CS294 DeepRL Notes

Kane Tian

1 9/19

- proof that $\theta' \leftarrow argmax_{\theta'} \sum_t E_{s_t \sim p_\theta(s_t)}[E_{a_t \sim \pi_\theta(a_t|s_t)}[\frac{\pi_\theta(a_t|s_t)}{\pi_\theta(a_t,s_t)} \lambda^t A^{\pi_\theta}(s_t, a_t)]]$ s.t. $D_{KL}(\pi_{\theta'}(a_t \mid s_t) \mid\mid \pi_\theta(a_t \mid s_t)) \leq \epsilon$ improves $J(\theta') - J(\theta)$
- use dual grad descent to enforce the KL constraint
- this equates to $\theta' \leftarrow argmax_{\theta'} \nabla_\theta J(\theta)^T (\theta' - \theta)$, and after approximations: $\theta' = \theta + \sqrt{\frac{2\epsilon}{\nabla_\theta J(\theta)^T F \nabla_\theta J(\theta)}} F^{-1} \nabla_\theta J(\theta)$, where F is the Fisher Info matrix
  - called natural policy gradient, helps to achieve objective and stabilize policy gradient method

2 9/14

- Problems and fixes with Q-Learning
  - correlated samples in online Q-learning since sequential states strongly correlated
    - parallelism solution
    - replay buffers solution
  - value changing that you want to converge to, so fix by updating new ϕ' less frequently in outer loop
  - actual $\neq$ predicted Q-values, too optimistic due to noise and expected value property
    - use double Q-learning (use current and target network to update together, noise will lessen)
- Q-Learning with N-step returns
  - sum up N-steps to place high weight on rewards instead of Q-function only if on policy: faster, less biased if Q-function not great
  - $y_{i,t} = (\sum_{t'=t}^{t'+N-1} r_{j,t'}) + \lambda^N max_{a_j, t+N} Q_{\phi'}(s_{j,t+N}, a_{j,t+N})$
- Q-Learning with continuous actions

- can sample actions
- can find easily maxmizable Q-function
- can train a maximizer (DDPG), another neural net essentially

3 9/12

- Q-Learning
  - forget policies, $argmax_{a_t} A^{\pi} s_t, a_t$ with policy $\pi'(a_t|s_t) = 1 if a_t is argmax$
  - use dynamic programming for cases with few states (the OPT values are Q or V)
  - use neural net for $s \rightarrow V(s)$ in cases where many states possible, but we cannot get V without knowing transition probabilities, so we learn Q instead
  - algo
    * $y_i \leftarrow r(s_i, a_i) + \lambda E[V_{\phi}(s_i')]$, $E[V_{\phi}(s_i')] = max_{a'} Q_{\phi}(s_i', a_i')$
    * $\phi \leftarrow argmax_{\phi} 0.5 \sum_i || Q_{\phi}(s_i, a_i) - y_i ||^2$
    * algorithm is off-policy
    * can allow exploration through epislon-greedy or Boltzmann exploration
    * unfortunately, fitted Value or Q iteration does not converge, and neither does batch actor-critic algorithm based on Bellman backup operator proof

4 9/7

- Actor-Critic Algorithm
  - if we replace $\hat{Q}_{i,t}$ with the actual $Q(s_t, a_t) = \sum_{t'=t}^{T} E_{\pi_{\theta}}[r(s_{t'}, a_{t'})|s_t, a_t]$, better variance and it's the actual value, not an approximation
  - if we replace with $Q(s_{i,t}, a_{i,t}) - V(s_i, t)$ then even better since it's a baseline, so define $A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$
  - $Q^{\pi}(s_t, a_t) = r(s_t, a_t) + E_{s_{t+1} \sim p(s_{t+1}|s_t, a_t)}[V^{\pi}(s_{t+1})] \approx r(s_t, a_t) + V^{\pi}(s_{t+1})$, so $A^{\pi}(s_t, a_t) \approx r(s_t, a_t) + V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$, so let's aim to fit $V^{\pi}(s)$ with a neural net
  - Policy Evaluation
    * $J(\theta) = E_{s_1 \sim p(s_1)}[V^{\pi}(s_1)]$
    * we can approximate $V^{\pi}(s_t) \approx 1/N \sum_{i=1}^{N} \sum_{t'=t}^{T} r(s_{t'}, a_{t'})$ or $\sum_{t'=t}^{T} r(s_{t'}, a_{t'})$

