CS209b Section : Deep Reinforcement Learning

Instructor : Pavlos Protopapas, TF : Srivatsan Srinivasan

27 March 2019

Abstract

In this tutorial, we learn how deep learning is utilized in the context of reinforcement learning. We discuss a wide summary of algorithms of the two popular classes : value function approximations and policy approximations. In the former we discuss parametric Q-learning followed by an introduction to Deep-Q Networks(DQN) and some iterative improvements to the baseline model - Double and Dueling DQN. We then discuss policy gradients, actor critic models, deterministic policy gradients and trust region policy optimization methods. followed by The ultimate aim of the tutorial is to facilitate the readers to become self-sufficient with the Deep RL fundamentals in order to comprehend the essentials of the state-of-the-art literature in this fast-growing domain.

1 Q-Learning and Parametric Q-Learning

The State-Action Value Function/Action Value function (or "Q-function") takes two inputs: "state" and "action." It returns the expected future reward of that action at that state assuming you follow the behavior policy from there-on. Q-Learning is a off-policy model-free TD Learning algorithm that learns these Q-values using only experiences sampled from the environment. In a nutshell, the algorithm can be described as follows.

- Initialize Q(s,a) for all state-action value pairs.
- Do steps 1 to 3 until convergence of Q-values
    1. Choose an action a based on current Q estimates using a ϵ- greedy policy.
    2. Take the action and observe the transition (s,a,r,s')
    3. Update $Q(s,a) := Q(s,a) + \alpha\bigg(r(s,a) + \gamma \max_a' Q(s',a') - Q(s,a)\bigg)$

This model is off-policy because Q-value update and action selection follows two different policies. We call the former the target policy (which is an approximation of the optimal policy - look at the max term) and the latter as the behavior policy as it is the policy that the agent executes on the environment.

ϵ-greedy Q-Learning - It is not always advisable to choose an action that maximizes the Q-value (self-serving). We would want to explore the environment and record enough transitions before we start trusting our Q-values. Thus, in an ϵ greedy policy, we choose an action that maximizes Q-values with $1-\epsilon$ and choose a random action with probability ϵ. The choice of ϵ throughout our learning balances the classic "exploration vs exploitation" tradeoff.

1.1 Need for parametric Q-Learning and linear Q-Learning Example

The number of Q-values that you hold in memory varies with the complexity of your state action space. To state this formally, $|Q| = |\mathcal{S}| \times |\mathcal{A}|$. Imagine real world problems where model-free RL is applied - for instance, learning to drive using videos of the external environment. Here, the state space is continuous whose cardinality is infinite. It is also fair to assume that the Q-function is fairly smooth and continuous in our state space $\mathcal{S}$ i.e. estimates of Q-values on states adjacent to each other are not arbitrarily different. Hence, we can parametrize Q-values with functional transformations on the state action space i.e. $Q(s,a) :\to f_\theta(s,a)$ and still employ the same algorithm as we saw earlier and train the model based on the TD-error between estimated targets and true function values.

A simple parametric form is linear function approximation and we can dig deeper on how it works. In Q-Learning, when we approximate a state-action value Q(s,a), we want it to be equal to the reward obtained, plus the (appropriately discounted) value of playing optimally from thereafter. Denote this "target" value as $Q^+(s,a)$. In the tabular version, we set the target as

$$Q_\theta^+(s,a) = \sum_{s'} P(s'|s,a)\big[R(s,a,s') + \gamma \max_{a'} Q_\theta^+(s',a')\big] \tag{1}$$

It is infeasible to sum over all states, but we can get a mini-batch estimate of this by simply considering the samples that the agent collects : $Q^+(s,a) \approx R(s,a,s') + \gamma \max_a Q^+(s,a)$. Averaged over the entire run, these is expected to average out to the true value. Assuming that our Q_θ truly follows the Bellman Equation, we can therefore define our loss as:

$$\mathcal{L}(\theta^{(i)}) = \frac{1}{2}\sum_i \big[Q_\theta^+(s_i,a_i,s_i') - Q_\theta(s_i,a_i)\big]^2 = \frac{1}{2}\sum_i \big[Q_\theta^+(s_i,a_i,s_i') - \theta^{(i)} \cdot \phi(s_i,a_i)^T\big]^2 \tag{2}$$

