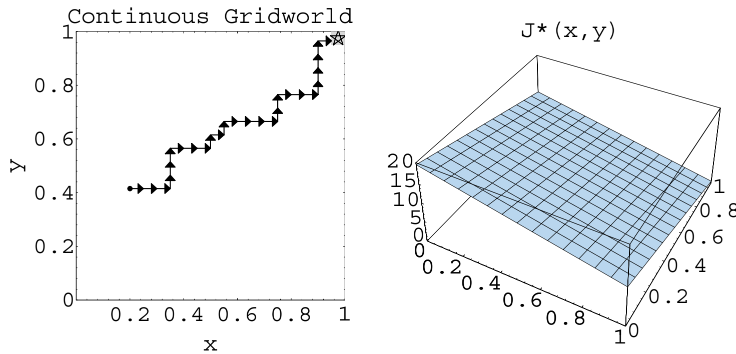

Figure 8.1.1: The continuous gridworld domain.

Figure 8.1.2 shows that VI with converges to the true value function.

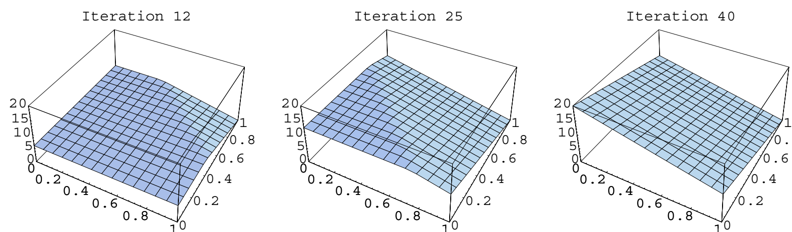

Figure 8.1.2: Training with discrete value iteration.

However, figure 8.1.3 shows that Fitted Value Iteration with quadratic regression fails to converge. The quadratic function, in trying to both be flat in the middle of state space and bend down toward 0 at the goal corner, must compensate by underestimating the values at the corner opposite the goal. These underestimates then enlarge on each iteration, as the one-step lookaheads indicate that points can lower their expected cost-to-go by stepping farther away from the goal.

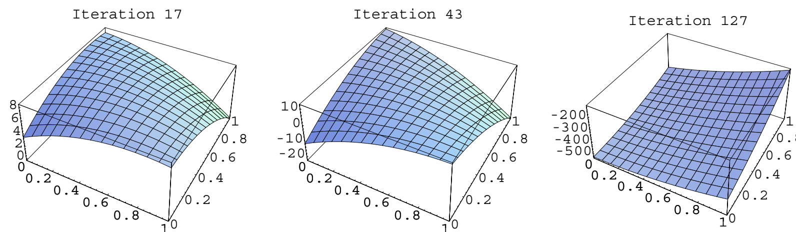

Figure 8.1.3: Training with quadratic regression. The value function diverges. Fitted Value Iteration with quadratic regression underestimates the values at the corner opposite the goal, and these underestimates amplify at each iteration.

Example: car on hill

Figure 8.1.4 shows the car-on-hill example.

Figure 8.1.5 shows that a two layer MLP can also diverge to underestimate the costs.

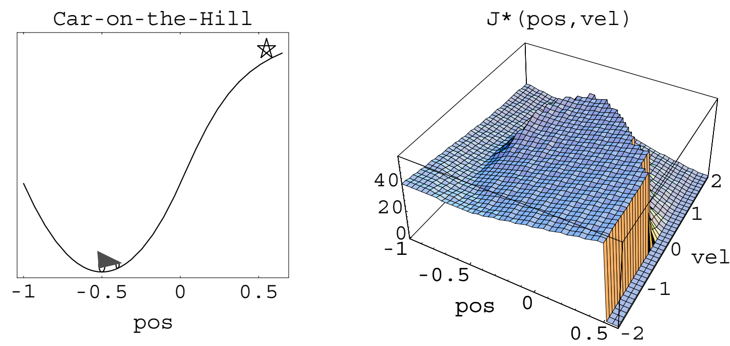

Figure 8.1.4: The car-on-the hill domain.

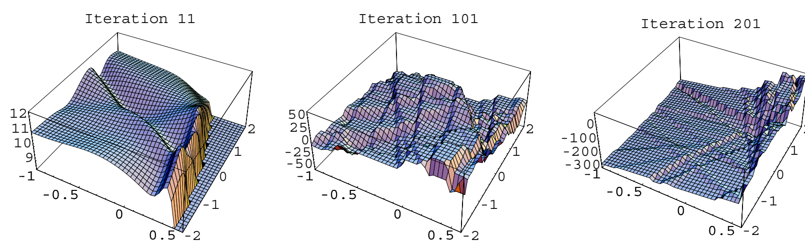

Figure 8.1.5: Training with neural network.

8.2 Approximate Policy Iteration

In the previous section we looked at how approximating the action-value function could lead to more efficient algorithms. The key idea in this section will be to approximate a policy from a batch of offline data and improve it by iterating over the data samples. As we will see, the process of evaluating a policy will be more stable compared with fitted value iteration as the min operation will no longer be used in the training loop. As with policy iteration, there are two fundamental steps involved in approximate policy iteration process - policy evaluation and policy improvement.

