**CS 182/282A: Designing, Visualizing and Understanding Deep Neural Networks** **Spring 2019**

# Section 9: Markov Decision Processes and Policy Gradient Methods

*Notes by: David Chan*

## 9.1 Course Logistics

- You should have received your Midterm 2 grades, regrades are open until Monday, 4/22 at 11PM.
- Please check in with your project group GSI. If you don't do this you won't receive credit under the participation grade for this part of the assignment.
- The poster session is 2PM - 4PM in 310 Jacobs Hall on Tuesday May 7th. The project report will be due on Monday, 5/13 at 11PM. We will be posting more detailed requirements soon.

## 9.2 Markov Decision Processes

Consider playing a game of chess. On each of your turns, you must survey the board (make an observation of the current state of the game) and choose an action which hopefully will lead you on to victory (or the promise of a high reward). We can formalize this process mathematically using what is called a **Markov Decision Process (MDP)**. In an MDP at every time step, we are in some state $s$. A decision maker, usually called the **agent**, must choose an action $a$ from the available set of actions in the state. The MDP responds to this action by transitioning to a new state $s'$ via some probability distribution dependent on the state, and the chosen action (notice the transition doesn't depend on time or any previous state. MDPs are markovian, so it only matters what state you are in, not how or when you got there). On the transition, MDPs also generate a real-valued reward $r$ (which can be positive or negative) corresponding to positive or negative actions in the environment. A sequence of such transitions $\tau = ((s_0, a_0), (s_1, a_1), (s_2, a_2)...(s_t, a_t, r_t))$ is called a trajectory.

Formally, an MDP is a discrete time stochastic control process consisting of a 4-tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{T}, r)$ where:

1. $\mathcal{S}$ is a finite set of states.
2. $\mathcal{A}$ is a finite set of actions. Sometimes this is written as $A(s)$, a function mapping a state to the set of actions available in that state.
3. $T(s, a) : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the "transition operator" which maps a current state, and an action taken in that state to a new state. A deterministic MDP has a deterministic transition operator, while a stochastic MDP may transition to a new state according to some distribution $P(s_{new}|s_{old}, a)$.
4. $r(s, a, s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function, that is, the amount of reward you get for taking an action $a$ in state $s$ and ending up in state $s'$. Sometimes this is specified as the reward for entering the state $s'$ (so that reward is only a function of $s$ and $a$, as in $r(s, a)$). This is not a major difference.

The **Solution** to an MDP is a **policy** $\pi^*$ which specifies for any state $s$, the action to take $a$ so that the expected sum of rewards is maximized. That is, given $\Pi$ is a set of potential policies:

$$\pi^* = \arg\max_{\pi \in \Pi} \mathbb{E}_{\tau \sim p_\pi(\tau)} \left[ \sum_{(s_t, a_t) \in \tau} r(s_t, a_t, s_{t+1}) \right] \tag{9.1}$$

There are a couple of confusing terms in the above. The expectation is taken with respect to $\tau \sim p_\pi(\tau)$. What this means, is that we are taking an expected value across the possible trajectories generated by the policy (each weighted by the likelihood that the policy explores that trajectory). This still might be a bit hard to digest, so let's start by considering a fully deterministic environment and policy with a fixed starting state. Because the policy specifies exactly the action to take in each state (the policy is deterministic), the environment always transitions to the same next state (the environment is deterministic), and we always start in the same state (fixed starting state) no matter what happens each policy generates *exactly* one trajectory, with a sum of rewards corresponding to that policy. Solving the MDP in this case degenerates to finding the policy/trajectory which achieves the highest reward on that one trajectory.

We will now relax the assumptions one by one. Consider now a non-fixed starting state (but still deterministic policy and environment). Then we have $|S|$ possible trajectories/sums of rewards that can be generated, and the expectation over $\tau \sim p_\pi(\tau)$ is a weighted sum over those trajectory rewards where the weights are the probability of starting in any given state. We can see then how to generalize this expectation to stochastic environments/policies. There are now many more possible trajectories/sums of rewards that can be generated - and each of those should be weighted by their likelihood.

Often, we add an additional term $\gamma \in (0, 1]$ to this formula:

$$\pi^* = \arg\max_{\pi \in \Pi} \mathbb{E}_{\tau \sim p_\pi(\tau)} \left[ \sum_{(s_t, a_t) \in \tau} \gamma^t r(s_t, a_t, s_{t+1}) \right] \tag{9.2}$$

$\gamma$ is called the **Discount Factor**, and represents a trade-off between rewards in the present, and rewards in the future. Notice how as $t$ grows larger, $\gamma^t$ becomes smaller - thus, we realize that a small reward earlier can become bigger than a large reward later.