$Q^+(s,a)$ can be obtained using our parametric approximation and TD-Update in Equation 1. The gradient update with ρ as learning rate can be derived as

$$\theta^{(i+1)} = \theta^{(i)} - \rho \cdot \sum_i \big(Q_\theta^+(s_i,a_i,s_i') - \theta^{(i)} \cdot \phi(s_i,a_i)^T\big) \cdot \phi(s_i,a_i) \tag{3}$$

2 Deep Learning + RL

Applications of deep learning in Reinforcement Learning have been continuously on the rise in the last 5 years. Some salient examples of extensive RL applications accelerated by deep learning from recent times include games (AlphaGo for Go, AlphaZero for chess, shogi and go OpenAI Five for DOTA 2, DeepStack for poker), robotics (autonomous driving, path planning, UAV control etc.), natural language(summarization, question answering, visual dialogue, program synthesis, knowledge graph reasoning), finance (option pricing, portfolio optimization) and healthcare (ICU interventions, psychiatry treatments). Integration of neural networks into RL models can be broadly categorized into three classes

- Value based (DL value function approximators)
- Policy based (DL policy approximators)
- Model based (DL approximator for transition dynamics model)

In this tutorial, we learn a few models from only the first two classes owing to time constraints.

3 Value based Deep RL

In this class of algorithms, usually the Q-value function is represented by a neural network parametrization and updates happen iteratively to Q-values using Bellman updates until convergence.

3.1 DQN, Target Networks and Experience Replay Buffers

Deep Q-Networks differ from linear Q-learning in that the Q-values are parametrized by a deep neural network. This parametrization could take different forms - for instance in simple continuous data, we could use a feedforward net, for learning to play from video demonstrations, we can use a deep convolutional network etc. The algorithm is trained using minibatch gradient descent based on the following loss function assuming we define our target $Q^{+}(s,a)$ similar to what we did in the previous section(with Q_θ representing a neural net parametrization of the Q-values)

$$\mathcal{L}(\theta^{(i)}) = \frac{1}{2}\big[Q^{+}(s,a) - Q(s,a)\big]^2 = \frac{1}{2}\big[Q^{+}(s,a) - Q_\theta(s,a)\big]^2 \tag{4}$$

A basic assumption of several deep learning models is that the samples that it encounters and trains on are i.i.d. In settings such as DQN, we collect a sample for every action and we can potentially update our network weights with each sample collected(or over a very small buffer of previous samples). This could be problematic with respect to the i.i.d. assumptions since there could be strong correlations between the training samples as current actions have strong implications on future states and actions in reinforcement learning settings and our current course of action essentially determines our future samples. To resolve this problem, DQN uses experience replay where it builds a data buffer (of a pre-determined size) and aggregates its past few experiences into the buffer. Every time it needs to update the model, it samples a set of transitions from the buffer rather and performs SGD updates on the loss function that we saw in Equation 4. This kind of an experience replay buffer offers two key advantages

- More efficient use of previous experience, by learning with it multiple times. Remember Q-learning updates are incremental and do not converge to optimality immediately.
- It helps break correlation between training samples to a large extent and thus satisfies the i.i.d. samples assumption of neural net regression.

