EE675A

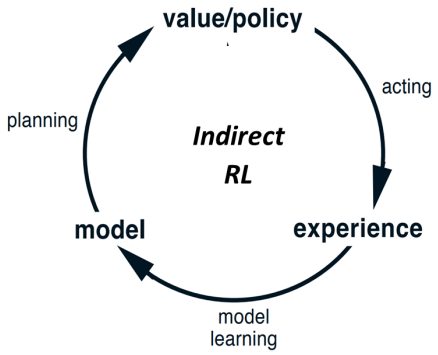
Lecture 20: Dyna, Function approximation using MC, TD and DQN

5/04/2023

Lecturer: Subrahmanya Swamy Peruru *Scribe: Lucky Kant Nayak and Toshith*

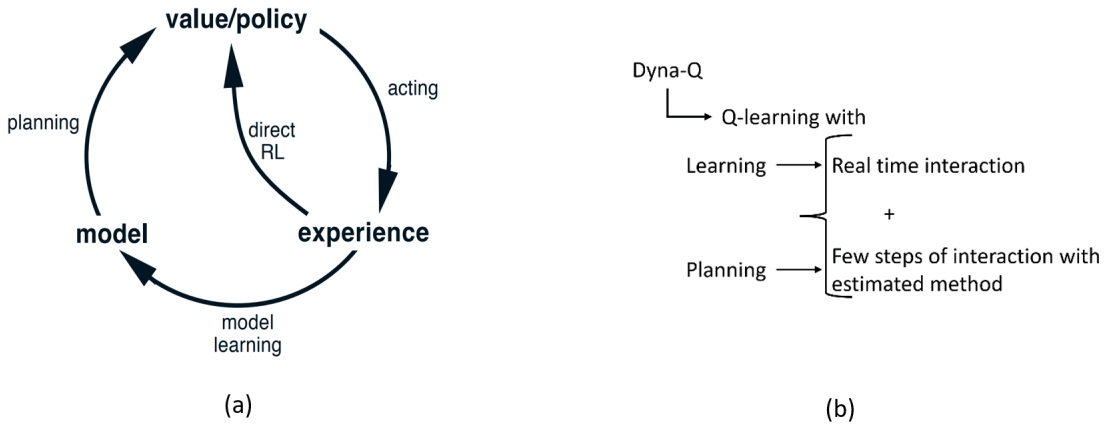
1 Model Based RL

Model-based RL algorithms use an explicit model of the environment to make predictions about the next state and reward. In model-based RL, the agent learns a model of the environment by observing the transitions between states and the corresponding rewards. Once the model is learned (i.e. the estimation of $P^a_{ss'}$ and R^a_s from interaction data) the agent can "simulate" experience by generating trajectories using the model, which can be used to update its value estimates and improve its policy (refer figure below). In contrast, model-free RL directly estimates the value function or policy from experience, without explicitly building a model of the environment.



1.1 Dyna-Q

Dyna-Q is a model-based reinforcement learning algorithm that integrates a model of the environment within the Q-learning framework to improve learning efficiency. Its main motivation is to combine the strengths of model-free and model-based learning approaches to achieve better performance in environments where learning from experience is slow or inefficient. Dyna is an algorithm that combines model-based and model-free learning. It learns a model of the environment using a subset of the observed experience and uses this model to generate simulated experience that is used to update the value function. Dyna-Q combines these two approaches by learning a model of the environment using observed experience and using it to simulate additional experience. Specifically, Dyna-Q uses a data structure called the "model" to represent the environment dynamics, which includes the transition probabilities between states($P^a_{ss'}$)and the expected rewards(R^a_s) associated with each transition. The model is updated as the agent interacts with the environment, by recording the observed state transitions and corresponding rewards.



Dyna-Q Processes

During learning, Dyna-Q uses the model to simulate additional experience, generating "planning" episodes from the current state to update its Q-values. These planning episodes involve selecting actions in the current state, using the learned model to generate a simulated next state and reward, and updating the Q-values using the simulated experience. By using simulated experience, Dyna-Q can learn more efficiently and update its value estimates more frequently than with real experience alone. The planning method is the random-sample one-step tabular Q-planning method.