## 9.3 Reinforcement Learning

The goal of reinforcement learning is to find a solution to an MDP. In the RL formulation, we have a policy $\pi_\theta : \mathcal{S} \rightarrow \mathcal{A}$ where $\theta$ is a set of parameters. The goal is to find values of $\theta$ which maximize the expected reward over the trajectories generated by that policy. Figure 9.1 gives an overview of the RL goals.

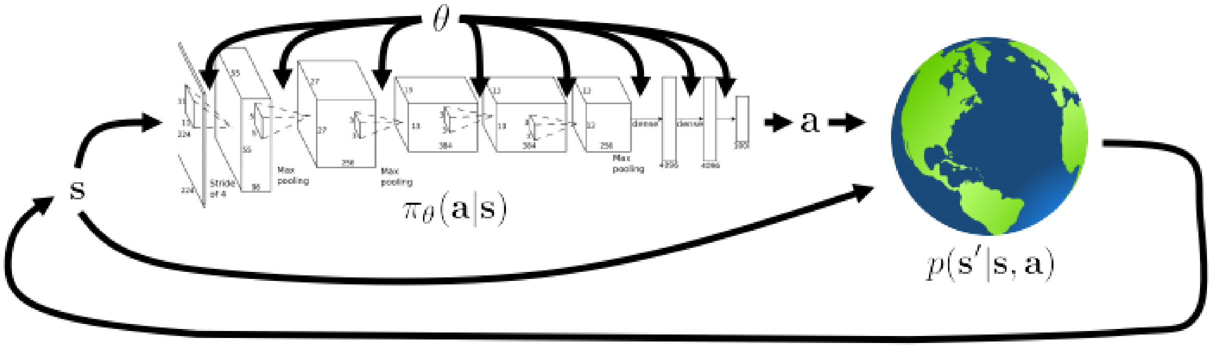
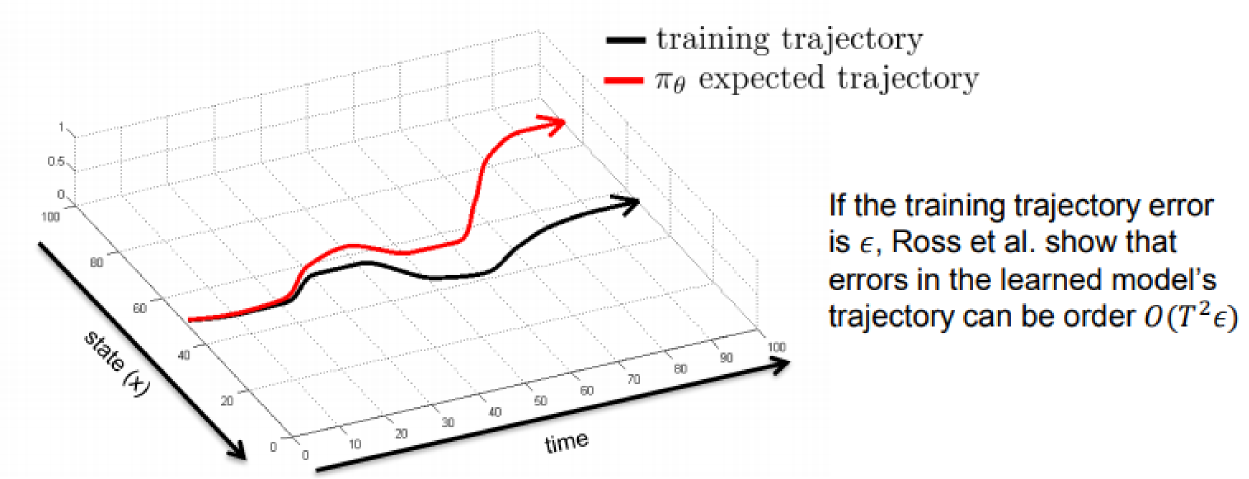

θ
s
a
Stride of 4
Max pooling
Max pooling
Max pooling
dense
dense
4096
4096
1000
$\pi_\theta(\mathbf{a}|\mathbf{s})$
$p(\mathbf{s}'|\mathbf{s},\mathbf{a})$

Figure 9.1: The figure above outlines the goals of reinforcement learning. In RL, we would like to find a policy which maximizes the sum of rewards. We do this by adjusting the parameters of the policy $\theta$. The policy can be a deep neural network, or something more fixed. We can then use the policy to choose actions, which drive the state transitions in the MDP for our agents.

