Approximate Dynamic Programming

Introduction

Approximate Dynamic Programming (ADP), also sometimes referred to as neuro-dynamic programming, attempts to overcome some of the limitations of value iteration. Mainly, it is too expensive to compute and store the entire value function, when the state space is large (e.g., Tetris). Furthermore, a strong access to the model is required to reconstruct the optimal policy from the value function. To address these problems, there are three approximations one could make:

1. Approximate the optimal policy
2. Approximate the value function V^*
3. Approximately satisfy the Bellman equation

Approximate Value Iteration

We begin by looking at methods that approximate the value function. However, we first describe the action-value function, a close analogue of the value function that quantifies the expected reward for every state-action pair, rather than every state.

Action-Value Functions

The quality function, Q-Function, or action-value function is defined as

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

These can be restated in terms of Q as

$$Q^*(x,a) = c(x,a) + \gamma \min_{a'} \mathbb{E}_{P(x'|x,a)}[Q^*(x',a')]$$
$$Q^\pi(x,a) = c(x,a) + \gamma \mathbb{E}_{P(x'|x,a)}[Q^\pi(x',\pi(x'))]$$

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

$$V^*(x) = \min_a Q^*(x, a)$$
$$\pi^*(x) = \operatorname*{argmin}_a Q^*(x, a)$$

The optimal policy is computed from the value function as

$$\pi^* = \operatorname*{argmin}_a c(s, a) + \mathbb{E}_{P(x'|x,a)}[V^*(x')]$$

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.
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|).

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 $D = \{x, a, x', c\}_{i=1,2,\ldots,n}$
  $Q^T(x) = 0 \; \forall x$
  forall $t \in [T-1, T-2, \ldots, 0]$ **do**
    $D^+ = \emptyset$
    forall $i \in 1, \ldots, n$ **do**
      input = $\{x_i, a_i\}$
      target = $c_i + \gamma \min_{a'} Q^{t+1}(x_i', a')$
      $D^+ = D^+ \cup \{\text{input}, \text{target}\}$
    end
    $Q^t = \textsc{Learn}(D^+)$
  end
  return $Q^{0:T-1}$

The algorithm takes as input a dataset D which contains samples of the form {state, action, next state, associated cost}. In practice, this

is obtained by augmenting expert demonstration data with random exploration samples. As in value iteration, the algorithm updates the Q function by iterating backwards from the horizon $T-1$. Essentially, at each iteration t, we create a training dataset D^+ by using the Q function learned in iteration $t+1$. 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^T, which may reduce convergence time in the infinite horizon case.

There are a few of things that we need to be aware of when using this in practice:

- We need to make sure that our samples include goal states in order to get meaningful iterations.
- Often we run on features of state x, not the raw state itself.
- We often run fitted Q-iteration repeatedly, augmenting the pattern 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/forbidden states are provided a large 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.
- 2 layer neural net for regression. 1 output for each action.
- Steering was the only controlling actions was discretized into 5 angle choices.

Challenges when using Fitted Q-Iteration

Fitted Q iteration is prone to bootstrapping issues in the sense that sometimes it does not converge. Since this method relies on approx-

imating 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 [4] demonstrate this problem. [4]

Example: 2D gridworld

Figure 15 shows the 2D grid world example, which has a linear true value function J^*.

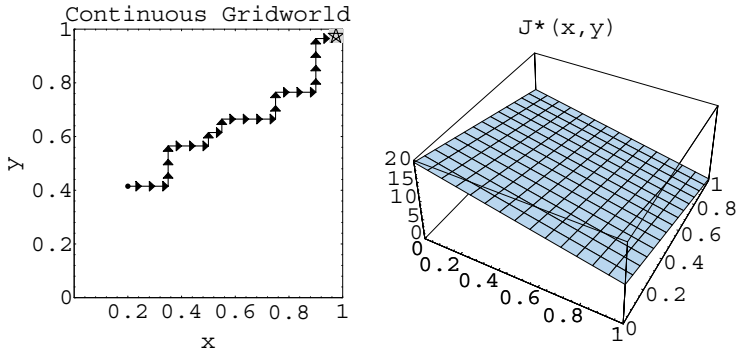

Figure 15: The continuous gridworld domain.