The other disadvantage with the initial setup of DQN is the non-stationarity of the targets. Remember the targets are treated as pseudo "ground-truths" and we update the parameters of the network via SGD for our Q-values to reach the target. If the targets are non-stationary, it leads to poor and extremely noisy learning of Q-values (think how the training will look like when you change the loss function in each iteration). DQN fixes this by using a target network with parameters θ^-, which is essentially a lagged version of the online network with parameters θ and the target network is updated once every τ steps with $\theta^- = \theta$. The targets are computed using θ^- instead of θ and they remain stationary till the parameter update happens.

$$Q^{+}(s,a) = \mathbb{E}_{s' \sim \mathcal{D}_{\text{batch}}}[R(s,a,s') + \gamma \max_{a'} Q_{\theta^-}(s',a')]$$

The pseudo-code for training DQN is highlighted with the following steps (should be repeated over several episodes)

1. Get initial state s_0 for the episode. Need : θ, θ^-, replay buffer $\mathcal{D}_{\text{rep}}$
2. Repeat steps 3-8 till the episode terminates
3. Select action with Q-values ($Q_\theta(s,a)$) for s_t in ϵ greedy fashion.
4. Execute a_t and record transition (s,a,r,s') in $\mathcal{D}_{\text{rep}}$.
5. Sample random mini-batch of K of transitions (s_i, a_i, r_i, s_i') from $\mathcal{D}_{\text{rep}}$.
6. Set the targets

$$Q^+(s_i, a_i, s_i') = \begin{cases} R(s_i, a_i, s_i'), & \text{if } s_i' \text{ is terminal.} \\ R(s_i, a_i, s_i') + \gamma \max_{a'} \mathrm{Q}_{\theta^-}(s_i', a'), & \text{otherwise.} \end{cases} \tag{5}$$

7. Perform SGD update for θ using the loss function $\frac{1}{2}\sum_{i=1}^{K}[Q^+(s_i, a_i, s_i') - Q_\theta(s_i, a_i)]^2$
8. Every C steps, reset the target network. $\theta^- = \theta$

3.2 Over-Optimism bias of DQNs and Double DQNs

Remember how we calculate the TD-target

$$\underbrace{Q(s,a)}_{\text{target}} = \underbrace{R(s,a)}_{\text{Reward}} + \underbrace{\gamma \max_{a'} Q(s',a')}_{\text{Discounted max possible Q-value from s'}}$$

In this setup, we face a direct problem: how are we sure that the best action for the next state is the action with the highest Q-value? We know that the accuracy of Q values depends on what action we tried and what neighboring states we explored till now. As a consequence, in the early phase of the training, we don't have enough information about the best action to take. Therefore, taking the maximum Q value (which is noisy) as the best action to take can lead to false positives. If non-optimal actions are regularly given a higher Q value than the optimal best action, the learning will be complicated and the network wont converge reasonably.

The idea of Double Q-learning is to reduce over-estimations by decomposing the max operation in the target term into action selection and action evaluation.

- Use our DQN network to select what is the best action to take for the next state (the action with the highest Q value).
- Use our target network to calculate the target Q value of taking that action at the next state.

Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks. This effectively changes the non-terminal part of step 6 of the psuedo-code for DQN we saw earlier.

$$Q^+(s,a,s') = R(s,a,s') + \gamma \underbrace{Q_{\theta^-}(s', \underbrace{\arg\max_{a'}(Q_\theta(s',a'))}_{\text{selection with online net}})}_{\text{evaluation with target net}}$$

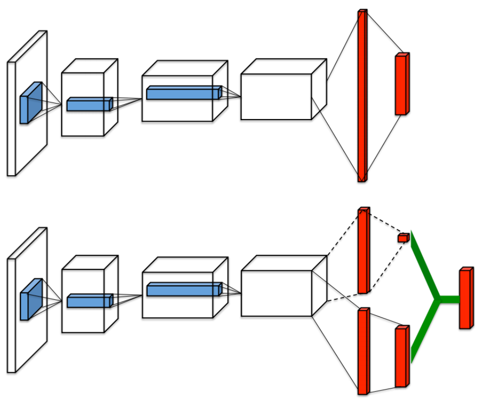

Figure 1: DQN(top) vs Dueling DQN(bottom) Look at how dueling DQN decouples value and advantage function into two channels and thus trains faster and more robustly.