## 9.4 Imitation Learning

In imitation learning, we treat finding a policy $\pi^*$ as a supervised learning problem. We can take sample trajectories from an expert, which gives us a set of state action pairs (presumably, if you're in the same state as an expert, you should take the same action). We can then train a network to directly predict the action from the state. This turns the solution to the MDP into a supervised learning problem which we are, by now, experts at dealing with.



If the training trajectory error is $\epsilon$, Ross et al. show that errors in the learned model's trajectory can be order $O(T^2\epsilon)$

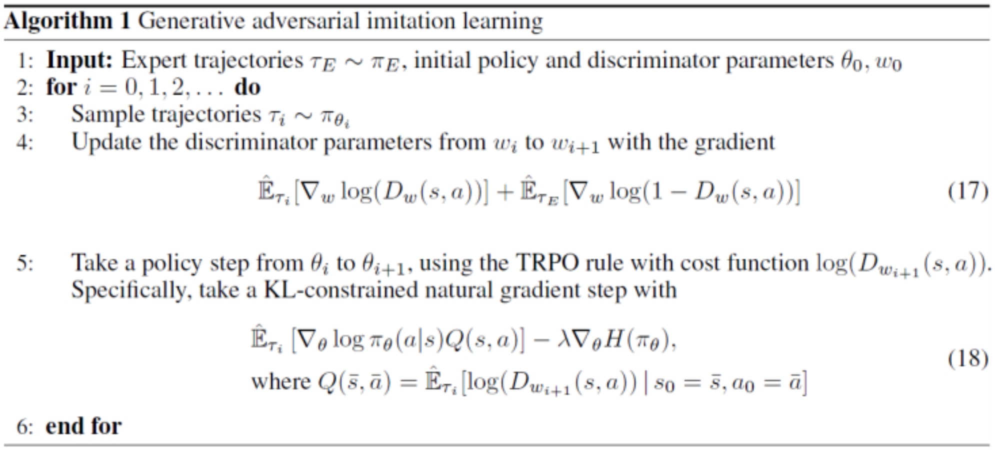
Figure 9.2: Figure showing the issue with imitation learning. Taken from the course slides.

While we have been able to achieve some remarkable results with imitation learning, Figure 9.2 shows why this supervised process doesn't work to perfection. Because experts explore only a small part of the state space in an environment the network isn't exposed to many of the potential things that could happen.

Thus, when the test time deviates from an expert trajectory, the imitation learning policy doesn't know how to recover from the errors leading to poor decisions, leading to even larger errors. This feedback loop can lead imitation learners to achieve poor performance even in simple environments.

One way of correcting for this, is by attempting to expand the set of states for which expert actions are available. A process called DAgger attempts to generate **on policy** training data from the experts. This training data more closely matches the distributions of states that an imitation learning policy might see; however, it can be time consuming to collect because it requires a human or some expert to label the data (i.e., provide the action that the agent should have taken at given states or observations). For more information on DAgger, see [1].

Another approach which has been used for supervised learning is GAIL, or Generative Adversarial Imitation Learning. In this algorithm, rather than try to blindly mimic the user, we attempt to solve the same control problem that the user is solving - that is, we attempt to directly estimate a cost function corresponding to the imitation data and then at test time we can optimize this cost function using traditional methods. GAIL works by sampling a set of trajectories from the policy, and using a discriminator to attempt to distinguish between real and generated state/action pairs. We then use this discriminator loss as a reward function for a policy gradient method in order to update the policy. Figure 9.3 gives the full algorithm.