Policy Evaluation

Algorithm `Evaluate(`$\{x_i, a_i, c_i, x_i'\}_{i=1}^n$`)`
- $Q^\pi(x, a, T) \leftarrow 0,\ \forall x \in \mathbb{X}, a \in \mathbb{A}$
- **forall** $t \in [T-1, T-2, \ldots, 0]$ **do**
  - $D^+ \leftarrow \emptyset$
  - **forall** $i \in 1, \ldots, n$ **do**
    - input $\leftarrow \{x_i, a_i\}$
    - target $\leftarrow c_i + \gamma Q^\pi(x_i', \pi(x_i', t+1), t+1)$
    - $D^+ \leftarrow D^+ \cup \{\text{input}, \text{target}\}$
  - **end**
  - $Q^\pi(\cdot, \cdot, t) \leftarrow \textsc{Learn}(D^+)$
- **end**
- **return** $Q^\pi(\cdot, \cdot, 0 : T-1)$

Algorithm 12: Approximate policy evaluation with finite horizon

In Algorithm 12, the stability of the function approximation comes from that fact that we are interested in approximating Q^π and not Q^*. This kind of stability often turns out to be critical, and many practical RL implentations favor a policy iteration variant. [9]

[9] Similar to fitted Q-iteration, there is also an infinite horizon version of the above algorithm.

Algorithm `Evaluate(`$\{x_i, a_i, c_i, x_i'\}_{i=1}^n, \pi$`)`
- $Q^\pi(x, a) \leftarrow 0,\ \forall x \in \mathbb{X}, a \in \mathbb{A}$
- **while** *not converged* **do**
  - $D^+ \leftarrow \emptyset$
  - **forall** $i \in 1, \ldots, n$ **do**
    - input $\leftarrow \{x_i, a_i\}$
    - target $\leftarrow c_i + \gamma Q^\pi(x_i', \pi(x_i'))$
    - $D^+ \leftarrow D^+ \cup \{\text{input}, \text{target}\}$
  - **end**
  - $Q^\pi \leftarrow \textsc{Update}(Q^\pi, D^+)$
- **end**
- **return** Q^π

Algorithm 13: Approximate policy evaluation with infinite horizon

Function approximation induces very significant problems in computing good policies or value functions. Lets take a closer look at the problems that result.

Function Approximation Divergence

We consider now the more stable variant– function approximation of the policy evaluation step alone– rather than the more complex (non-linear) value iteration variant. [10] Even here, Tsitsiklis and Van Roy [6] demonstrate that without care, function approximation will behave poorly.

[10] Below we'll discuss that the more difficult to manage problems come from the changing the policy.

Consider the MDP in Figure 8.2.1 has two states S1 and S2. The following details the setup:

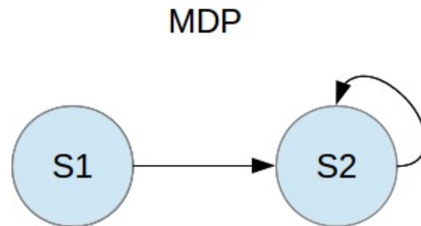

Figure 8.2.1: Two state MDP

1. The reward for being at any state (hence the true value function) is given by $\{0, 0\}$
2. Consider a discount factor $\gamma = 0.9$
3. The feature $\{x\}$ is simple the numerical value of the state $\{1, 2\}$
4. The value function is approximated with linear function: $V(s) \leftarrow w^T x$

The graphical view of the value function approximation is shown in Figure 8.2.2. Since the reward is always 0, we know the true value function is $\{0, 0\}$. This corresponds to $w = 0$. We will now examine if the approximation converges to this value.

Lets start with $w = 1$. One round of value iteration yields the following target values for the function approximator

$$\begin{aligned} V_\pi(s) &= r(s, \pi(s)) + \gamma V(s') \\ V(s1) &\leftarrow 0 + \gamma w * 2 = 1.8 \\ V(s2) &\leftarrow 0 + \gamma w * 2 = 1.8 \end{aligned}$$

If a least squares approach is used to fit to this data, we'd arrive at $w = 1.2$. Repeated iteration eventually results in the function approximator blowing up exponentially in iterations.

Some Remedies for Divergence

If the training data is weighted by how much time one visits that state, then divergence problem can be arrested *for linear function approximators*. In our example, if we spend $t = 1$ time-steps in S1, then we spend $\frac{\gamma}{1-\gamma} = 9$ time-steps at S2. If this is used as a weight in the weighted least squares fitting, then after the first iteration $w = 0.92$, i.e, it proceeds towards the correct value 0. This *on-policy* weighting, where the loss is weighed by the time spent in each state can be demonstrated to ensure convergence. Unfortunately, the same result does not hold for a more general class of function approximators. [6] An entire literature has grown up around attempts to maintain the advantages of approximating the dynamic programming iterations while ensuring

convergence in more general settings. Rich Sutton and Andy Barto's book [11] extensively covers these efforts.