3.3 Dueling DQN

Remember that Q-values correspond to how good it is to be at that state and taking an action at that state Q(s,a). So we can decompose Q(s,a) as the sum of V(s): the value of being at that state and A(s,a): the advantage of taking that action at that state (how much better is to take this action versus all other possible actions at that state). With dueling DQN, we want to separate the estimator of these two elements, using two new streams and then aggregate them after learning them independently. By decoupling the estimation, intuitively the Dueling-DQN (Figure 1) can learn which states are (or are not) valuable without having to learn the effect of each action at each state (since it's also calculating V(s)). Realize that there is no point in evaluating all actions from a state when the value of the state is bad.Therefore, this architecture helps us accelerate the training. We can calculate the value of a state without calculating the Q(s,a) for each action at that state. And it can help us find much more reliable Q values for each action by decoupling the estimation between two streams. For more details on the theoretical proofs and training, refer the original paper - https://arxiv.org/pdf/1511.06581.pdf

4 Policy Based Reinforcement Learning

4.1 Policy Gradients, REINFORCE

The policy gradient(PG) methods target at modeling and optimizing the policy directly. The policy is usually modeled with a parameterized function respect to $\theta, \pi_\theta(a|s)$. In these PG algorithms we define the returns of the policy as :

$$J(\theta)^1 = \sum_{s \in \mathcal{S}} d^\pi(s) V^\pi(s) = \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q(s,a) \tag{6}$$

[1] Verify that J(π) $= \mathbb{E}_\pi(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)) = \mathbb{E}_{s_0} V^\pi(s_0)$

where $d_\pi(s)$ is the stationary distribution of Markov chain for π_θ i.e. $d^\pi(s) = \lim_{t\to\infty} P(s_t = s|s_0, \pi_\theta)$ is the probability that $s_t = s$ when starting from s_0 and following the policy π_θ for t steps. We can use gradient ascent to update the parameters of θ till optimal returns. Typically, policy gradient approaches work well for large continuous state-action spaces because value based algorithms would be computationally intractable in these cases (e.g. taking max over a continuous state space of actions).

Theorem 1 (Policy Gradient Theorem)**.** *If we define $\mathbb{E}_\pi = \mathbb{E}_{s\sim d^\pi, a\sim\pi_\theta}$, then we can rewrite the gradient of the returns under a policy π_θ as $\nabla_\theta J(\theta) = \mathbb{E}_\pi[Q_\pi(s,a)\nabla_\theta \log \pi_\theta(a|s)]$*

Proof. In the proof, let us label the probability of transitioning from state s to state x with policy π_θ after k steps as $\rho_\pi(s \to x, k)$. It is obvious that $\rho_\pi(s \to s, k=1) = \sum_a \pi_\theta(a|s)P(s'|s,a)$ and we can recursively write $\rho_\pi(s \to x, k+1) = \sum_{s'} \rho_\pi(s \to s', k)\rho_\pi(s' \to x, 1)$