Figure 16 shows that VI converges to the true value function.

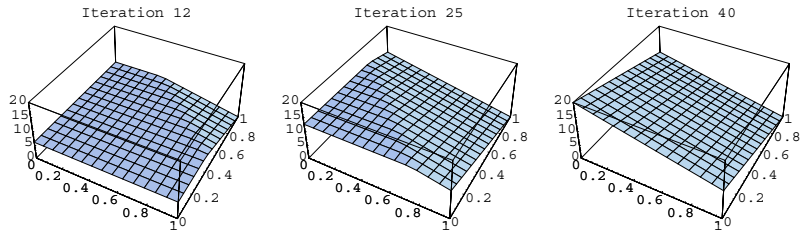

Figure 16: Training with discrete value iteration.

However, figure 17 shows that FVI with quadratic regression underestimates the values at the corner opposite the goal, and these underestimates amplify at each iteration.

Example: car on hill

Figure 18 shows the car-on-hill example.

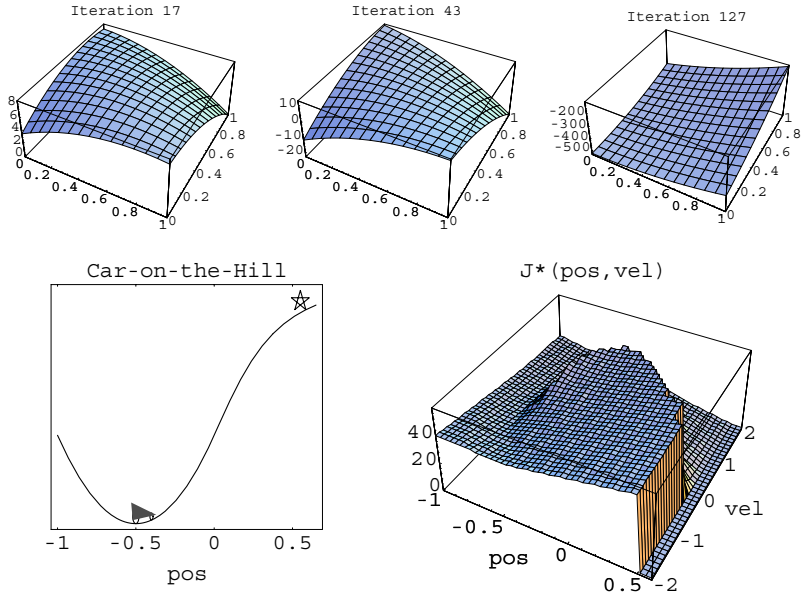

Figure 17: Training with quadratic regression. The Q function diverges. The quadratic function, in trying to both be flat in the middle of state space and bend down toward 0 at the goal corner, must compensate by underestimating the values at the corner opposite the goal. These underestimates then enlarge on each iteration, as the one-step lookaheads indicate that points can lower their expected costs-to-go by stepping farther away from the goal.

Figure 18: The car-on-the-hill domain.

Figure 19 shows that a two layer MLP can also diverge to underestimate the costs.

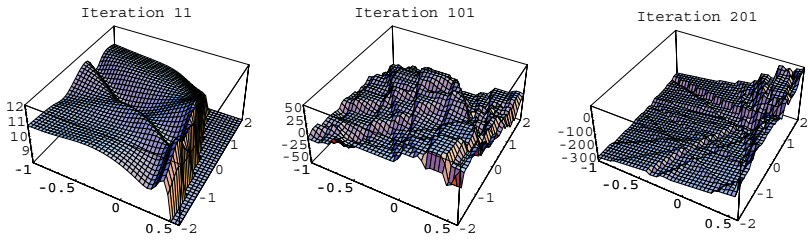

Figure 19: Training with neural network.

In the following lecture, we will see how fitted policy iteration can fix these bootstrapping problems.