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

Section 10: Value-based Methods and Exploration

Notes by: Forrest Huang

10.1 Course Logistics

- Final project poster is due Saturday 5/4. We will post the requirements for the poster and the write-up soon. You will be submitting the poster electronically, and we will print them out for you. Please follow the requirements carefully or else your poster might not be printed.
- Please fill out the course survey. Your opinions are valuable for us to make this class better in the future.

10.2 Value-based Method

Recall last time that we introduced policy gradient approaches for solving Reinforcement Learning problems. In this section, we discuss another group of RL algorithms: Value-based methods. Instead of deducing an optimal policy for policy-gradient methods, value-based methods optimize towards finding a good approximation of the expected rewards at various states. We consider this approximation the *Value Function* $V(s)$. That is, instead of attributing our rewards to our policy, attribute them to the states and actions sets of trajectories.

10.2.1 Value Function

Given a state s_i, the value function with respect to a given policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ should provide you the expected reward of all trajectories passing through this state assuming the agent executes policy π. Hence, we can define the state's value $V^{\pi}(s_i)$ as the (discounted) reward-to-go from that state:

$$V^{\pi}(s_i) = \sum_{a \in \mathcal{A}} \pi(a_i|s_i) \left(r(s_i, a_i) + \gamma \sum_{s_{i+1}} V^{\pi}(s_{i+1}) p(s_{i+1}|s_i, a_i) \right)$$

where above, we assume our set of states $\mathcal{S}$ and actions $\mathcal{A}$ is discrete[1], and that $\gamma \in [0, 1]$ is the *discount factor* (0.99 is a common setting). While this defines the value function that formulates our optimization goal, when optimizing a network towards a solution, we can choose other formulations of the 'Value Functions.' Typically, since our MDP trajectory involves *both states and actions*, it is straightforward to use a state-action-specific 'Value function' before we can derive the $V(s)$.

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[r_0 + \gamma r_1 + \gamma^2 r_2 ... | r_0 = r(s, a)]$$

[1] If you've taken CS 188 at Berkeley, this is the definition you've likely encountered, since it's common to introduce value functions in the tabular setting, where we can explicitly enumerate and quantify the value of all possible combinations of states and actions.

This gives you a state-action-value function that approximates the potential rewards by taken a certain action a at a certain state s. This is also called the Q-function which is used in DQN and DDQN, as explained in the later sections of this paper. With this, we can arrive at $V^{\pi}(s)$ using expected values:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[r_0 + \gamma r_1 + \gamma^2 r_2 ...] = \mathbb{E}_{a\sim\pi}[Q^{\pi}(s,a)]$$

which takes into account all possible actions taken from the current state in our current policy.

In addition to state values and state-action values, the *advantage function* is another common class of value functions used:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

which provides you with an estimation of how good a certain action is compared to all actions on average, and hence the 'advantage' you get by taking certain actions.

10.2.2 DQN with replay buffers

With the formulation of these value functions, we seek to optimize these functions using a deep network, which is introduced as Deep Q-learning.

As you have implemented in the last homework, Deep Q-learning optimizes towards learning the expected reward at certain state-action pairs, such that it measures the difference between our Q-function's estimation and the sum of reward at current state plus the maximum Q-function estimation of the upcoming state, we should estimate the reward of the entire trajectory towards the end.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 10.1: The DQN algorithm, taken straight from the Nature 2015 paper.

DQN is often referred to as having two key technical accomplishments over the "straightforward idea" of

simply adding a deep neural network to one-step Q-learning.[2] The first is that *replay buffers* in DQN are used to de-correlate sequential states. Second, the fixed and delayed update between current and target networks solve the problem of instability in defining a target value for Q-learning.

10.2.3 Double DQN with replay buffers

Double DQN solves the overestimation of Q-function of the next value which the selection of actions is done using the same network as computing the value function. This can be resolved by using two different networks for action selection and value approximation in the target, which in practice just uses the current and target network in DQN.

10.3 Comparison between Value-based and Policy Gradient methods

10.3.1 On-policy and Off-policy learning

A large difference between these two sets of algorithms is that policy gradient methods are typically on-policy, whereas the value iteration methods mentioned above (DQN, Double-DQN) are off-policy. Recall that in RL, an agent explores in the environment by following its *behavioral* or *exploration* policy, but it really wants to learn an optimal policy (or *target* policy) from its exploration.

1. In *on-policy* algorithms, the learner uses it's own experience and derives a policy that will again be used for exploration, so the behavioral policy matches the target policy.
2. In *off-policy* learning, the exploration policy is not the same as the target policy. In the case of off-policy Q-learning methods, the learner may only have access to the experience of the agents, and does not have control over the policy that was actually used to get those states. Nonetheless, assuming the set of states matches certain conditions (e.g., that it sufficiently covers the overall state) Q-learning can be guaranteed to converge to an *optimal* policy (under tabular settings, not the function approximation case unfortunately) despite how the samples it uses for learning came from a different policy.