$$\begin{aligned}
\nabla_\theta V^\pi(s) &= \nabla_\theta \sum_{a\in\mathcal{A}} \pi_\theta(a|s)Q^\pi(s,a) \\
&= \sum_{a\in\mathcal{A}} \nabla_\theta\pi_\theta(a|s)Q^\pi(s,a) + \pi_\theta(a|s)\nabla_\theta Q^\pi(s,a) \\
&= \underbrace{\sum_{a\in\mathcal{A}} \nabla_\theta\pi_\theta(a|s)Q^\pi(s,a)}_{\phi(s,a)} + \pi_\theta(a|s)\sum_{s'} P(s'|s,a)\nabla_\theta V^\pi(s') \\
&= \phi(s) + \sum_{s'}\sum_{a} \pi_\theta(a|s)P(s'|s,a)\nabla_\theta V^\pi(s') \\
&= \phi(s) + \sum_{s} \rho_\pi(s \to s', 1)\nabla_\theta V^\pi(s') \\
&= \phi(s) + \sum_{s} \rho_\pi(s \to s', 1)[\phi(s') + \rho_\pi(s' \to s'', 1)\nabla_\theta V^\pi(s'')] \\
&= \sum_{x\in\mathcal{S}}\sum_{k=0}^{\infty} \rho_\pi(s \to x, k)\phi(x) \\
\nabla_\theta J(\theta) &= \sum_{s}\sum_{k=0}^{\infty} \rho_\pi(s_0 \to s, k)\phi(s) \\
&= \sum_{s} d^\pi(s)\sum_{a} \nabla_\theta\pi_\theta(a|s)Q^\pi(s,a) \\
&= \sum_{s} d^\pi(s)\sum_{a} \pi_\theta(a|s)Q^\pi(s,a)\frac{\nabla_\theta\pi_\theta(a|s)}{\pi_\theta(a|s)} \\
&= \mathbb{E}_\pi[Q_\pi(s,a)\nabla_\theta \log\pi_\theta(a|s)] \\
&= \underbrace{\frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t=1}^{T}R(s_{i,t},a_{i,t})\right)}_{\mathbb{E}\text{ approximated with N rollouts of the policy}}
\end{aligned}$$

□

We can also reduce variance in our policy gradients expression by subtracting a baseline term.

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_\pi[Q_\pi(s,a)\nabla_\theta \log \pi_\theta(a|s)] \\ &= \mathbb{E}_\pi[(A_\pi(s,a) + V(s)) \cdot \nabla_\theta \log \pi_\theta(a|s)] \\ &= \mathbb{E}_\pi[A_\pi(s,a)\nabla_\theta \log \pi_\theta(a|s)]\end{aligned} \tag{7}$$

The policy gradient theorem lays the theoretical foundation for various policy gradient algorithms which we are going to learn in the following sections.

REINFORCE algorithm uses Monte-Carlo rollouts to compute the expectation of the policy gradients theorem. The algorithm would entail sampling N trajectories $\tau_i; i = 1, ...N$, computing the rollouts based gradients and updating the policy based on gradient ascent on $J^\pi(\theta)$.

4.2 Actor-Critic Algorithms

Two main components in policy gradient are the policy model and the value function. It makes a lot of sense to learn the value function in addition to the policy, since knowing the value function can assist the policy update, such as by reducing gradient variance in vanilla policy gradients. Actor-Critic algorithms implement this idea broadly where they consist of two models which may optionally share parameters.

- **Critic** updates the value function parameters w and different variants could either model the Q-function $Q(s, a; w)$ or the value function $V(s; w)$
- **Actor** updates the policy parameters θ for $\pi_\theta(a|s)$ in the direction suggested by the critic.

The general training procedure for any actor critic algorithm can be summarized as follows. Note that several improvements we learned with policy and value based algorithms (replay buffer, dueling etc.) can be used to extend this basic version of the algorithm.

1. Initialize s, θ, w at random; sample $a \sim \pi_\theta(a|s)$.
2. Do steps 2-7 till episode terminates.
3. Sample next reward r_t and next state s'.
4. Sample next action $a \sim \pi_\theta(a|s)$
5. Update policy network $\theta \leftarrow \theta + \alpha_\theta Q(s, a; w)\nabla_\theta \log \pi_\theta(a|s)$
6. Compute the TD error for action-value at time t: $\delta_t = r_t + \gamma Q(s', a'; w) - Q(s, a; w)$ and use it to update the parameters of action-value function: $w \leftarrow w + \alpha_w \delta_t \nabla_w Q(s, a; w)$
7. Update $a \leftarrow a'$ and $s \leftarrow s'$.

