8
Approximate Dynamic Programming

Approximate Dynamic Programming (ADP), also sometimes referred to as *neuro-dynamic programming*, attempts to overcome the limitations of value and policy 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. We'll consider algorithms in the rest of these lecture notes that relax this strong notion of access.

Access Models

Up until this chapter, we've implicitly largely assumed that we have complete, white box access to a full description of the system dynamics for the purposes of applying dynamic programming. In practice, reinforcement learning problems differ by the degree of "system access" that is available. For the Tetris problem we often assign as homework for this class, we can recreate the exact same state over and over again while learning (or testing our algorithms). For robotic systems, we typically have a weaker form of access – we can never create *exactly* the same state again, but we can often run multiple trials. It's worth reviewing here some notions of access model for a system as the techniques we can apply and which will be most effective are largely governed by this access. We review them in order of the strength of the model access; each of the earlier access models can trivially simulate the ones below it, but not (necessarily) visa-versa.

1. Full Probabilistic Description

   In this model, we have access to an explicit cost function and the transition function for every state-action written down as a large matrix that can be manipulated. A major downside of having this kind of model is that it easily can become so large as to be computationally intractable for a non-trivial problem. It is also information-theoretically hard to identify this type of model from data– it simply isn't possible to visit a very large state-space.

2. Deterministic Simulative Model In the simplest version, we have a function that maps $f(x, a) \rightarrow x'$, deterministically. [1] More broadly, a deterministic simulative model can mean that we while the dynamics are stochastic, we have access to the random seed in a computer program, so

[1] We'll assume through the discussion below that the cost function has the same access. It can be the case, however, that we can be "in between" such models. For instance, we might have a more complete description of a cost function as a quadratic, while only having *reset model* access to the dynamics.

we can recreate trajectories including the randomness that occurred. Such access is what is typically available in computer simulations. [2]

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.

[2] Although, unfortunately, non-determinism in simulators is more prevalent than one might expect.

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.[3] [2] Practically, the performance is often (but not always!) worse than 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* [4] and *Conservative Policy Iteration*[5]. [6]

[3] 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

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

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

[6] Methods like the Natural Policy gradient 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|).[7]

[7] 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 10: Fitted Q-iteration with finite horizon.

Algorithm `FittedQIteration`$(\{x_i, a_i, c_i, x_i'\}_{i=1}^n, T)$
  $Q(x, a, T) \leftarrow 0,\ \forall x \in \mathbb{X}, a \in \mathbb{A}$
  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
    $Q(\cdot, \cdot, t) \leftarrow \textsc{Learn}(D^+)$
  end
  return $Q(\cdot, \cdot, 0 : T-1)$

The algorithm takes as input a data-set 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 data-set D^+ by using the learned Q function learned for time step $t+1$. This data-set is fed into a function approximator LEARN, which could be any of your favorite machine learning models (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 $Q(\cdot, T)$. [8]

[8] 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 11: Fitted Q-iteration with infinite horizon.

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
    $Q \leftarrow \textsc{Update}(Q, D^+)$
  end
  return Q

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 pairs themselves.

- 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 ∞.
- Value functions are not smooth in general (e.g. mountain car problem). A simple trick to fix this is to add noise to the transition model, which smooths out discontinuities.

Case Study

A robotics example of work using Fitted Q-Iteration is demonstrated in [9]

[9] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.3532&rep=rep1&type=pdf

The authors demonstrate that a neural (in modern parlance, "deep") fitted Q-learning algorithm can learn a control strategy 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). All data for learning came from actual driving; i.e. there is neither a model nor a use of data-augmentation. As has been, perhaps surprisingly, common in robotics, the actual network is a relatively shallow 3 layer neural net for regression. This contrasts with work for imitation learning of driving controllers like that of [10] where very deep networks were used.

[10] M. Nechyba and J. A. Bagnell. Stabilizing human control strategies through reinforcement learning. In *Proc. IEEE Hong Kong Symp. on Robotics and Control*, volume 1, pages 39–44, April 1999

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 [11] as the number of iterations of value-iteration (or policy-iteration) steps 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.

[11] Under suitable assumptions discussed earlier.

