10

Black-Box Policy Optimization

Up to this point we have learned primarily about dynamic programming-based approaches to control. These problems set up a Bellman equation which can be solved to discover an optimal or approximately optimal controller. This lecture focuses on ways to avoid relying on Bellman equations and the perils of dynamic programming and distribution shift. Put more poetically by Andrew Moore, this chapter focuses on how not to be "blinded by the beauty of the Bellman equation."

The following approaches will focus on finding a set of parameters which defines a good controller. For example, in Tetris, we could imagine defining a policy $\pi_\theta : x \mapsto a$ which is parameterized by θ. These parameters might be weights on various features defined on state-action pair (x, a), such as the maximum height of the board or the number of holes of the resulting configuration. A policy under this parameterization can be defined at every state x as,

$$\pi_\theta(x) = \underset{a \in \mathbb{A}}{\operatorname{argmin}} \left(\theta_1 \times \#\text{ of Holes}(x, a) + \theta_2 \times \text{Height}(x, a)\right).$$

In general, we have

$$\pi_\theta(x) = \underset{a \in \mathbb{A}}{\operatorname{argmin}}\ \theta^T f(x, a),$$

where $f(x, a)$ is a vector of features of the state-action pair (x, a).

Let ξ denote a *trajectory* of states and actions, $\xi = (x_0, a_0, \ldots, x_{T-1}, a_{T-1})$. We define the *total reward of the trajectory* ξ as,

$$R(\xi) \ = \ \sum_{t=0}^{T-1} r(x_t, a_t).$$

Our goal is to find the parameters that produce the policy that maximizes the expected total reward of the trajectories,

$$J(\theta) = E_{p(\xi|\theta)}[R(\xi)] = E_{p(\xi|\theta)}\left[\sum_{t=0}^{T-1} r(x_t, a_t)\right],$$

where $p(\xi|\theta)$ is the probability of the trajectory ξ given the policy parameterized by θ.

There are tremendous advantages to this simple, stochastic optimization viewpoint on the reinforcement learning problem:

Pros of Policy Optimization with Parameterized Policies:

- **No** dependence on size of state space (directly)
- A policy can be much simpler than a value function. For example, in the famed mountain car problem [1], the optimal policy is simple to specify: move backwards until a certain point, then move forwards. The value function for this problem, however, is significantly more complex.
- Engineering knowledge about the domain can be put directly into the policy by selecting good features.
- Only needs a crude reset access model to optimize the policy directly.
- Simple and easy to code up!

Cons of Policy Optimization with Parametrized Policies:

- Needs careful design of features. With poor features, no amount of searching will find a good policy. Also, the features need to have somewhat smooth gradients for this type of gradient descent to be effective.
- Strong dependence on the number of parameters. Irrelevant or redundant parameters make the problem much harder (potentially exponentially harder).
- Exploration is particularly difficult in this setting and can lead to exponentially slow convergence in the number of states.
- As we'll discuss in the next chapter, we're ignoring key, known, information including the relationship (e.g. Jacobian) between parameters and action choices and the markov structure of states and rewards. This may come at significant sample complexity cost.

10.1 How to find a good parameter set θ?

Gradient Ascent/Descent

Perhaps the most obvious way to solve this problem would be to use a gradient ascent style algorithm. Gradient ascent starts at some intial point, evaluates the gradient of the objective function $J(\theta)$, which is the expected total reward function in our case, and then takes a step up the gradient (if we are maximizing). Gradient ascent continues stepping in the direction of the gradient (the direction that the function J has the greatest rate of increase) until it converges, i.e., the gradient is small enough.

Problem 1:

- $J(\theta)$ may not be differentiable, i.e., changing θ by a infinitesimal small change δ could cause J to jump substantially
- $J(\theta)$ may be very hard to differentiate analytically, particularly because, by assumption we only have *0th* order access. That is, we can draw samples, but we can't get access to derivates of the world dynamics.