Asynchronous Advantage Actor-Critic(A3C) is a special version of actor-critic algorithms that are designed to work well for parallel training. IN A3C, the critics learn the value function while multiple actors are trained in parallel and get synced with global parameters from time to time. **A2C** is a synchronous version of the same algorithm where there is a coordinator node across the learners. The details and code for different parallel/distributed variants of DQN and A3C can be obtained from the original paper https://arxiv.org/pdf/1602.01783.pdf.

4.3 Off-Policy Policy Gradients

Both REINFORCE and A3C perform policy gradients in on-policy setting where the training samples are collected according to the target policy — the very same policy that we try to optimize for. As we saw in Q-Learning, off policy methods offer few salient advantages - a.) allows for better exploration since behavior policy differs from that of target, b.) does not require sequential transitions in the form of trajectories and can use samples from replay buffers. Let us define the behavior policy for collecting samples as a known policy (predefined just like a hyperparameter), labelled as $\beta(a|s)$. The objective function sums up the reward over the state distribution defined by this behavior policy

$$\begin{aligned} J(\theta) &= \sum_{s\in\mathcal{S}} d^{\beta}(s) \sum_{a\in\mathcal{A}} Q^{\pi}(s,a)\pi_\theta(a|s) \\ &= \mathbb{E}_{s\sim d^\beta}\left[\sum_{a\in\mathcal{A}} Q^{\pi}(s,a)\pi_\theta(a|s)\right] \\ \nabla_\theta J(\theta) &\approx \mathbb{E}_{s\sim d^\beta}\left[\sum_{a\in A} Q^{\pi}(s,a)\nabla_\theta\pi_\theta(a|s)\right] \quad \text{Omitted a term. \underline{Detailed Proof}} \\ &= \mathbb{E}_\beta\left[\underbrace{\frac{\pi_\theta(a|s)}{\beta(a|s)}}_{\text{Importance weight}} \cdot Q^{\pi}(s,a)\nabla_\theta \log\pi_\theta(a|s)\right] \quad \text{Multiply and divide by } \beta(a|s) \end{aligned} \tag{8}$$

Using importance weights, applying policy gradient in an off-policy setting essentially amounts to taking a weighted sum (with importance weights) of ordinary policy gradients. While we do not explicitly state them here, this form of off-policy policy gradients can be incorporated into several versions of PG algorithms that we have seen earlier and about to see in the following sections.

4.4 DPG, DDPG

In methods described above, the policy function $\pi(.|s)$ is always modeled as a probability distribution over actions A given the current state and thus it is stochastic. Deterministic policy gradient (DPG) instead models the policy as a deterministic decision: $a = \mu(s)$. As before, the objective to optimize for is the overall returns under the policy $J(\theta) = \int_{\mathcal{S}} \rho^{\mu}(s)Q(s,\mu_\theta(s))ds$. We can use chain rule to see that

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{s\sim\rho^\mu}\left[\nabla_a Q^{\mu}(s,a)\nabla_\theta\mu_\theta(s)\mid_{a=\mu_\theta(s)}\right] \\ &\approx \frac{1}{N}\sum_i \nabla_a Q(s_i,a;\theta)\mid_{a=\mu_\theta(s_i)} \nabla_\theta\mu_\theta(s) \end{aligned} \tag{9}$$

We can consider the deterministic policy as a special case of the stochastic one, when the probability distribution contains only one extreme non-zero value over one action. Compared to the deterministic policy, we expect the stochastic policy to require more samples as it integrates the data over the whole state and action space and hence DPG algorithms are easier to train. The deterministic policy gradients can be plugged into any policy gradient frameworks.

DDPG - Deep Deterministic Policy Gradient is a model-free off-policy actor-critic algorithm, combining DPG with DQN. Recall that DQN (Deep Q-Network) stabilizes the learning of

Q-function by experience replay and the frozen target network. The original DQN works in discrete space, and DDPG extends it to continuous space with the actor-critic framework while learning a deterministic policy. In order to promote exploration within the training phase, DDPG constructs a noisy deterministic policy $\mu'(s) = \mu_\theta(s) + \mathcal{N}$. The following steps define the procedure of DDPG in each episode.