[11] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998

Another approach is a method of "Averagers" [4]. Its a class of function approximators of the form $V(x) = \sum_i \beta_i J(y_i)$ $(0 \leq \beta \leq 1, \sum_i \beta_i \leq 1)$. This was essentially putting a grid over the state space and converting the problem to a MDP on the grid. There exists exact solution to Value iteration on the grid. The value function anywhere else is computed by interpolating value function from the grid.

The upside to the approach is since the function approximator does not extrapolate, the divergence problem does not occur. The downside is a relatively weak class of function approximators.

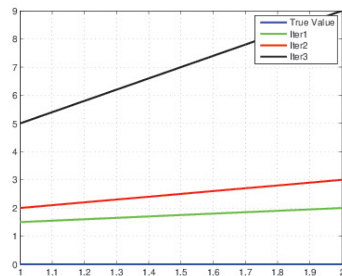

Figure 8.2.2: Approximate Value Function Iteration

Policy Improvement

This is the second step in the Approximate Policy Iteration process. We select a policy by simply acting greedily with respect to the estimated Q-function

$$\pi'(x, t) = \arg\min_a Q^{\pi}(x, a, t) \tag{8.2.1}$$

The Core Problem of Approximate Dynamic Programming

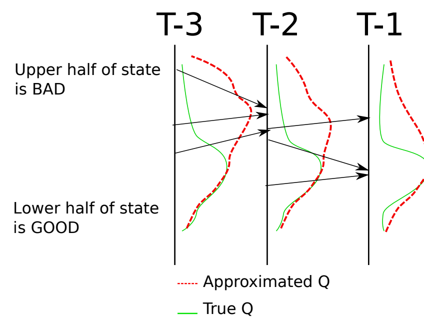

Figure 8.2.3: Value function overestimation in value iteration

We discussed before the problem of value function approximation overestimating how good it thinks a state is and this error amplifying as it proceeds backwards in time. Figure 8.2.3 shows an illustration of this effect. Because the upper half of the state space (which is bad) is overestimated by the function approximator, policies switch to direct towards that state. Error in overestimation of the value function has a cascading effect as we iterate backwards in time.

However, until now, our focus has been on *boostrapping* and approximating the value function either of a given, or optimal, policy. Can we use the core idea of dynamic programming without bootstrapping values? A quote from Richard Bellman sheds light on this issue

> "An optimal policy has the property that whatever the initial decision may be, the remaining decisions constitute an optimal policy .... [for the resulting state]"

This idea is related to the monotonic improvement of policy iteration. If we cache the policies and re-estimate the value function at every iteration backwards in time, we avoid the overestimation problem discussed above as we get unbiased estimates of the costs that will occur in the future.

Policy Search by Dynamic Programming

As is standard in dynamic programming, we proceed backwards (over a finite horizon) from $T-1$. At iteration $T-\tau$, instead of memoizing (approximately) a value function in the future and bootstrapping from that, we memoize just the policies in the future and rollout the result all the way to $T-1$. A new policy is learned via estimating an action-value function at $T-\tau$. [12] for a single time step given the rollouts. A new policy is installed at the time-step $T-\tau$. Effectively, instead of approximating Q^t we approximate $\pi^{*,t}$. Let's walk through how this works below.

[12] Or, often powerfully, simply optimizing the policy directly

Time T − 1:

We approximate $\tilde{\pi}^{*,T-1}(x) = \arg\min_a c(x,a)$ either analytically or via sampled states and actions from a (for now fixed) distribution $d(s,a,T-1)$. This becomes the optimal policy at $T-1$.

Time T − 2:

For an input pair $\{x_i, a\}$, the target value is $c_i + \gamma c(x', \pi^{*,T-1}) + \ldots$. So an error in approximation of π does not bootstrap, it shows up as the policy is always evaluated. [13]

[13] While described here as a regression problem, it is equally possible to directly learn classifiers that minimize a weighted 0/1 loss.

8.3 *Related Reading*

[1] Ernst, Damien, Pierre Geurts, and Louis Wehenkel, *Tree-based batch mode reinforcement learning*. Journal of Machine Learning Research 2005.

[2] Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. In Machine Learning Proceedings 1995 (pp. 30-37). Morgan Kaufmann.

[3] Boyan, Justin A and Moore, Andrew W, *Generalization in Reinforcement Learning: Safely Approximating the Value Function*. NIPS 1994.

[4] Gordon, Geoffrey J, *Stable function approximation in dynamic programming*. DTIC Document 1995.

[5] Bagnell, J. A., Kakade, S. M., Schneider, J. G., and Ng, A. Y. (2004). Policy search by dynamic programming. NIPS, 2004.

[6] J. N. Tsitsiklis and B. Van Roy, *An Analysis of Temporal-Difference Learning with Function Approximation*, IEEE Transactions on Automatic Control, Vol. 42, No. 5, 1997.