[1] https://en.wikipedia.org/wiki/Mountain_car_problem

Idea: approximate the gradient using finite differences

For each parameter we could add a small scalar δ to it and evaluate the value of J at $\theta + \delta_i$, where $\delta_i = (0, \ldots, 0, \delta, 0, \ldots, 0)$.

Then, we can use the finite difference $\frac{1}{\delta}(J(\theta + \delta_i) - J(\theta))$ to estimate the derivative in the ith direction. The estimated gradient then is

$$\widetilde{\nabla} = \frac{1}{\delta} \cdot \begin{bmatrix} J(\theta + \delta_1) - J(\theta) \\ \vdots \\ J(\theta + \delta_n) - J(\theta) \end{bmatrix}.$$

Problem 2:

- We may not have access to the value of $J(\theta)$, rather, we may have a noisy sample $\tilde{J}(\theta)$, which is the case for the expected total reward function.

Idea: estimate the gradient using the samples.

Similarly, we could add a small scalar δ to each parameter and take a single *sample* $\tilde{J}(\theta + \delta_i)$ to estimate the derivative in the ith direction. The estimated gradient is

$$\widetilde{\nabla} = \frac{1}{\delta} \cdot \begin{bmatrix} \tilde{J}(\theta + \delta_1) - \tilde{J}(\theta) \\ \vdots \\ \tilde{J}(\theta + \delta_n) - \tilde{J}(\theta) \end{bmatrix}.$$

However, this estimate can be noisy. If we want a better estimate of the gradient, we could sample multiple times and take an average. A better way would be to use a linear least squares approach for a large number of sample vectors. Specifically, we create tuples, $\{\Delta^{(j)}, \tilde{J}(\theta + \Delta^{(j)}) - \tilde{J}(\theta)\}_{j=1}^{N}$. Then, by the Taylor series expansion, we have,

$$\tilde{J}(\theta + \Delta^{(j)}) - \tilde{J}(\theta) \approx (\nabla_\theta J)^\top \, \Delta^{(j)}$$

Then, the problem of estimating gradient can be interpreted as the following linear least squares regression problem,

$$\tilde{\nabla} \;=\; \underset{\nabla'}{\operatorname{argmin}} \sum_{j=1}^{N} \left| \, (\nabla')^\top \, \Delta^{(j)} - \left(\tilde{J}(\theta + \Delta^{(j)}) - \tilde{J}(\theta) \right) \right|^2.$$

In the next lecture, we will see other methods to estimate $\nabla_\theta J$ called the *policy gradient methods*.

In some domains, such as a deterministic simulator (although the simulator may simulate randomness, it itself is deterministic, such as Tetris), we can use the so called *Pegasus [1] trick*: simply fix the random seed. This can be useful because it fixes a single (noisy) estimate of the true gradient and helps keep the gradient consistent. This can be dangerous because it is sacrificing bias to obtain a lower variance estimate and may drive θ towards areas areas that are not actually a local optima.

With the gradient estimate, we can update the parameter θ:

$$\theta \leftarrow \theta + \alpha \, \widetilde{\nabla}$$

where α is the *step size* or *learning rate*. In practice, for good convergence we need $\alpha \approx \frac{1}{\sqrt{T}}$ where T is the time horizon of the problem.

Note, however, that poor gradient estimates can cause incorrect behavior. In the worst case, the estimated gradient near an almost flat section could be 0 in all directions.

Alternative and Useful oth-Order Optimization Algorithms

In addition to the algorithms covered in more detail below, it may be worth considering other black box techniques:

- Nelder-Mead. At least one of those others always had good luck with this method (sometimes called the simplex method). [2]
- CMA-ES and Cross-Entropy [3] These algorithms balance explortation with "gradient-like" exploitation by maintaining a probability distribution over parametric hypthothesis. We detail one variant below.
- Simulated annealing. This method performs gradient descent like updates (more precisely, hill-climing updates). At each iteration, another set of parameters $\theta + \Delta$ is randomly generated with a small Δ, if $J(\theta + \Delta) > J(\theta)$, we update the parameters $\theta \leftarrow \theta + \Delta$. Otherwise, we still accept the update $\theta \leftarrow \theta + \Delta$ with some probability related to the "temperature" of the system. Initially, the "temperature" is high which means the algorithm tends towards random movement, i.e., even if the value is not better, we still make the updates with high probability. As the search continues the temperature decreases and the algorithm is more likely to move in the ascent direction.
- Genetic Algorithms. These are generally a method of last resort. They evaluate a bunch of random parameters and then the best parameters "survive" and "reproduce" with some "mutation" to create a new set of parameters. This method is nice because it requires basically no knowledge of the problem and, when tuned properly, will explore the space nicely, although it can be very difficult to tune the hyper-parameters of these approaches.
- Q2. This method generates a bunch of samples and fits a quadratic, then solves a quadratic program to optimize the weights. To avoid running outside the region about which the algorithm "quadraticized", it applies linear constraints to bound the solution. It then re-quadraticizes about the new estimate. [4]
- Coordinate Descent. In order to find a minimum, this algorithm performs a line search along one coordinate direction at the current point during each iteration. Different coordinate directions are cycled through as the algorithm iterates.

Nelder-Mead

(See the wikipedia article: http://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method for more info and a nice animated gif)

The Nelder–Mead method was proposed by John Nelder and Roger Mead, two English statisticians working at the National Vegetable Research Station[5]. Perhaps the best summary for the Nelder–Mead method is what Nelder said himself during an interview [3]:

> "There are occasions where it has been spectacularly good. . . Mathematicians hate it because you can't prove convergence; engineers seem to love it because it often works."

Nelder-Mead has many popular variants, one of which is the default algorithm used in MATLAB's `fminsearch` function. It does not require any

[2] ; ; and

[3]

[4]

[5] Nelder later notes that *"Our address (National Vegetable Research Station) also caused surprise in one famous US laboratory, whose staff clearly doubted if turnipbashers could be numerate."* [3]

knowledge of the derivatives or the analytic form of the function being optimized, but it does expect deterministic functions.

Nelder-Mead works on an n-dimensional function by creating a simplex of $n + 1$ points which it modifies to try to surround the optimum. At each iteration, it evaluates the function at each of the vertices of the simplex and follows some complicated rules to move the points until it shrinks the simplex down on a local minima. The original version of the algorithm is not guaranteed to converge.

The following is an overview of the rules used:

- Consider points along the line between the worst point and the (possibly weighted) average of the other points
- Try to reflect the worst point about plane between other points
  - If the reflected point is better than the second worst, but not better than the best, replace the worst with the reflected point.
  - If the reflected point is better than best point, compute a further expanded point past the reflected point. If this point is better than the reflection, replace the worst point with it, otherwise replace the worst point with the reflection.
  - If neither are better, consider contracting the simplex by shortening the distances between the best point and the other points

Note: you should really consult [6] and other references if you were considering implementing this in anger as there are many variants of this algorithm.

Even though it may not have good theoretical properties, in practice this algorithm tends to be very effective. This approach can also be extended to take 4 or 8 samples at each point on the simplex instead of just sampling once. These methods (Nelder-Mead-4 / Nelder-Mead-8) can potentially improve robustness to noise.

Cross-Entropy Method

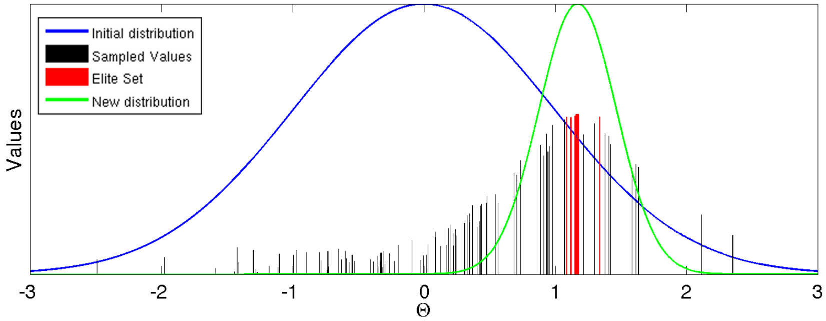

Figure 10.1.1: The first iteration of cross-entropy. The initial distribution is a prior Gaussian (blue) and the green Gaussian is the one fitted to the elite set.