Fitted Q-iteration and Fitted Value Iteration (a similar algorithm as fitted Q-iteration but approximates the value function and counts on a model to find optimal controls), 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– that is to say, states that look more attractive than they should. This typically happens in areas of state spaces which are very few training samples and could, in fact, be quite bad places to arrive. 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 [12].

[12] 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^*.

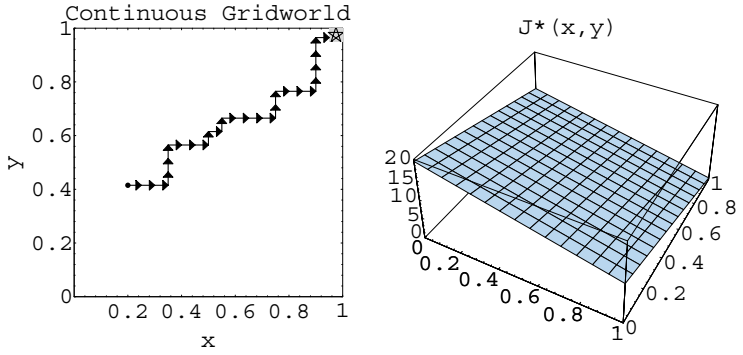

Figure 8.1.1: The continuous gridworld domain.

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

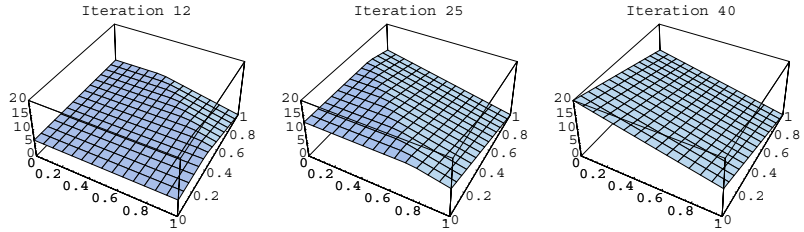

Figure 8.1.2: Training with discrete value iteration.

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.

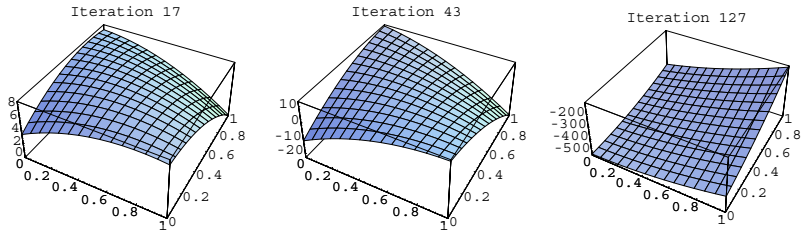

Figure 8.1.3: Training with quadratic regression. The value function diverges. Fitted Value Iteration with quadratic regression underestimates the values at the corner opposite the goal, and these underestimates amplify at each iteration.

Example: car on hill

Figure 8.1.4 shows the car-on-hill example.

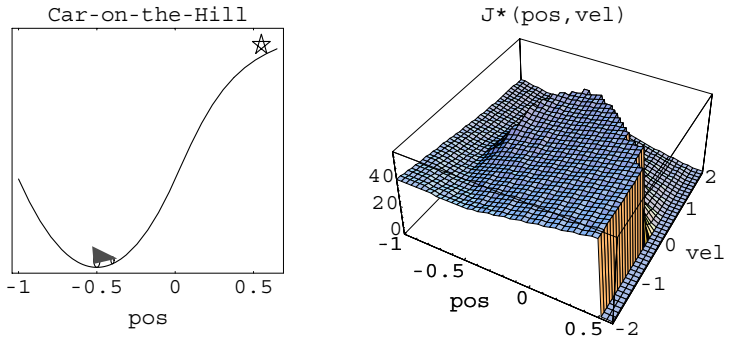

Figure 8.1.4: The car-on-the hill domain.