1. Initialize critic network $Q(s, a; \theta^Q)$,actor $\mu(s; \theta^\mu)$, set target networks Q', μ' with weights $\theta^{Q'}, \theta^{\mu'}$, a random noise process $\mathcal{N}$, random buffer D_{exp} and receive initial observation s_1.
2. Repeat steps 3-8 till episode terminates.
3. Select action $a_t = \mu_{\theta^\mu}(s_t) + \mathcal{N}$ balancing exploration and exploitation.
4. Execute and collect transition to store (s_t, a_t, r_T, S_{t+1})in buffer D_{exp}.
5. Sample a random minibatch N of transitions (s_i, a_i, r_i, s'_i) from D_{exp}.
6. Set target $Q_i^+ = r_i + \gamma Q'(s'_i, \mu'(s'_i; \theta^{\mu'}); \theta^{Q'})$
7. Update critic by minimizing TD-error loss $\sum_{i=1}^{N}(Q_i^+ - Q(s_i, a_i; \theta^Q))^2$
8. Update the actor policy using the sampled policy gradient from equation 9.
9. Update both of the target networks (Soft updates using $\tau << 1$) $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$ and $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$

4.5 TRPO, PPO

The last class of policy gradient algorithms we study are the trust-region algorithms. When we are updating policy parameters, we do not want the agent's policy to differ significantly between each update - if it does, training will be volatile and hence not efficient. Before we define Trust Region Policy Optimization(TRPO), let us try to understand two other essentials.

Trust Region - There are two major optimization methods: line search and trust region. Gradient descent is a line search. We determine the descending direction first and then take a step towards that direction. In the trust region, we determine the maximum step size that we want to explore and then we locate the optimal point within this trust region. Mathematically this could bee formulated into an optimization problem

$$\max_x m(x) \quad \text{such that} \quad ||x|| < \delta$$

where m(x) is some local approximation to the function we would like to minimize and δ defines the trust radius. We adjust the trust radius during our training based on the n-th order derivatives of the function. THINK :Advantages/ Disadvantages of trust region over GD.

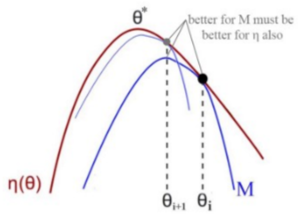

Figure 2: Minorize Maximization principle

Minorize-Maximization Algorithm The main idea of MM (minorization-maximization) algorithms is that, intuitively, for a maximization problem, we first find a approximated lower bound of the original objective as the surrogate objective and then maximize the approximated lower bound so as to optimize the original objective. Let $\eta(\theta)$(red line in Figure 2) be the function we want to optimize and let M represent our local approximation which is a lower bound (it is assumed M is easy to optimize). Based on our assumptions, it is guaranteed that at each step, the guess θ_{i+1} is an improvement over θ_i.

TRPO - Trust Region Policy Optimization : Using the definition of advantage functions and return functions that we learned earlier, we can prove the following(assuming π is known and $\hat{\pi}$ is unknown)

$$J(\hat{\pi}) = J(\pi) + \sum_s \rho_{\hat{\pi}}(s) \sum_a \hat{\pi}(a|s) A_\pi(s,a)$$

This is difficult to optimize directly because of the dependency on $\rho_{\hat{\pi}}$. Hence, TRPO defines a surrogate function ignoring changes in state visitation density due to changes in the policy. For parametric policies, this approximation matches the true function to the first order(Proof in paper)

$$L(\hat{\pi}) = J(\pi) + \sum_s \rho_\pi(s) \sum_a \hat{\pi}(a|s) A_\pi(s,a)$$

TRPO essentially uses the two ideas we learned earlier in performing policy optimization. It uses the KL divergence between the old policy and updated policy as a measurement for the trust region. They also develop a surrogate loss function that is a lower bound of the true objective - the expected cumulative return of the policy. Under parametric settings(with a neural net policy), let θ_{old}, θ refer to the parameters in previous iteration and the parameters we would like to update in the current iteration.