Algorithm 1 Random-sample one-step tabular Q-planning

Loop forever:

1. Select a state, $s \in S$ and an action, $a \in A(s)$, at random
2. Send s, a to a sample model, and obtain sample next reward, r, and sample next state, s'
3. Apply one-step tabular Q-learning to s, a, r, s' :

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a Q(s',a) - Q(s,a)]$$

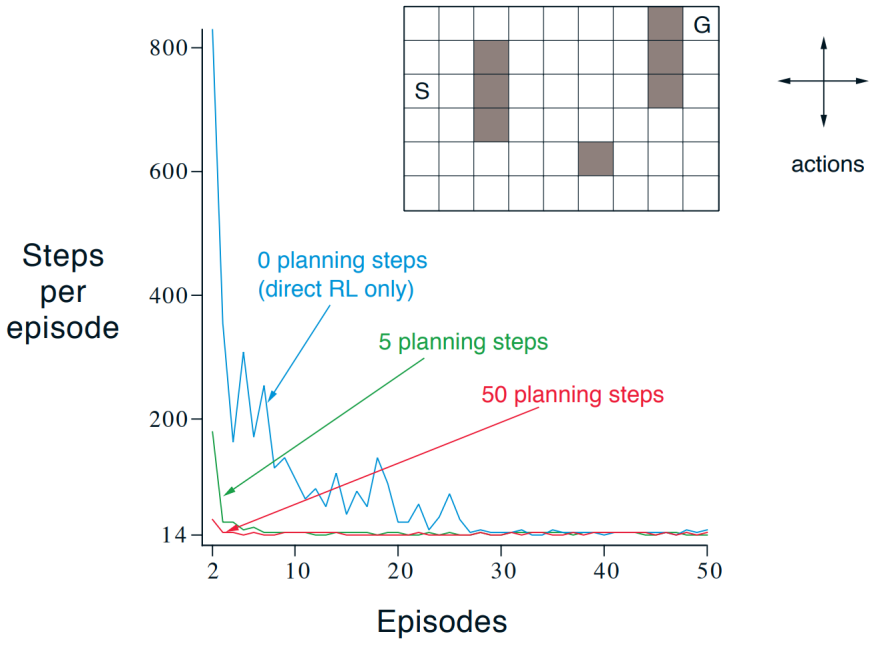
Note how experience can improve value functions and policies either directly or in-directly via the model (refer Dyna-Q processes diagram above). The use of indirect methods in reinforcement learning allows for more efficient utilization of limited experience, resulting in the creation of a better policy through fewer interactions with the environment. However, direct methods are comparatively simpler and not susceptible to model design biases.

In summary, Dyna-Q is a model-based RL algorithm that combines the strengths of model-free and model-based approaches to improve learning efficiency. It uses a learned model of the environment to simulate the additional experience, allowing for more frequent updates of the Q-values and policy.

Algorithm 2 Tabular Dyna-Q

Initialize $Q(s,a)$and$Model(s,a)$ for all $s \in S$ and $a \in A(s)$
Loop forever:

1. $S \leftarrow$ current (nonterminal) state
2. $A \leftarrow \epsilon$-greedy(s,Q)
3. Take action a; observe resultant reward, r, and state, s'
4. $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a Q(s',a) - Q(s,a)]$
5. $Model(s,a) \leftarrow r, s'$ (assuming deterministic environment)
6. Loop repeat n times:
   $S \leftarrow$ random previously observed state
   $A \leftarrow$ random action previously taken in s
   $r, s' \leftarrow Model(s,a)$
   $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a Q(s',a) - Q(s,a)]$



A simple maze (inset) and the average learning curves for Dyna-Q agents varying in their number of planning steps (n) per real step. The task is to travel from **S** to **G** as quickly as possible. For more information refer 'Reinforcement Learning: An Introduction' by Sutton and Barto

2 Function Approximation

All the algorithms that we have seen till now are "Tabular methods" which are based on representing the value function or policy using a table or matrix that stores the value of each state or state-action pair.

Problem: Tabular approach is feasible for small and discrete state spaces, but it becomes impractical for large or continuous state spaces, which are common in many real-world applications. For example, Go game has 10^{170} states.

Solution: Maintain features and use 'Function Approximation'. Function approximation is a technique that seeks to estimate the value function or policy using a parametric function (containing set of features), which can map the input state(s) to the corresponding output value:

$$X(s) = \begin{bmatrix} X_1(s) \\ X_2(s) \\ X_3(s) \end{bmatrix} \mapsto \text{3-dimensional space (state features)}$$

or state-action pair ((s, a))to the corresponding output value:

$$(s, a) \longrightarrow X(s, a) = \begin{bmatrix} X_1(s, a) \\ X_2(s, a) \end{bmatrix} \mapsto \text{((state,action) features)}$$