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

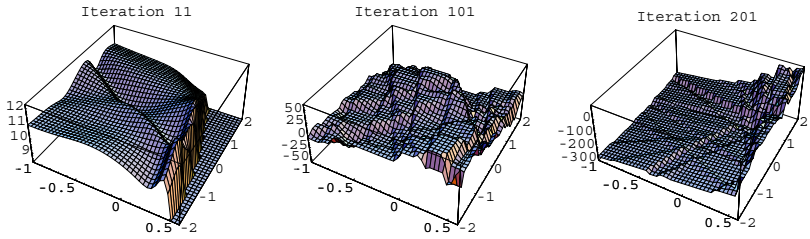

Figure 8.1.5: Training with neural network.

8.2 *Approximate Policy Iteration*

In the previous section we looked at how approximating the action-value function can potentially be effective in large state spaces. In this section, we'll consider approximating the action-value function for a policy from a batch of offline data and then improving that policy. The process of evaluating a policy will be more stable compared with fitted Q 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. We'll consider how *trust-region* and *line search techniques* can control the change of distribution problems that results when we update the policy later in later lectures.

Policy Evaluation

Algorithm 12: Approximate policy evaluation with finite horizon

Algorithm `Evaluate`$(\{x_i, a_i, c_i, x'_i\}_{i=1}^n, T)$
- $Q^\pi(x, a, T) \leftarrow 0,\ \forall x \in \mathbb{X}, a \in \mathbb{A}$
- **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** $Q^\pi(\cdot, \cdot, 0 : T-1)$

In Algorithm 12, improved 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 implementations favor a policy iteration variant. Naturally, there is also a "batch"

infinite-horizon version of the above algorithm:

Algorithm 13: Approximate policy evaluation with infinite horizon

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

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) Q-iteration variant. [13] Even here, Tsitsiklis and Van Roy [6] demonstrate that without care, function approximation has the potential to behave very poorly.

[13] 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:

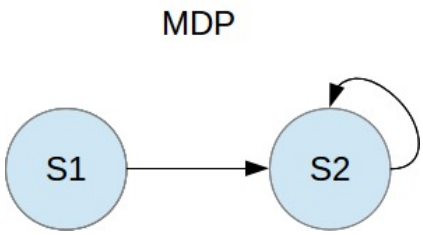

Figure 8.2.1: Two state MDP

1. The reward for being at any state (hence the true value function) is $\{0, 0\}$
2. Consider a discount factor $\gamma = 0.9$
3. The feature $\{x\}$ is simply 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.

Let's start with $w = 1$. One round of value iteration yields the following

target values for the function approximator

$$V_\pi(s) = r(s, \pi(s)) + \gamma V(s')$$
$$V(s1) \leftarrow 0 + \gamma w * 2 = 1.8$$
$$V(s2) \leftarrow 0 + \gamma w * 2 = 1.8$$

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 the number of iterations iterations/number of backups that are performed. [14]

[14] One might hope that the finite horizon variant might not suffer from divergence in this example. That is technically correct (observed by Wen Sun), but is ineffective as the error instead simply grows exponentially in horizon length).

Some Remedies for Divergence

If the training data is weighted by how much time the agent visits a 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 weighted 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. Sutton and Barto's book [15] extensively covers these efforts and is highly recommended.

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

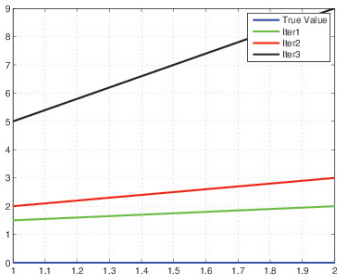

Figure 8.2.2: Approximate Value Function Iteration

Policy Improvement

The second step of the Approximate Policy Iteration process is to update or *improve* the policy. We select a new policy by simply acting greedily with respect to the estimated Q-function of the old one:

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

In API we have moved the dominant form of instability to *this* step of the process.