**Algorithm 1** Generative adversarial imitation learning

```
**Input:** Expert trajectories $\tau_E \sim \pi_E$, initial policy and discriminator parameters $\theta_0, w_0$
**for** $i = 0, 1, 2, \dots$ **do**
    Sample trajectories $\tau_i \sim \pi_{\theta_i}$
    Update the discriminator parameters from $w_i$ to $w_{i+1}$ with the gradient
        $\hat{\mathbb{E}}_{\tau_i}[\nabla_w \log(D_w(s,a))] + \hat{\mathbb{E}}_{\tau_E}[\nabla_w \log(1 - D_w(s,a))]$    (17)
    Take a policy step from $\theta_i$ to $\theta_{i+1}$, using the TRPO rule with cost function $\log(D_{w_{i+1}}(s,a))$.
    Specifically, take a KL-constrained natural gradient step with
        $\hat{\mathbb{E}}_{\tau_i}[\nabla_\theta \log \pi_\theta(a|s) Q(s,a)] - \lambda \nabla_\theta H(\pi_\theta),$
        where $Q(\bar{s}, \bar{a}) = \hat{\mathbb{E}}_{\tau_i}[\log(D_{w_{i+1}}(s,a)) \mid s_0 = \bar{s}, a_0 = \bar{a}]$    (18)
**end for**
```

Figure 9.3: The full GAIL algorithm

## 9.5 Policy Gradient Methods

So far, we have only looked at solving MDPs when we have access to expert data. But what if we don't have access to that data? What if we want to develop a policy from scratch merely by exploring the environment? One of the ways to do this is to directly attempt to optimize $\theta$ in $\pi_\theta$. That is, given a state, directly predict the actions that we should take. The question is then, "Why can't we just differentiate the reward with respect to $\theta$ to optimize the policy?" There are two main reasons:

1. The reward is a *function* of the action selected by the policy and the action set is discrete for many problems. We can't directly differentiate through this discrete set.

2. The reward function is a black box - thus it's impossible to differentiate through it.

Another major issue is that rewards can depend on the current state which can depend on previous actions and states, thus, it's hard to know exactly which action was responsible for achieving high reward. This is the *temporal credit assignment problem*.

### 9.5.1 REINFORCE

The good news that that while we can't directly differentiate the reward with respect to the gradients, we can estimate the gradient by sampling large numbers of trajectories and computing the gradients along those trajectories. This corresponds to what is, essentially, a trial and error process. We sample many trajectories, and if an action in a state leads to high reward on average (over the large number of trajectories), we can increase the probability of that action (and if it leads to a low reward on average, we can decrease the probability of that action).

Given we know:

$$\theta^* = \arg\max_\theta \ \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_{(s_t,a_t)\in\tau} r(s_t,a_t)\right] \tag{9.3}$$

We can design a cost function:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_{(s_t,a_t)\in\tau} r(s_t,a_t)\right] \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_i} r(s_{i,t},a_{i,t}) \tag{9.4}$$

Where $N$ is the number of sampled trajectories, $T_i$ is the length of trajectory $i$ and $s_{i,t}/a_{i,t}$ are the states/actions at time $t$ in trajectory $i$.

First, we let:

$$r(\tau) = \left[\sum_{(s_t,a_t)\in\tau} r(s_t,a_t)\right] \qquad \pi_\theta(\tau) = P(\tau|\pi_\theta) \tag{9.5}$$

We can then rewrite Equation 9.4 as:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[r(\tau)] = \int \pi_\theta(\tau)r(\tau)d\tau \tag{9.6}$$

Noticing an interesting identity gives us:

$$\nabla_\theta\pi_\theta(\tau) = \pi_\theta(\tau)\frac{\nabla_\theta\pi_\theta(\tau)}{\pi_\theta(\tau)} = \pi_\theta(\tau)\nabla_\theta\log\pi_\theta(\tau) \tag{9.7}$$

We then take the gradient with respect to $\theta$, and using Equation 9.7:

$$\nabla_\theta J(\theta) = \int \nabla_\theta\pi_\theta(\tau)r(\tau)d\tau = \int \pi_\theta(\tau)\nabla_\theta\log\pi_\theta(\tau)r(\tau)d\tau = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta\log\pi_\theta(\tau)r(\tau)] \tag{9.8}$$

Effectively, what this derivation has shown is that the derivative of the cost function in Equation 9.4 is equivalent to the expected value over the gradients with respect to the log probability of each of the trajectories (times the reward). Let's now consider $log\pi_\theta(\tau)$.