Function approximation is performed by choosing a function class, such as linear functions, neural networks, or decision trees, and learning the parameters of the function based on a set of observed state-action pairs and corresponding rewards. The learning process typically involves minimizing a loss function that measures the error between the predicted and observed values. Compared to tabular methods, function approximation has the advantage of being able to generalize across similar states, reducing the amount of data required to estimate the value function or policy. However, function approximation can introduce bias and approximation errors, which may degrade the performance of the resulting policy. Therefore, it comes with its own set of challenges, including the choice of function class and the potential for approximation errors.
Example,

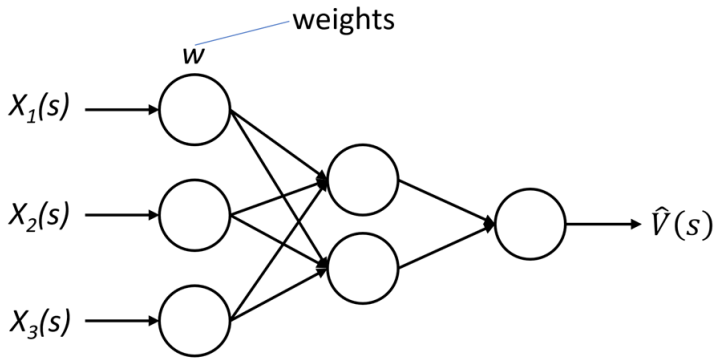
1. Linear Function Approximation:

$$V(s, w) = w_1 X_1(s) + w_2 X_2(s) + w_3 X_3(s)$$

   where, $V(s, w)$is value function estimate and w_i is the weights for different features that approximate the function.
   In linear function approximation, the features used to represent the function are chosen based on domain knowledge or learned automatically from the data. The weight vector is learned using an optimization algorithm such as stochastic gradient descent, which minimizes the mean squared error between the estimated value function or policy and the true value function or policy. Compared to tabular methods, linear function approximation has the advantage of being able to generalize to unseen states or actions, which is not possible with tabular

methods. However, linear function approximation is prone to approximation errors, which can lead to suboptimal performance in some cases. Furthermore, choosing the right features for linear function approximation can be challenging and requires domain knowledge or extensive experimentation.

2. Neural Network Approximation *Input:* the features of state or state-action pair, and *Output:* the estimated value function ($\hat{V}(s)$) or policy.



The network is trained using a set of experience tuples, where each tuple contains the state, action, reward, and next state. The network is trained to minimize the difference between the estimated value function or policy and the actual value function or policy, as measured by a loss function. Neural networks can learn to generalize from a set of experiences, allowing them to estimate the value function or policy for unseen states or actions. Compared to tabular methods and linear function approximation, neural network approximation can be more computationally expensive and may require more experience to train. However, they can also achieve better performance in complex environments with high-dimensional state and action spaces.

Finding best value function approximator

Given: Policy($\pi(s)$, i.e probability of visiting state s while following policy π); True Value Function ($V_\pi(s)$)
Goal: To find the parameters/weights (w^*) such that $\hat{V}_\pi(s, w)$ is the best possible fit

$$w^* = \arg\min_w \sum_s \left(\hat{V}_\pi(s, w) \breve{} V_\pi(s)\right)^2$$

However, we might not be interested in minimising the error for some state for which we rarely encounter with our policy π. We are interested to minimise the error in the states where the policy π spends most of its time. So, instead of giving equal weightage to every possible state in the error we give higher importance to state that are likely to be visited during policy π.

i.e.

$$\overline{VE} = \arg\min_{w} \sum_{s} \left(\hat{V}_{\pi}(s, w) \breve{} V_{\pi}(s) \right)^2$$

where $\overline{VE}$ is the mean square value error, which roughly gives us measure of how much the approximate values differ from the true values.

Stochastic Gradient Descent for value estimation

$$w_{\text{new}} \leftarrow w_{\text{old}} - \alpha \left(\hat{V}_{\pi}(s; w) - V_{\pi}(s) \right) \nabla \hat{V}_{\pi}(s; w) \tag{1}$$

Problem: We don't know the true value function $V_{\pi}(s)$.
Solution: We approximate it using Monte Carlo (MC) or Temporal Difference (TD) methods.
MC function approximation:
True $V_{\pi}(s) \approx G_t$

$$w_{\text{new}} \leftarrow w_{\text{old}} - \alpha \left(\hat{V}_{\pi}(s; w) - G_t \right) \nabla \hat{V}_{\pi}(s; w) \tag{2}$$

TD function approximation:
True $V_{\pi}(s) \approx r + \gamma \hat{V}_{\pi}(s'; w)$

$$w_{\text{new}} \leftarrow w_{\text{old}} - \alpha \left(\hat{V}_{\pi}(s; w) - r + \gamma \hat{V}_{\pi}(s'; w) \right) \nabla \hat{V}_{\pi}(s; w) \tag{3}$$

Note:

- MC function approximation is SGD.
- However, TD function approximation is not true SGD, because we are not taking the derivative w.r.t to $\hat{V}_{\pi}(s'; w)$ although it is a function of w.
- Hence, TD function approximation is called "semi-gradient" method.