The Central Problem of Approximate Dynamic Programming

We discussed before the problem of value function approximation overestimating how good it thinks a state is, and then this error amplifying as

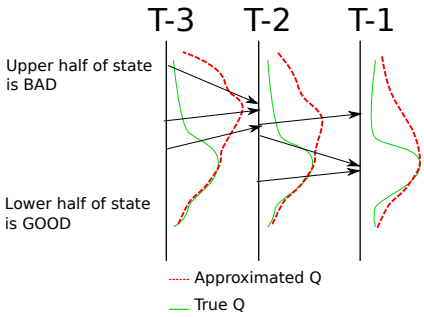

Figure 8.2.3: Value function overestimation in value iteration

Bellman backups proceed. 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 probability mass towards that state by choosing actions that make arriving at these states more likely. Error in overestimation of the value function has a cascading effect as we iterate backwards in time.

We further noted that the pure policy evaluation variant of dynamic programming is much more stable– without the *max* to drive behavior towards states with high value estimates we are less subject to the amplification of errors. However, on the surface it seems that we've merely pushed the problem into the *policy improvement* step. That is, while the estimation of the action-value function for a current policy becomes stable, the improvement step would instead drive probability mass towards states-actions that tend to be over-estimates of quality, leading to instability between iterations of any approximate policy iteration procedure.

This objection is, in fact, well-founded and approximate policy iteration algorithms aren't noted to be more stable or effective than approximate value iteration counterparts. However, the maintenance of an explicit policy opens up a new possibility: the ability to manage or mitigate the distribution shift that occurs when we update the policy.

Conservativity and Trust Regions

A broad class of algorithms, initiated by the seminal development of *Conservative Policy Iteration (CPI)* [16] constrain modification to the current policy to prevent the state-action distribution from changing too radically between iterations and thus ensure errors don't explode. The result is algorithms that are stable and effective, although they can be slower than raw policy iteration. CPI modifies the policy update step to *stochastically mix* [17] between policies $\pi^{\text{new}} = \alpha\pi^{\text{new greedy}} + (1-\alpha)\pi^{\text{old}}$, where the mixing weight α is interpreted as the probability of choosing that component. Careful analysis in [18] ensures a strategy for choosing α that ensures improvement, while in practice a simple *line-search* strategy can be employed to ensure monotonic improvement.

This is a somewhat impractical algorithm as it can take many steps and requires maintenance of a mixture of a number of policies equal to the number of update steps. Later approaches, including *No-Regret Policy Iteration* [19] and the Natural or Covariant Policy Search Approach [20] (and later implementations of these like "Trust Region Policy Optimization" [21]) manage to keep one policy, albeit a typically stochastic one, but keep the same intuition of a controlled policy change through the stability of no-regret learning, line

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

[17] That is to say, choose with that probability at each time-step of execution of the policy.

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

[19] S. Ross and J. A. Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *arXiv preprint arXiv:1406.5979*, 2014

[20] ; and J. A. Bagnell, A. Y. Ng, S. Kakade, and J. Schneider. Policy search by dynamic programming. In *Advances in Neural Information Processing Systems*, 2003

[21]

search or trust-region constraints. We defer discussion of these methods to the *Policy Search* chapter **??**.

Interestingly, there becomes no clear line between the modern, controlled *Approximate Policy Iteration* algorithms and algorithms that are variants of *Policy Gradient*. When an action-value function estimator is used that "bootstraps" using Bellman updates, we tend to view them as *API* algorithms. When the updates are made using pure roll-out estimates, we tend to view them as "policy gradient" algorithms. In practice, the distinction in practical use of the terms is somewhat artificial.

8.3 Policy Search by Dynamic Programming

Our focus thus far has been on Bellman *bootstrapping* and approximating the value function either of a given, or optimal, policy. Can we use the core idea of dynamic programming *without* bootstrapping values? Richard Bellman's thoughts shed some 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]"

The central idea of dynamic programming is that it does not matter how a decision maker arrives at a state; rather, what matters is that given arrival at the state, the decision maker chooses optimally thereafter. This insight allows us to solve problems recursively. This notion is related to the monotonic improvement of policy iteration. If we *cache policies* and re-estimate the value function at every iteration backwards in time, we avoid the over-estimation and compounding errors problem discussed above as we get unbiased estimates of the real costs that will occur in the future, and errors are not amplified as we proceed through iterations.

Let's try and make this intuition about caching policies concrete. 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 "roll-out" the total cost of an action and future policy decisions all the way to $T-1$. A new policy is learned via estimating an action-value function at $T-\tau$. [22] for a single time step given the rollouts. A new policy is installed at the time-step $T-\tau$. Let's walk through how this works below.