$$\begin{aligned} L(\pi_\theta) &= J(\pi_{\theta_{old}}) + \sum_{s\in\mathcal{S}} \rho^{\pi_{old}} \sum_{a\in\mathcal{A}} \pi_\theta(a|s) \hat{A}_{\theta_{old}}(s,a) \\ &= \mathbb{E}_\pi \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} \hat{A}_{\theta_{old}}(s,a) \end{aligned}$$

$$\text{TRPO objective} : \max L(\pi_\theta)$$
$$\text{such that} \quad D_{KL}(\pi_{\theta_{old}}(.|s)||\pi_\theta(.|s)) < \delta$$

PPO - Proximal Policy Optimization : Given that TRPO is relatively complicated and we still want to implement a similar constraint, proximal policy optimization (PPO) simplifies it by using a clipped surrogate objective while retaining similar performance. Remember the TRPO loss function

$$J^{TRPO}(\theta) = \mathbb{E}\Big[\underbrace{\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}}_{r(\theta)}\hat{A}_{\theta_{old}}(s,a)\Big]$$

If we try to just optimize this without any relative constraints between the two policies, we would succumb to instability with extremely large parameter updates and big policy ratios. PPO imposes a different constraint on this ratio $r(\theta)$ to stay within a small interval around 1.

$$J^{CLIP}(\theta) = \mathbb{E}\Big[\min\Big(r(\theta)\hat{A}_{\theta_{old}}(s,a), \quad \text{clip}(1-\epsilon, r(\theta), 1+\epsilon)\hat{A}_{\theta_{old}}(s,a)\Big)\Big]$$

The objective function of PPO takes the minimum one between the original value and the clipped version and therefore the model loses any motivation for increasing the policy update to extremes for better rewards. When applying PPO on the network architecture with shared parameters for both policy (actor) and value (critic) functions, in addition to the clipped reward, the objective function is augmented with an error term on the value estimation and an entropy term to encourage sufficient exploration.

$$J^{PPO}(\theta) = \mathbb{E}\Big[J^{CLIP}(\theta^\pi) - c_1\underbrace{\sum_k (V^+(s) - V_{\theta^V}(s))^2}_{\text{usual TD loss}} + c_2 H(s, \pi_\theta(.))\Big]$$

5 Conclusion

Overall, this tutorial presented an extremely high level introduction to two broad classes of deep RL methods - value based (in which we performed value/policy iteration) and policy based (direct parametrization and gradient based optimization). Beyond the procedure wise details of these algorithms, there have been several salient RL modeling ideas that are common across these algorithms such as variance reduction(while keeping bias unchanged), Off policy methods as more data efficient and exploratory, experience replay buffers, target networks as lagged versions of online networks to prevent overoptimism, entropy regularization to promote exploration, shared parameter networks(DDQN, critic+actor), deterministic PG as a single point approximation of PG, preventing divergence between policy updates, exploring convex and approximate convex optimization literature for new methods.

6 Credits

This tutorial has been prepared predominantly with materials from the following links(blogs and papers) and the staff of CS209 sincerely thank the authors for their work.

1. https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html
2. https://icml.cc/2016/tutorials/deep_rl_tutorial.pdf
3. http://178.79.149.207/posts/trpo.html
4. http://proceedings.mlr.press/v32/silver14.pdf
5. https://arxiv.org/pdf/1511.06581.pdf
6. https://arxiv.org/pdf/1509.06461.pdf
7. https://arxiv.org/pdf/1502.05477.pdf
8. https://arxiv.org/pdf/1602.01783.pdf
9. https://arxiv.org/abs/1707.06347