        * training data is therefore $(s_{i,t}, \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'}))$, but we can do better by using the true ideal expected value target: $y_{i,t} = \sum_{t'=t}^{T} E_{\pi_\theta}[r(s_{t'}, a_{t'})|s_{i,t}] \approx r(s_{i,t}, a_{i,t}) + V^{\pi}(s_{i,t+1}) \approx r(s_{i,t}, a_{i,t}) + V_{\phi}^{\pi}(s_{i,t+1})$, which is the previous fitted value function (incorrect but could tune to get close, this method reduces variances but increases bias)
    - the algorithm
        * sample s_i, a_i from $\pi_\theta(a|s)$
        * fit $V_{\phi}^{\pi}(s)$ to sampled reward sums
        * evaluate $A^{\pi}(s_i, a_i) = r(s_i, a_i) + V_{\phi}^{\pi}(s_i') - V_{\phi}^{\pi}(s_i)$
        * calculate $\nabla_\theta J(\theta)$
        * update θ
    - For infinite time steps, we add in a decaying factor because rewards now > rewards later, since you could "die"
        * $y_{i,t} \approx r(s_{i,t}, a_{i,t}) + \lambda \hat{V}_{\phi}^{\pi}(s_{i,t+1})$
- Actor-critic vs policy gradients
    - higher bias, lower variance (policy gradients can improve baseline to $\hat{V}_{\theta}^{\pi}(s_{i,t})$)
    - can actually combine Monte Carlo (policy gradient estimate using trajectories) and actor-critic: $A_n^{\pi}(s_t, a_t) = \sum_{t'=t}^{t+n} \lambda^{t'-t} r(s_{t'}, a_{t'}) - \hat{V}_{\phi}^{\pi}(s_t) + \lambda^n \hat{V}_{\phi}^{\pi}(s_{t+n})$, add in the last term because represents the steps outside of n
    - we then can weight these terms for all n from $1 \to \infty$

5 9/5

- Policy Gradient (model-free)
    - Evaluating objective of RL
        * $\theta^* = argmax_\theta E_{\tau \sim p_\theta(\tau)}[\sum_t r(s_t, a_t)]$
        * Let $J(\theta) = E_{\tau \sim p_\theta(\tau)}[\sum_t r(s_t, a_t)] \approx 1/N \sum_i \sum_t r(s_{i,t}, a_{i,t})$ (essentially to approximate the expected reward, take trajectories [call the policy repeatedly] and find their rewards, take their mean)
        * $J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[r(\tau)] = \int \pi_\theta(\tau) r(\tau)\, d\tau$, and our goal is to find $\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) r(\tau)\, d\tau = \int \pi_\theta(\tau) \nabla \log(\pi_\theta(\tau)) r(\tau)\, d\tau = E_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log(\pi_\theta(\tau)) r(\tau)]$
        * $\log(\pi_\theta(\tau)) = \log(p(s_1)) + \sum_{t=1}^{T} \log(\pi_\theta(a_t|s_t)) + \log(p(s_{t+1}|s_t, a_t))$, so $\nabla_\theta \log(\pi_\theta(\tau)) = \nabla_\theta \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t)$, so $\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t))(\sum_{t=1}^{T} r(s_t, a_t))]$

    - can approximate with samples: $\nabla_\theta J(\theta) \approx 1/N \sum_{i=1}^{T}[(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t))(\sum_{t=1}^{T} r(s_t, a_t))]$, update theta with this in gradient ascent
    - this is REINFORCE algorithm
  - What is policy in the formula?
    - can model as neural network (gaussian), with variance
  - Partial observability
    - just replace state with observation in formula above, Markov property is not actually used
  - What's wrong with policy gradient
    - hard to converge because high variance
  - Reducing Variance
    - Due to causality, the sum of rewards can be summed from t to T instead of 1 to T
    - Use baselines to replace $r(\tau)$: $r(\tau) - 1/N \sum_{i=1}^{N} r(\tau)$, does not affect gradient (expectation does not change) but reduces variance
      - Var $= E_{\tau\sim\pi_\theta(\tau)}[(\nabla_\theta \log \pi_\theta(\tau)(r(\tau)-b))^2] - E_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau)(r(\tau)-b)]^2$, first term is affected by b while the second isn't
      - take derivative of first term, set to 0, get $b = E[(\nabla_\theta \log \pi_\theta(\tau))^2 r(\tau)]/E[(\nabla_\theta \log \pi_\theta(\tau))^2]$, just weighted mean
  - Policy gradient is on-policy, which can be extremely inefficient
  - Off-policy learning through importance sampling
    - Samples from $\overline{\pi}(\tau)$, another distribution
    - $E_{x\sim p(x)}[f(x)] = \int p(x) f(x)\, dx = \int q(x) \frac{p(x)}{q(x)} f(x)\, dx = E_{x\sim q(x)}[p(x)/q(x)\, dx]$, so $\nabla J(\theta') = E_{\tau\sim\pi_\theta(\tau)}[\frac{\nabla \pi_{\theta'}(\tau)}{\pi_\theta(\tau)} r(\tau)] = E_{\tau\sim\pi_\theta(\tau)}[\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} \nabla \log_{\pi_{\theta'}}(\tau) r(\tau)]$
    - can add causality after replacing the τ with the full summation notation
    - can also extend to state-action marginal
  - Policy gradient in practice
    - bigger batches, other tricks to reduce variance because it's a big problem