3 Deep Q-Network

Deep Q-Learning is a popular algorithm for reinforcement learning, which combines deep neural networks as a function approximate to learn optimal policies in environments with large state and action spaces. However from Deep learning perspective it faces two major issues : 1) In Deep Learning the target is a stationary ground truth value, while in DQN the target is a non-stationary approximate to ground truth value, 2) Deep Learning assumes that the training samples are independent , while in DQN the samples are sequences of highly correlated states, which is alleviated by taking another stationary target network and experience replay mechanism respectively.

Fit Network to optimal values $Q^*(s, a)$:

$$\min_{\theta} \mathbb{E}_{s,a} \left[\left(Q(s, a; \theta) - Q^*(s, a) \right)^2 \right] \tag{4}$$

Update weights using SGD:

$$\theta_{i+1} \leftarrow \theta_i + \alpha \cdot (Q(s, a; \theta_i) - Q^*(s, a)) \cdot \nabla_{\theta_i} Q(s, a; \theta_i) \tag{5}$$

As we don't have optimal $Q^*(s, a)$ values:

$$Q^*(s, a) \approx r + \gamma \max_{a'} Q(s', a'; \theta_i) \tag{6}$$

Issues:

1. The target $r + \gamma \max_{a'} Q(s', a', \theta_i)$ is non-stationary as the θ_i changes with time.
2. If we update in an online way, then after updating (s_t, a_t) pair at the current step, at the next step we are updating (s_{t+1}, a_{t+1}) pair which is highly correlated with the previous pair.

Solutions:

1. Target Network: Keep a separate target network to compute the target $r+\gamma \max_{a'} Q(s', a'; \theta_i)$. Keep the parameters fixed for some c iterations and then update the parameters to the online network parameters.
2. Memory Replay Buffer: Store the last few transitions (s, a, r, s') data generated through the agent-environment interaction. At a given time, sample a random transition data from the buffer and apply SGD to update the weights. This breaks the correlation between updates at time t and $t + 1$.

3.1 Algorithm

The Deep Q-Learning algorithm with a stationary target network can be summarized as follows:

Algorithm 3 Deep Q-Learning

Initialize replay buffer D with capacity N
Initialize Online Q-network with random weights θ
Initialize Target Q-network with the same weights $\theta^- \leftarrow \theta$
Initialize exploration strategy (e.g., ϵ-greedy)
for episode $= 1$ to M **do**
  Initialize state s
  for time step $t = 1$ to T **do**
    Choose action a based on the exploration strategy
    Take action a, observe reward r and new state s'
    Store transition (s, a, r, s') in D
    Sample a mini-batch of transitions from D
    Compute target Q-values y using θ^- for each transition:

$$y_i = r_i + \gamma \max_{a'} Q(s'_i, a'; \theta^-)$$

    The loss function to minimize:

$$\mathcal{L}(\theta) = \frac{1}{B} \sum_{i=1}^{B} (y_i - Q(s_i, a_i; \theta))^2$$

    Update θ using semi-gradient descent:

$$\theta \leftarrow \theta - \alpha \cdot \frac{1}{B} \sum_{i=1}^{B} (y_i - Q(s_i, a_i; \theta)) \cdot \nabla_\theta Q(s_i, a_i; \theta)$$

    Every c iterations, update θ^- with the current online weights θ :

$$\theta^- \leftarrow \theta$$

    Update state $s \leftarrow s'$
  end for
end for

Here, D is the replay buffer that stores the transitions experienced by the agent, Q is the Q-network that approximates the Q-values, and Q_t is the target network that is used to compute the target Q-values. The agent chooses actions based on an exploration strategy, such as ϵ-greedy, and updates the Q-network by minimizing the loss function using a mini-batch of transitions from the replay buffer. The target network is updated every c iterations to stabilize the training process and prevent overfitting.

The material is primarily based on the content from [1] and class Lectures

References

[1] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.