To illustrate the distinction between the two categories, we compare between Q-learning and SARSA, which is respectively an off-policy and on-policy learning algorithm with similar optimization objectives.[3]

In Q-learning, the objective is:

$$Q_{\theta'}(a,s) \leftarrow Q_\theta(a,s) + \alpha(r_s + \gamma \max_{a'} Q_\theta(a',s') - Q_\theta(a,s))$$

whereas in SARSA:

$$Q_{\theta'}(a,s) \leftarrow Q_\theta(a,s) + \alpha(r_s + \gamma Q_\theta(a',s') - Q_\theta(a,s))$$

This means SARSA, all updates follow the trajectories taken by the current policy that in terms depends on the Q-function. Whereas in Q-learning, the learning objective only takes a greedy estimation of the

[2] Though to be clear, neither of these two are novel in a research sense, as prior work had done these, but DeepMind was the first to get these engineered and working in practice on complicated environments like the Atari 2600 games.

[3] For more details, see Chapter 6 of the RL book: `http://incompleteideas.net/book/RLbook2018.pdf`.

policy (by taking max) but does not need to be coupled with the policy taken in the exploration phase. For instance, in deep-Q learning, with a probability of ϵ the exploration step does not follow the Q-function, but sample an action randomly to encourage *exploration*. As such, Q-learning requires only one extra step in the end to extract the policy, which could be greedy by picking the max Q-function since this will produce the best policy when converged to optimality.

Similarly, we can deduce that other kinds of policy-gradient algorithms are on-policy algorithms, such that the optimization target involves directly the trajectories generated by the current policies, in order to optimize the current policy. This process is done iteratively in policy-gradient methods.

As such, policy gradient methods can only be used when there is control over the exploration agent's action in the environment, while value-based methods can be more flexible and potentially even learn from existing trajectories.

10.3.2 Discrete vs. Continuous Action Space

Because policy-gradient methods learn policies which are distributions of actions given a state, compared to value-based methods which use greedy selection of actions in the optimization objective, policy gradients methods are usually preferred in *continuous* action domains. Using discretization can be one potential way to solve this problem for value-based methods.

10.3.3 Sampling from slow environments

Since policy-gradient methods can only use trajectories once before the policy is updated, it often requires more samples of trajectories from the environment to learn an optimal policy. In contrast, value iteration methods can re-use trajectory samples since it is not dependent on the policy that was taken to collect those samples. As such, value-iteration methods are more suitable for environments that may take a long time to sample from.

10.4 Exploration

Selecting adequate exploration strategies is important to arriving at an optimal solution in RL in addition to optimizing rewards in current experiences of the agents. In particular, both policy gradients and value-based methods rely on states already explored by π for the optimization to explore it. One simple way to resolve that is the epsilon-greedy policy shown in the DQN algorithm. However, the randomness added to the exploration is inefficient. This is the problem of *exploration vs. exploitation*, where for exploration, the learner attempts to take previously unknown moves to find better optimal policies, whereas exploitation refers to the learner maximizing its rewards within all currently known states and actions. In this section, we explore more advanced exploration methods and other advances that help generalize the learned policies to cover a larger variety of states (or a more "important" set of states) in the environment.

10.4.1 Exploration Methods

We explore three types of main explorations in deep RL:

1. Optimistic exploration refers to the idea that new states are beneficial to learning. This is enforced

by keeping track of visitation frequencies and give bonuses to new states. If we define $N(s)$ as the number of times the agent has visited state s, then we can provide a bonus by augmenting the reward function:

$$r^+(s,a) = r(s,a) + \beta(N(s))$$

where $r(s,a)$ is the "usual" environment reward, but $\beta(N(s))$ augments it by adding a bonus which decreases as $N(s)$ increases. For example, $\beta(N(s)) = \sqrt{\frac{2\ln T}{N(s)}}$. Then $r^+(s,a)$ is actually the reward that the agent uses when running RL. However, counting is tricky because in practice, for high dimensional states, one cannot keep track of a table with all the $N(s)$ values, and it's unlikely that the agent visits states more than once. At a high level, one could solve this by approximating $N(s)$ with function approximation.

2. Thompson-sampling type algorithms that learn distributions of Q-functions and policies in an ensemble of models, which can specialize in exploring certain aspects of the environment.

3. Information gain style algorithms: Curiosity-driven exploration that models the 'predictability' of certain states, and prioritizes those that are currently not predictable by the model as new states that can carry important information which warrant further exploration.