[22] Or, often more powerfully, simply optimizing the policy directly to choose actions with high future returns.

A Sketch of an Algorithm

Let's see what it might look like to use dynamic programming *without* memoizing values:

Time T − 1:

We can approximate $\tilde{\pi}^{*,T-1}(x) = \arg\min_a c(x,a)$ either analytically or via sampled states from a (for now fixed) distribution $\mu_{T-1}(s)$ which we'll call the *baseline distribution*. We'll assume for now that actions are simply chosen uniformly at random [23]. This forms our approximation of the optimal policy at $T-1$.

[23] Or that all of them are tried instead for a given state! That has lower variance, but requires a reset access model that lets us return to the exact same state.

Time T − 2:

For any sampled input pair $\{x_i, a_i\}$, the target value is $c(x_i, a_i) + c(x', \pi^{x',T-1})$. So an error in approximation of π does not bootstrap, it shows up as the policy is always evaluated honestly. However, we again need to specify a distribution of states to optimize with respect to, $\mu_{T-2}(s)$, and given these samples can attempt to find a one-step optimal policy $\tilde{\pi}^{*,T-2}(x)$ that minimizes the average cost under the distribution of samples.

Similarly now for any k, (starting with k=2 and moving backwards in time) we can compute: **Time T − k:**

For a sampled input pair $\{x_i, a_\}$, the target value is $c(x_i, a_i) + c(x', \pi^{x',T-k+1}) + c(x'', \pi^{x',T-k+2} + \cdots$.

Note that this approach address the problem of learning over an *exponentially* large set of policies, but it does with a *quadratic* in horizon T dependence, rather than the *linear* in T dependence that is achieved by policy or value iteration. [24]

[24] Of course, whether this is cost is "worth" it or not depends on both the horizon length and how much errors are amplified by backups.

The Baseline distribution

Note that the algorithm requires a distribution $\mu_t(s)$ from which to draw sample states (as do any of the batch fitted iteration methods). This presents something of a *chicken-or-the-egg* situation as intuitively (and which we'll quantify below) we'd like to sample states from where the optimal policy would visit. This kind of requirement of having an idea of where to sample states is fairly common though: The PSDP approach was partially inspired by the work of [25] and by differential dynamic programming (DDP) generally where policies are generated using as input an initial sample of trajectories. An insight provided by that paper is the usefulness of having information regarding where good policies spend time. This can come from a heuristic initial policy or demonstration by an "expert" at a task. The idea of *baseline distribution* is the natural probabilistic generalization of an *initial trajectory*.

[25]

In essence, the baseline distribution tells our learners where to focus their effort, and shortcuts the the difficulties of *global exploration*. We count on the reset access model and the baseline distribution to solve the hard problem of identifying and getting to states that really matter. We'll spend more time in later lectures discussing baseline distributions and exploration as the general idea of leveraging a baseline distribution is a powerful "cheat" that is equally applicable to policy search/gradient methods as it is to the dynamic programming ones mentioned here.

PSDP as classification

With the crude sketch of a meta-algorithm in hand, we can consider a natural instantiation of PSDP using calls to a supervised learning classification

algorithm.

Algorithm 14: PSDP using classification

Data: Given weighted classification algorithm C and baseline distribution $\mu_t(s)$.
foreach $t \in T-1, T-2, \ldots, 0$ **do**
  Sample a set of n states s_i according to μ_t:
  foreach s_i **do**
    foreach *a in* $\mathcal{A}$ **do**
      Estimate $Q_{a,\pi_{t+1}\ldots,\pi_{T-1}}(s_i)$ by rolling out a trajectory of length $T-t$ starting from s_i and using the action a on the generative model. At each time step after the initial one use the previously computed (near-optimal) policies to choose actions.
    Compute (dis)advantages: For each sampled state s_i, compute the best (highest value) action and denote it a_i^*.
  Create a training set L consisting of tuples of size $|\mathcal{A}|n$: $\langle s_i, a, Q_{a_i^*,\pi_{t+1}\ldots,\pi_{T-1}}(s_i) - Q_{a,\pi_{t+1}\ldots,\pi_{T-1}}(s_i)\rangle$
  Set $\pi_t = C(L)$, where the classifier C is attempting to minimize the weighted 0/1 loss, or an appropriate surrogate.

