Weekly report - 2/13 - 2/20. Imitiation Learning and Policy Optimization

Siddharthan Rajasekaran

1 Summary

In this report we will discuss some of the methods to do reinforcement learning (forward control) in high dimensional continuous systems. This is important to us because at every iteration IRL, we have to do solve the forward control to see how good our reward function learned is. This also gives information about out gradient along which we can improve our reward function.

2 Problems with Value iteration

Solving the forward problem exactly using value iteration takes $|\mathscr{S}|^2 \times |\mathscr{A}| \times |\mathscr{I}|$ computation steps where $\mathscr{S}|$ is the number of states, $|\mathscr{A}|$ is the number of actions and $|\mathscr{I}|$ is the number of minimum number of iterations to converge (which can be proved to be a polynomial in $|\mathscr{S}|$). Clearly, this will not be possible in large or continuous state spaces (which is the case in robotic systems). Hence we go for approximate solutions to the RL problem.

3 RL for high dimensional/ continuous systems

The high dimensional problem can be solved using

1. Reward Shaping [**?**]
2. Policy search
    - Cross-Entropy / Evolution Strategy based methods [**?**, **?**, **?**, **?**]
    - Policy Optimization (gradient based) [**?**, **?**]
3. Approximate Q learning [**?**]

We will discuss Policy search based methods more in detail as they seem promising for our continuous space tasks. We should also explore approximate Q learning based methods in future. Reward shaping however, requires the shaping function to be a difference potential function and has limited capability to scale compared to policy search.

3.1 Policy Optimization

The main motivation behind policy optimization is that it is often hard to define value function or Q function for a specific task. For example, in the task of grasping, one can easily accept a the set of actions (the policy) to be good. However, assigning a value to particular pose during the grasping problem is harder to track [**?**].

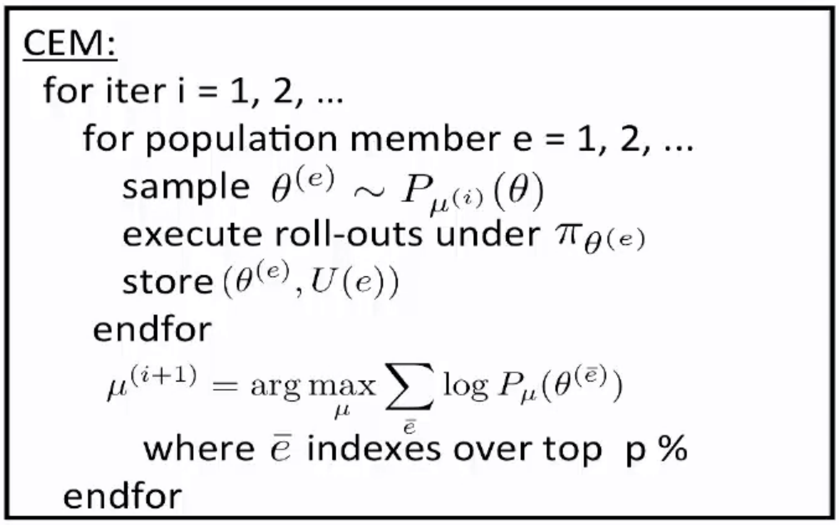
In policy optimization, the way we parameterize the policy π using θ. For example, the parameters θ may be the weights in a huge neural networks that maps states to actions or directly observations to actions. The reinforcement learning objective in the context of policy optimization becomes,

$$\theta = \arg\max_{\theta} E[\sum_{t=0}^{T} R(s_t)|\pi_\theta] \tag{1}$$

Often stochastic policy class is chosen to smooth out the policy optimization problem in case the underlying reward function is discontinuous. This works better even though we know that we only need a deterministic policy.

3.1.1 Cross Entropy Method

This is a evolution (genetic algorithm) based method. The following shows the algorithm.



```
CEM:
  for iter i = 1, 2, ...
    for population member e = 1, 2, ...
      sample $\theta^{(e)} \sim P_{\mu^{(i)}}(\theta)$
      execute roll-outs under $\pi_{\theta^{(e)}}$
      store $(\theta^{(e)}, U(e))$
    endfor
    $\mu^{(i+1)} = \arg\max_{\mu} \sum_{\bar{e}} \log P_{\mu}(\theta^{(\bar{e})})$
      where $\bar{e}$ indexes over top p %
  endfor
```

Figure 1: Algorithm - Cross Entropy Method

In the above algorithm we have a parameterized distribution P_μ for the parameter θ of the policy. We first sample n parameters from P_μ that is $\theta^{(e)} \sim P_{\mu^{(i)}}(\theta)$. We execute roll-outs for each of the member in the population and find their cumulative reward $U(e)$. We pick the top $p\%$ individuals with respect to their U of the population and perform a soft-max to update the parameter of our distribution.