   For example, in model-based RL and curiosity-driven RL[4] one could define a model $\phi(s)$ to map high dimensional states s into a lower dimensional representation, since modeling $p(\phi(s_{i+1})|\phi(s_i), a_i))$ is easier than modeling $p(s_{i+1}|s_i, a_i)$. (This is not just because of lower dimensions, but because in many cases we are not interested in all the precise details of the states but simply the "main idea" or the "essence" of the next state.) When the agent explores in the environment, it automatically generates a dataset $\{s_1, a_1, r_1, s_2, a_2, r_2, \ldots\}$. This can be used as labels to train a *forward dynamics* model which needs to predict $\phi(s_{i+1})$ given $\phi(s_i)$ and a_i, by defining the reward bonus

   $$r_i = \|\hat{\phi}(s_{i+1}) - \phi(s_{i+1})\|_2^2$$

   where $\hat{\phi}(s_{i+1})$ is the estimate of the successor state *in the latent space*. Thus, the higher the error, the higher the reward bonus. It makes sense to add a reward bonus here because if the agent was unable to tell what the next state should be, perhaps it should explore there!

10.4.2 Information Bottleneck

While the information bottleneck method does not directly encourage exploration, it is a new approach that improves robustness and generalization in RL.

For our example, we use a probabilistic model as our neural network. We set a bottleneck Z that accepts both S and G and we use both Z and S to form our policy $\pi(A|S,G)$ as shown in the figure below. This corresponds to an encoder such that Z now carries state-dependent information about the goal $G|S$. And the decoder directly parametrizes the final policy $\pi(A|G,S)$.

[4]Unlike in the model-free setting, model-based reinforcement learning (as the name suggests) implies learning the transition dynamics, then (or in parallel to) figuring out how to choose actions. Contrast this with an algorithm like DQN, which only needs to learn $Q_\theta(s,a)$, and doesn't need to learn a model to define $p(s_{i+1}|s_i, a_i)$.

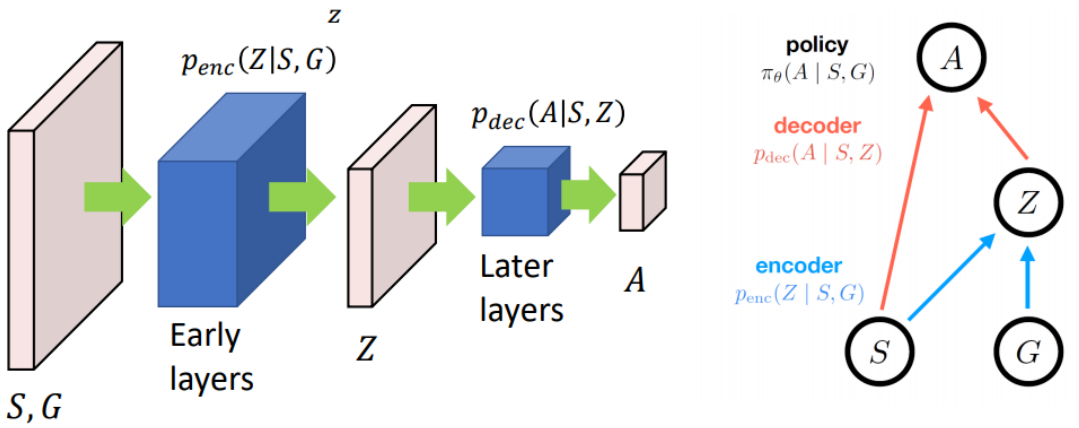

Figure 10.2: Information Bottleneck for Reinforcement Learning

In the information bottleneck strategy, we want to minimize goal specific information in the network. Hence we minimize the KL divergence between $\pi(A|G,S)$ and $\pi(A|S)$ that does not depend on G. Since only Z depends on G, we can instead minimize the KL divergence between $p(Z|S,G)$ and $p(Z|S)$, where

$$p(Z|S) = \sum_g p(g)p(Z|S,g)$$

is the marginalized encoding. In relation to the original information bottleneck, this is equivalent to minimize the mutual information $I(Z;G|S)$ between Z and $G|S$ and maximizing the latter.

While this KL divergence can be computationally prohibitive to optimize because it can be difficult to arrive at even a reasonable approximation of $p(G)$. As such, we can reformulate this problem using adversarial methods. To achieve this, we can train a goal predictor $q(g|s,z)$ (note: the formal definition of information bottleneck requires the approximation of $p(g|s)$, but this is marginalized because it does not depend on z.) Then, using this 'adversary' that attempts to predict the goal from the state and z, we minimize this probability over our model parameters. This encourages the model to encode the less information in z that allows the adversarial goal predictor to successfully predict the goal using z, and is equivalent to our original objective of minimizing the mutual information.