6 8/31

- reward function $r(s, a)$ tells which actions/states are better
- Markov chain
  - $M = \{S, T\}$, where S = state space, T = transition operator (conditional probabilities to transition between states)

- $\mu_{t+1} = T\mu_t$, where μ_t is vector where $\mu_{t,i} = p(s_t = i)$, $T_{i,j} = p(s_{t+1} = i|s_t = j)$: transition from one vector to next

- Markov Decision Process
  - $M = \{S, A, T, r\}$, adding action space and reward function
  - similar formulas, $\mu_{t,j} = p(s_t = j), \xi_{t,k} = p(a_t = k), T_{i,j,k} = p(s_{t+1} = i|s_t = j, a_t = k)$, so $\mu_{t,i} = \sum_{j,k} T_{i,j,k}\mu_{t,j}\xi_{t,k}$ (non-vectorized)
  - partially observed MDP: $M = \{S, A, T, O, \epsilon, r\}$ added O = observation space, ϵ = emission probability ($p(o_t|s_t)$)

- Goal of RL
  - trying to learn θ in $\pi_\theta(a|s)$, which can be represented as a neural network
  - can write probability of getting a SEQUENCE of states/actions: $p_\theta(s_1, a_1, ..., s_T, a_T) = p(s_1)\Pi_{t=1}^T \pi_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t)$ (getting the correct action * getting the correct next state)
    - the terms in the product follow Markov chain given a "state" of (s, a)
    - $p((s_{t+1}, a_{t+1})|(s_t, a_t)) = p(s_{t+1}|s_t, a_t)\pi_\theta(s_t, a_t))$ follows Markov's rule
  - GOAL: $\theta^* = argmax_\theta E_{\tau\sim p_\theta(\tau)}[\sum_t r(s_t, a_t)]$ (the arguments that get the highest expectation of rewards given the possible sequences of states/actions)
    - equivalent: $\theta^* = argmax_\theta 1/T \sum_{t=1}^T E_{(s_t,a_t)\sim p_\theta(s_t,a_t)}[r(s_t, a_t)]$, useful to find stationary distribution
    - Conditional expectations
      - $\sum_{t=1}^T E_{(s_t,a_t)\sim p_\theta(s_t,a_t)}[r(s_t, a_t)] = E_{s_1\sim p(s_1)}[E_{a_1\sim\pi(a_1|s_1)}[r(s_1, a_1)+ E_{s_2\sim p(s_2|s_1,a_1)}[E_{a_2\sim\pi(a_2|s_2)}[r(s_2, a_2) + ...]|s_1, a_1]|s_1]]$
      - Define $Q(s_1, a_1)$ s.t. this evaluates to $E_{s_1\sim p(s_1)}[E_{a_1\sim\pi(a_1|s_1)}[Q(s_1, a_1)|s_1]]$, we can easily modify the policy $\pi_\theta(a_1|s_1)$ if we know $Q(s_1, a_1)$
    - Q-function
      - $Q^\pi(s_t, a_t) = \sum_{t'=t}^T E_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t, a_t]$: total reward from taking a_t at s_t if you follow the policy
    - Value-function
      - $V^\pi(s_t) = \sum_{t'=t}^T E_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t]$: total reward from s_t if you follow the policy
      - $V^\pi(s_t) = E_{a_t\sim\pi_\theta(a_t|s_t)[Q^\pi(s_t,a_t)]}$
      - RL Objective can be written as $E_{s_1\sim p(s_1)}[V^\pi(s_1)]$
    - Using Q and Value-functions
      - can improve π if we know π and $Q(s, a)$ by setting $\pi'(a|s) = 1$ if $a = argmax_a Q^\pi(s, a)$

        - can compute gradients to improve prob of getting a better than average a: if $Q^\pi(s,a) > V^\pi(s)$, modify $\pi(a|s)$ to increase probability of a
    - anatomy of RL algorithm
        - run policy to generate samples
            - bottleneck is get real-time samples if not simulator
        - fit model/estimate returns
            - bottleneck is if model-based, getting a function to predict next state is hard
        - use to improve policy