We notice first that:

$$\pi_\theta(\tau) = \pi_\theta(s_1,a_1,s_2,a_2,...,s_T,a_T) = p(s_1)\prod_{t=1}^{T}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t,a_t) \tag{9.9}$$

Notice that this is just the likelihood of the trajectory $\tau$: We start in state $s_1$, and make the choice of $a_1$ which is the probability $\pi_\theta(s_1, a_1) = p(a_1|s_1, \pi_\theta)$. Then, we have to account for the transition dynamics of the MDP (if it's stochastic) so we multiply the probability of actually transitioning into the next state given the current state and action (chosen by the policy). We can take the log of the above to give us:

$$\log \pi_\theta(\tau) = \log p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) = \log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t) \tag{9.10}$$

Substituting this back into Equation 9.8, we notice that:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)} \left[ \nabla_\theta \left( \log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t) \right) r(\tau) \right] \tag{9.11}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta(\tau)} \left[ \nabla_\theta \left( \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \right) r(\tau) \right] \tag{9.12}$$

In the second step, we have eliminated all of the variables which don't depend on $\theta$ since the gradient will be zero. Substituting in the definition of $r(\tau)$ and moving the gradient inside the sum we finally get:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)} \left[ \nabla_\theta \left( \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \right) r(\tau) \right] \tag{9.13}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta(\tau)} \left[ \left( \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left( \sum_{t=1}^{T} r(s_t, a_t) \right) \right] \tag{9.14}$$

Sampling this expectation we get:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)} \left[ \left( \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left( \sum_{t=1}^{T} r(s_t, a_t) \right) \right] \tag{9.15}$$

$$\approx \frac{1}{N} \sum_{i=1}^{N} \left( \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left( \sum_{t=1}^{T} r(s_{i,t}, a_{i,t}) \right) \tag{9.16}$$

From this equation, we derive the first algorithm, the **REINFORCE** algorithm, execute the following in a loop:

- Sample $\{\tau^i\}$ from $\pi_\theta(a_t|s_t)$ (Run the policy $N$ times on the environment)
- Compute the gradients using Equation 9.15. Notice that this is well defined and doesn't require differentiating a black box.
- Update the parameters: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

### 9.5.2 PPO/TRPO

One of the fundamental problems with REINFORCE is that:

- Each gradient step is expensive (requires $O(NT)$ samples from a usually slow environment), so we want to minimize the number of steps.

- But the gradients are extremely noisy, so we can't take larger steps (otherwise the learning process will be relatively unstable).

Methods such as PPO (Proximal Policy Optimization) and TRPO (Trust Region Policy Optimization) were designed to take safe (stable) steps, while maximizing the amount of reward gain.

#### 9.5.2.1 TRPO

The way that they do this is by attempting to maximize the reward while adding a penalty for large changes in $\pi_\theta$, the action distribution. To do this, we maximize the objective:

$$L(\theta') - c\, KL(\pi_\theta || \pi'_\theta) \tag{9.17}$$

$$L(\theta') = \mathbb{E}_{\tau\sim\pi_\theta}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)}A(s,a)\right] \tag{9.18}$$

Where $KL(\pi_\theta||\pi'_\theta)$ is the KL divergence between the action distributions on the samples, A is the advantage (a quantity similar to reward), and $c \in \mathbb{R}$ is a hyper-parameter.

The intuition behind the choice of this optimization is a bit tricky to understand. Notice that $L(\theta')$ is a measure of how much better/worse $\theta'$ performs on the sampled actions than $\theta$. Thus, we are trying to choose $\theta$ to maximize the improvement on the sampled elements. The $KL$ divergence term attempts to minimize the amount the action distributions change between the policies. Thus, we want our new policy $\pi_{\theta'}$ to perform as well as possible, while doing as few things as possible differently (based on the hyper-parameter $c$).

The parameter update is $\theta' = \theta + \frac{1}{c}\nabla^2_\theta KL(\pi_\theta||\pi'_\theta)\nabla_\theta(L(\theta))$. The implementation of such is left as an exercise to the reader.

#### 9.5.2.2 PPO

While TRPO can be complicated to compute, PPO (Proximal Policy Optimization) combines the regularization loss from TRPO (the KL divergence) and the main optimization into a single function. PPO attempts to maximize:

$$L(\theta') = \mathbb{E}_{\tau\sim\pi_\theta}\left[\min\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)}A(s,a), \text{clip}\left(\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)}, 1+\epsilon, 1-\epsilon\right)A(s,a)\right]\right] \tag{9.19}$$

This is a much simpler function to optimize (since it doesn't require computing the Fischer information matrix (the Hessian of the KL divergence) during the update step however notice how it still maintains that only a very small difference between the action distributions should occur on each step. PPO has been used very successfully, notably with OpenAI-Five (Dota) which has achieved super-human performance on complex environments.

## References

[1] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. "A reduction of imitation learning and structured prediction to no-regret online learning". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 627–635.