In the algorithm above we have replaced idealized expectations with Monte Carlo estimates and what it naturally an optimization over one-step policies by a call to an arbitrary supervised-learning algorithm. If we can perform the supervised learning task at each step well we will achieve good performance at the RL task *at least relative to the baseline distribution*.

Action-value approximation via regression. A particular variant of the sample based PSDP above can be implemented if it is possible to efficiently find an approximate action-value function $\tilde{Q}_{a,\pi_{t+1}\ldots,\pi_{T-1}}(s)$, *i.e.*, if at each time-step we can ensure that $\epsilon \geq \mathrm{E}_{s\sim\mu_t(s)}[\max_{a\in A}|\tilde{Q}_{a,\pi_{t+1}\ldots,\pi_{T-1}}(s) - Q_{a,\pi_{t+1}\ldots,\pi_{T-1}}(s)|]$.

(Recall that the policy sequence $(a, \pi_{t+1} \ldots, \pi_{T-1})$ always begins by taking action a.) If the policy π_t is greedy with respect to the estimated action value $\tilde{Q}_{a,\pi_{t+1}\ldots,\pi_{T-1}}(s)$, then we can show that this induces a policy that is within $2T\epsilon$ [26] of choosing the optimal action according to the distribution μ. It is important to note that this error is phrased in terms of an *average error* over state-space, as opposed to the worst case errors over the state space that are more standard in dynamic programming algorithms and drive the instabilities of those approaches. We can intuitively grasp this by observing that value iteration style algorithms may amplify any small error in the value function by pushing more probability mass through where these errors are. PSDP, however, as it does not use value function backups, cannot make this same error; the use of the computed policies in the future keeps it honest. There are numerous efficient regression algorithms that can minimize this, or approximations to it.

[26] Think about where the 2 comes from here!

"Convergence" and Partial Observability

Note that even as the time horizon we consider gets very large when we are in the function approximation setting, Q does **not** necessarily converge as

$k \to T$ even when it would in the full tabular setting. To see why this might be so, consider an extreme example.

Imagine a hypothetical situation of making a two legged robot walk. Further, imagine we limit our policy to have only a single feature given to the function approximation: the time-step t — *i.e.* no description of state whatever. As demonstrated in [27], an algorithm like PSDP can actually learn a sequence of open loop torques that make the robot perform an effective, albeit brittle, walking motion. The value of choosing some action will be *very different* even at neighboring time-steps, of course, because we're encoding an open-loop strategy here. Generically, this is true: we can get different Q-functions at neighboring time steps– this is a strong indication, however, of aliasing of real underlying states.

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

Algorithm 15: Iterated Policy Search by Dynamic Programming

Algorithm `Iterated-PSDP(`π`)`

Start with arbitrary time-varying policy π_0
$k \leftarrow 0$
while *not converged* **do**
  for $\forall x \in \mathbb{X}, t \in \{0, \ldots, T-1\}$ **do**
    $\pi_{k+1}(x,t) \leftarrow \operatorname{argmin}_{a \in \mathbb{A}} Q^{\pi_k}(x,a,t)$ Collect samples $\{\{x_t^{(i)}, a_t^{(i)}, c_t^{(i)}, x_{t+1}^{(i)}\}_{t=0}^{T-1}\}_{i=1}^{n}$ by executing policy π_k
    $Q^{\pi_k+1} \leftarrow$ `Learn(`$\{\{x_t^{(i)}, a_t^{(i)}, c_t^{(i)}, x_{t+1}^{(i)}\}_{t=0}^{T-1}\}_{i=1}^{n}, \pi_k$`)`
  end
  $k \leftarrow k+1$
end
return $\pi_k(x), \forall x$

Understanding Performance Guarantees

There are multiple possible bounds that can be established for the algorithms above. Perhaps the most insightful on is a bound on performance with a multiplicative dependence on average error in the case that we are willing to compare ourselves against *all* policies rather than expressing regret with respect to a limited class. [28] We state the bound as follows:

[28] Although the search in PSDP may, of course, still be conducted over the limited class.