7 8/29

- More general analysis, assuming states in same distribution as training set $p_{train}(s)$ follow $\pi_\theta(a \neq \pi^*(s)|s) \leq \epsilon$
    - when $p_{train}(s) \neq p_\theta(s)$
        - $p_\theta(s) = (1-\epsilon)^t p_{train}(s_t) + (1-(1-\epsilon)^t) p_{mistake}(s_t)$
        - or, $|p_\theta(s_t) - p_{train}(s_t)| = (1-(1-\epsilon)^t)|p_{mistake}(s_t) - p_{train}(s_t)| \leq 2 * (1-(1-\epsilon)^t)$ (summing over all states, which is why it's 2)
        - since $(1-\epsilon)^t \leq 1 - \epsilon t$ (by algebraic trick), $|p_\theta(s_t) - p_{train}(s_t)| \leq 2\epsilon t$
        - therefore, $\sum_t E_{p_\theta(s_t)}[c_t] = \sum_t \sum_{s_t} p_\theta(s_t) c_t(s_t) \leq \sum_t \sum_{s_t} p_{train}(s_t) c_t(s_t) + |p_\theta(s_t) - p_\theta(s_t)| c_{max} \leq \sum_t \epsilon + 2\epsilon t \leq \epsilon T + 2\epsilon T^2$, or $O(\epsilon T^2)$

8 8/24

- $\pi_\theta(a_t|o_t) = \pi_\theta(a_t|s_t)$ when world is fully observed, but not realistic
- s_{t+1} independent of s_t, a_t (previous states won't help determining future states), but the same cannot be said for o_{t+1} and its relationship to o_t
    - this makes sense because of what state means (whole world, including vectors for speed, etc.)
- Imitation Learning
    - interesting example with three different examples, approximates policy better since has samples outside of norm to correct for drift
    - training perfect policy is hard, so maybe change data to closely match p_{π_θ}: DAgger (Dataset Aggregation)
        - repeatedly add data by running policy but asking human to label with correct action, then retrain using this new policy to explore the entire space

        * assumption that humans can pick good actions
    - Why fail fit the expert?
        * Non-Markovian behavior (e.g. humans look at previous previous states instead of just previous)
            · can train on RNN or something using whole history
        * Multimodal behavior (e.g. humans are very complicated, can behave differently for same situation due to whatever constants)
            · can output mixture of gaussians
            · can use latent variable models
            · can use autoregressive discretization

- Rewards/Costs
    - $r(s, a)$ = reward function (negative cost function)
    - For imitation learning, $r(s, a) = \log p(a = \pi^*(s)|s)$ is good candidate, where asterisk is the master policy
        * other is $c(s, a) = 0$ if $a = \pi^*(s)$, 1 otherwise
        * analysis
            · assume $\pi_\theta(a \neq \pi^*(s)|s) \leq \epsilon$ for all $s \in D_{train}$
            · $E[\sum_t c(s_t, a_t)] \leq \epsilon T + (1-\epsilon)(\epsilon(T-1)+...)$, or T terms, each $O(\epsilon T)$, so $O(eT^2)$ total cost
            · for DAgger, $E[\sum_t c(s_t, a_t)] \leq \epsilon T$

9 8/22

- Decisions $\leftrightarrow$ observations, rewards
- Deep models allow solve reinforcement learning algos from end to end
- Interesting car jam situation with autonomous car
- Basic reinforcement learning = maximizing rewards but not enough to generalize to situations in real world
    - Learn from demonstrations
    - Observing the world
    - Learn from other tasks (transfer learning), inferring intentions
- Predicting what actions will do, or planning actions
- General learning in the future, have learning algorithm(s) for all situations
    - What must this single algorithm do
        * Interpret rich sensory inputs
        * Choose complex actions

  - Deep = interpret, reinforcement learning = complex actions

- Brain actually seems to work like deep reinforcement learning (deep learning and reinforcement learning), like basal ganglia = reward function
- NOW: deep rl great in simple rules, learn simple skills, learn from imitation
- CHALLENGING: humans learn much faster, humans can do transfer learning (OPEN PROBLEM), not clear what reward function should be, not clear what role of prediction should be
- thought: human main purpose is to sustain life, it may be beneficial for the reward function to be "negative" in the sense that any move that isn't optimal causes neurons in the neural net to be removed at random, this actually mimics real life when neurons are being pared during development
  - the objective would be to keep as many neurons as possible
  - if mistakes are made, then the neural net would become smaller but more specialized for the task at hand
  - if we start with a really large neural net (babies have really large brains), we won't run into having a neural net that cannot generalize to the problem at hand
  - couple this with prediction