There are several different ways one could update the parameter μ. In [**?**] they perform a reward weighted regression in which the parameter update is given by

$$\mu^{(i+1)} = \arg\max_{\mu} \sum_{e} q(U(e), P_\mu(\theta^{(e)})) \log P_\mu(\theta^{(e)}) \tag{2}$$

Here each individual is weighted by a function q of the utility if that individual $U(e)$ and the probability of sampling the parameter $\theta^{(e)}$ from distribution P_μ. In [**?**], instead of picking top $p\%$, you look at all roll-outs and you weigh each roll out by exponentiated utility value $U(e)$ of that roll-out. The update is given by,

$$\mu^{(i+1)} = \arg\max_{\mu} \sum_{e} \exp(\lambda U(e)) \log P_\mu(\theta^{(e)}) \tag{3}$$

In [**?**], we keep a distribution in a form of a Gaussian parameterized by mean μ and covariance Σ. The update here is given by,

$$(\mu^{(i+1)}, \Sigma^{(i+1)}) = \arg\max_{\mu,\Sigma} w(U(\bar{e})) \log \mathcal{N}(\theta^{(\bar{e}}; \mu, \Sigma) \tag{4}$$

In [**?**], we optimize the expected log probabilities of the roll-outs that we got. The distributions are again assumed to be Gaussian The update here is,

$$\mu^{(i+1)} = \mu^{(i)} + \Big(\sum_{e} (\theta^{(e)} - \mu^{(i)}) U(e)\Big) / \Big(\sum_{e} U(e)\Big) \tag{5}$$

This method, called PoWER, has been tested on a rather dynamic task where the robot arm has to swing a ball and drop in a cup it is holding. The link here shows the demonstration of PoWER in action - `https://www.youtube.com/watch?v=qtqubguikMk`

The main problem with these methods is that, they work well in case we need to find relatively few parameters. In case we want to learn the parameters of a neural net with hundred thousand parameters, it is difficult to make these methods work.

Another way to approach this would be to use policy gradient methods. This is where the stochastic policy smooths out a discontinuous reward function. The gaol here can be broadly described as

$$\max_{\theta} U(\theta) = \max_{\theta} \sum_{\tau} P(\tau;\theta)R(\tau) \tag{6}$$

The gradient is given by

$$\bigtriangledown_{\theta} U(\theta) = \sum_{\tau} P(\tau;\theta) \bigtriangledown_{\theta} \log P(\tau;\theta)R(\tau) \tag{7}$$

Empirically, this is estimated in [?] as,

$$\bigtriangledown_{\theta} U(\theta) = \frac{1}{m} \sum_{i=1}^{m} \bigtriangledown_{\theta} \log P(\tau;\theta)R(\tau) \tag{8}$$

Intuitively, this just moves the parameters in the direction in which the probability of paths with positive R is increased and that of negative R is decreased.

In a Markovian process, the gradient of the probability P of a path τ becomes independent of the system dynamics. That is,

$$\begin{aligned} \bigtriangledown_{\theta} \log P(\tau;\theta) &= \bigtriangledown_{\theta} \log \left[\Pi_{t=0}^{H} P(\cdot)\pi_{\theta}(\cdot)\right] \\ &= \sum_{t=0}^{H} \bigtriangledown_{\theta} \log \pi_{\theta}(u_t^{(i)}|s_t^{(i)}) \end{aligned}$$

The bias of the method is proved to be zero and several variance reduction techniques are discussed in [?] for the empirical estimate of the gradient.

In [?], they come up with a method that optimizes an approximation to objective function (cumulative reward), a surrogate objective, which makes sure that we monotonically increase the reward while using gradient method. Also, [?] guarantees monotonic increase in reward for highly non linear parametric policy such as the ones in neural nets. This allows us to use deep networks for the policy and still apply policy gradient methods. This allows us to effectively combine deep learning and robotics unlike imitation learning.

4 Conclusion

The report summarizes some deep learning methods for imitation learning and policy optimization methods. Need to read more on inverse reinforcement learning and form a ”Big picture” of how to approach Learning from Demonstration problems.

5 Bibliography