Theorem 2 (MDP Performance Bound)**.** *Let $\pi = (\pi_0, \ldots, \pi_{T-1})$ be a non-stationary policy returned by an ε-approximate version of PSDP in which, on each step, the policy π_t found comes within ε of maximizing the value over* all *policies. I.e.,*

$$\mathrm{E}_{s \sim \mu_t}[V_{\pi_t, \pi_{t+1}, \ldots, \pi_{T-1}}(s)] \geq \max_{\pi'} \mathrm{E}_{s \sim \mu_t}[V_{\pi', \pi_{t+1}, \ldots, \pi_{T-1}}(s)] - \varepsilon\,. \tag{8.3.1}$$

Then for all possible policies (including including the optimal one) π_{ref} and its induced distribution over states $\mu_{\pi_{\mathrm{ref}}}$ we have that

$$V_\pi(s_0) \geq V_{\pi_{\mathrm{ref}}}(s_0) - \sum_t \varepsilon || \frac{\mu^t_{\pi_{\mathrm{ref}}}}{\mu^t} ||_\infty$$

where the infinity norm refers to the sup *of state space.*

We sketch a proof here. It is cleanest to apply the *Performance Difference Lemma* we developed earlier, but there is a certain value to understanding the induction that is being applied directly. [29] **Proof:** It is enough to show that for all $t \in T-1, T-2, \ldots, 0$,

[29] Exercise: Apply the performance difference lemma for a simplified version of this proof.

$$\mathrm{E}_{s\sim\mu^t_{\pi_{\mathrm{ref}}}}[V_{\pi_{\mathrm{ref}}{}^t,\ldots}(s) - V_{\pi^t,\ldots}(s)] \leq \sum_{\tau=t}^{T-1} \varepsilon ||\frac{\mu^\tau_{\pi_{\mathrm{ref}}}}{\mu^\tau}||_\infty$$

This again follows by induction, the inductive step being the non-trivial part:

$$\begin{aligned}
& \mathrm{E}_{s\sim\mu^t_{\pi_{\mathrm{ref}}}}[V_{\pi_{\mathrm{ref}}{}^t,\ldots}(s) - V_{\pi^t,\ldots}(s)] \qquad (8.3.2)\\
= \;& \mathrm{E}_{s\sim\mu^t_{\pi_{\mathrm{ref}}}}[V_{\pi_{\mathrm{ref}}{}^t,\ldots}(s) - V_{\pi_{\mathrm{ref}}{}^t,\pi^{t+1}\ldots}(s)] + \mathrm{E}_{s\sim\mu^t_{\pi_{\mathrm{ref}}}}[V_{\pi_{\mathrm{ref}}{}^t,\pi^{t+1}\ldots}(s) - V_{\pi^t,\ldots}(s)]\\
\leq \;& \mathrm{E}_{s\sim\mu^{t+1}_{\pi_{\mathrm{ref}}}}[V_{\pi_{\mathrm{ref}}{}^{t+1},\ldots}(s) - V_{\pi^{t+1}\ldots}(s)] + \mathrm{E}_{s\sim\mu^t_{\pi_{\mathrm{ref}}}}[\max_{a_t} Q_{a^t,\pi^{t+1}\ldots}(s) - V_{\pi^t,\ldots}(s)]\\
\leq \;& \mathrm{E}_{s\sim\mu^{t+1}_{\pi_{\mathrm{ref}}}}[V_{\pi_{\mathrm{ref}}{}^{t+1},\ldots}(s) - V_{\pi^{t+1}\ldots}(s)] + \mathrm{E}_{s\sim\mu^t_{\pi}}[\max_{a_t} Q_{a^t,\pi^{t+1}\ldots}(s) - V_{\pi^t,\ldots}(s)]||\frac{\mu^t_{\pi_{\mathrm{ref}}}}{\mu^t}||_\infty\\
\leq \;& \mathrm{E}_{s\sim\mu^t_{\pi}}[V_{\pi_{\mathrm{ref}}{}^t,\pi^{t+1}\ldots}(s) - V_{\pi^t,\ldots}(s)] + \varepsilon||\frac{\mu^t_{\pi_{\mathrm{ref}}}}{\mu^t}||_\infty\\
= \;& \sum_{\tau=t}^{T-1} \varepsilon||\frac{\mu^\tau_{\pi_{\mathrm{ref}}}}{\mu^\tau}||_\infty
\end{aligned}$$