The Cross-Entropy Method [7] [8] samples from a distribution, and then updates the distribution based on which samples scored the highest. This method originated as an approach for importance sampling and has impacts

[6]

[7]

[8] See in particular Algorithm 4.1 of the reference.

in queuing theory as well as being useful as an optimization technique. It's a surprisingly effective brute force method.

The method, shown in Algorithm 19 and illustrated in Figure 10.1.1 starts with a distribution over the parameter space, often a Gaussian, but it can be any distribution. Then samples are taken from the distribution as points at which to evaluate the function. Typically about 100 samples are taken. Then the "elite set" is computed, which is the top $1 - 5\%$ of the samples. The parameters that make up the elite set are then used to create a new distribution. The actual values of the elite set are ignored, only their parameters are used to train a new distribution. Then the new distribution is sampled from and the process repeats until the distribution settles in on a local optimum. The parameters returned could be the mean of the final distribution, or one could track the best value overall and use that as the final parameter set.

Algorithm 19: Cross entropy method

```
given: An initial distribution $\mathcal{D}_\theta$ over the set of parameters
outputs: A final set of parameters $\theta_n$
while not converged do
    for $i = 1$ to $k$ do
        sample $\theta_i$ from $\mathcal{D}_\theta$
        $v_i \leftarrow J(\theta_i)$    {Run the simulator to obtain a value}
    end for
    $E \leftarrow \emptyset$
    for $j = 1$ to $e$ do
        $i \leftarrow \text{argmax}_{i \notin E} v_i$
        $E \leftarrow E \cup \theta_i$    {Find the $e$ best values to create the elite set}
    end for
    $\mathcal{D}_\theta \leftarrow \text{fit}(E)$    {Fit a new distribution to the $x_j$ in the elite set}
end while
```

This method has an interesting set of guarantees, as it has nice exploration property since it samples randomly at each step. [9]

One possible modification is to mix the old and new distributions, such as by linearly interpolating the mean and covariance in the case of Gaussians and in general regularizing the learned distribution. Typically the interpolation is weighted $70 - 90\%$ in favor of the new distribution. This modification is useful to help avoid singular covariance matrices.

Another nice property of Cross-Entropy is that it can deal with irrelevant or noisy features. If two features are related, their covariance in the distribution will be high.

There are, however, issues with these methods

- Inaccuracies in modeling the true distribution. If the actual distribution is multi modal, then that can cause the covariance to keep growing to accommodate the bimodal nature of the underlying distribution
- If the sampling is not done right, then there might be too few elements in the covariance matrix. To fix this, some people try to increase the diagonals along the covariance by adding a 'regularizer' term to the covariance

[9]

matrix, i.e. a λI, or by linearly combining the distributions as mentioned earlier.

- This method actually optimizes quantiles [2] rather than the actual expected values. Thus, if using a black box method, it will converge, but if a stochastic policy method is used, it will not converge because of noise.

Black box methods usually must evaluate J many times, and thus work well when evaluating J is cheap. However, this is almost never the case in robotics. Their simplicity and robustness to incorrect modeling assumptions (partial observability and difficult approximating value functions) make them particularly appealing and often they can be used on a learned model of a system to improve sampl efficiency. We study the use of learned models in a later lecture.

10.2 Related Reading

[1] PEGASUS: A policy search method for large MDPs and POMDPs. Ng, Andrew Y and Jordan, Michael

[2] The Cross-Entropy Method Optimizes for Quantiles. Goschin, Weinstein and Littman

[3] Optimization stories. GrÃűtschel, Martin, ed. Dt. Mathematiker-Vereinigung, 2012.