■

We are able to "change measures" with the infinity norm (i.e., the largest value taken at any state) of the ratio of the probability distributions here because the action-value function optimized over a is **always** greater than the value of any other policy. This change of measure follows directly by multiplying inside the second expectation by $\frac{\mu_t}{\mu_t}$, and then bounding the result.

This bound is powerful in that it lets our error go to zero even if we do not get a perfect distribution μ_t as long as we drive our expected error ε to be low. We can also drop the dependence of μ_t on t, by simply averaging all the time slice distributions together

$$\frac{1}{T}\sum_{t=0}^{T-1} \mu_t$$

if we are willing to learn our policies to ε/T error.

This bound provides insight by indicating that it is very important that the "training" distribution be close in a particular way to the distribution induced by any policy (notably the optimal one) we want to compete against. In particular, when training each classifier we want to ensure that we put mass on all places where a good policy spends time so that the ratio $\frac{\mu^\tau_{\pi_{\mathrm{ref}}}}{\mu^\tau}$ is never too large. This makes intuitive sense– we're better to make sure our learner has seen examples of possible situations it can get into even if this means removing some of the mass from more probable instances. Crudely speaking, in choosing a μ we should err on the side of "smearing" our best guess of the distribution $\mu_{\pi_{\mathrm{ref}}}$ induced by the optimal policy across neighboring states.

This style of proof where we use some variation on the performance difference lemma and a change of distribution is very common to the analysis of almost all modern approximate policy iteration methods.

Iterated PSDP

PSDP as presented takes as input the set of space-time distributions μ_t and generates a policy in polynomial time with non-trivial *global* performance guarantees with respect to μ_t. In many applications, we are able to provide a useful state-time distribution to the algorithm. For instance, in control and decision tasks human performance can often provide a baseline distribution to initialize PSDP. We also often have heuristic policies that can be used to initialize the algorithm. Finally, domain knowledge often provides useful indications of where good policies spend time.

In any of these cases, we do not have an accurate estimate of μ for an optimal policy. A natural approach is to apply PSDP as the inner loop for a broader algorithm that attempts to simultaneously compute μs and uses PSDP to compute optimal policies with respect to it. Perhaps the most natural such algorithm is given below.

Algorithm 16: IteratedPSDP(μ, π, v)

Let π_{new} = PSDP(μ)
v_{new} = Value(π_{new})
μ_{new} = ComputeInduced$\mu(\pi_{new})$ **in**
if $v_{new} \leq v$ **then**
  return π

else
  return IteratedPSDP ($\mu_{new}, \pi_{new}, v_{new}$)

Value here is a function that returns the performance of the policy and ComputeInducedμ returns a new baseline distribution corresponding to a policy. These can both be implemented a number of ways, perhaps the most important being by Monte-Carlo sampling.

We start Iterated PSDP with $v = 0$ and a null policy in addition to our "best-guess" μ. Iterated PSDP can be seen as a kind of search where the inner loop is done optimally. For exact PSDP ($\varepsilon = 0$) on an MDP with finite states and actions, we can prove that performance improves with each loop of the algorithm and converges in a finite number of iterations.

In the case of approximation, it is less clear what guarantees we can make. Performance improvement occurs as long as we can learn policies at each step that have smaller average residual advantages than the policy we are attempting to improve over.

Summary

PSDP is a useful algorithm template and can serve as a kind of design pattern for approximate DP algorithms and as a tool of theoretical ananlsis. While there are a number of practical applications of PSDP, even within robotics[30], it is not nearly as commonly used in practice as online variants of the approximate dynamic programming or policy gradient algorithms we'll investigate later.

[30]

8.4 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.