Stabilizing Q-learning with Linear Architectures for Provably Efficient Learning

S.Huang

The Q-learning algorithm is simple and widely used a stochastic approximation scheme for reinforcement learning, but the basic protocol can exhibit instability in conjunction with function approximation. Such instability can be observed even with linear function approximation. In practice, tools such as target networks and experience replay appear to be essential, but the individual contribution of each of these mechanisms is not well understood theoretically. This work proposes an exploration variant of the basic Q-learning protocol with linear function approximation. Our the modular analysis illustrates the role played by each algorithmic tool that we adopt: a second order update rule, a set of target networks, and a mechanism akin to experience replay. Together, they enable state-of-the-art regret bounds on linear MDPs while preserving the most prominent feature of the algorithm, namely a space complexity independent of the number of steps that elapsed. We show that the performance of the algorithm degrades very gracefully under a novel and more permissive the notion of approximation error. The algorithm also exhibits a form of instance dependence, in that, its performance depends on the "effective" feature dimension.

1 Introduction

Q-learning is an algorithm that is a model-free, off-policy, and value-based algorithm under reinforcement learning. A general Q-learning algorithm will be like:

$$\underbrace{NewQ(s,a)}_{\text{New q value for that state and that action}} = \underbrace{Q(s,a)}_{\text{current q value}} + \alpha \left[\underbrace{R(s,a)}_{\text{reward}} + \gamma max \underbrace{Q'(s',a')}_{\text{future reward}} - Q(s,a) \right]$$

Here, α is the learning rate and γ is the decay rate

Affiliation Independent
Correspondence skylar.huang@tum.de
Draft June 11, 2024

As we learned from this paper, this model was inspired by the deep RL with the concept policy replay mechanism since it stabilizes the learning process by eliminating the distribution shift problem.
THere are three main ingredients that are key in our analysis.

- a second-order update rule for improved statistical efficiency;
- a set of target networks (Mnih et al., 2015) to stabilize the updates.
- a replay mechanism called policy replay.This mechanism is similar to experience replay used in the deep RL literature.

While the second-order scheme and the target networks have been used in the optimization and the RL literature before, the policy replay mechanism is one key reinforcement learning contribution made in this paper.

2 Dataset

In this paper, we focus on the finite-horizon Markov decision processor. Thus, we take Grid world as our input dataset.

There are 5 steps I need to complete in my code:

- Defining the Grid World environment: First you need to define the Grid World environment, including the grid size, start position, end position, and all possible actions.
- Defining reward functions: In the Grid World environment you need to define reward functions. The reward function can determine the size of the reward depending on the current state and action. We can use the boud function that is defined in the paper.
- Initialize parameters and data structures: Depending on the characteristics of the Grid World environment, you need to initialize the parameters and data structures in the algorithm.
- Implementing the S4Q-LEARNING algorithm: According to the pseudocode of the algorithm, you need to implement the specific steps of the S4Q-LEARNING algorithm. This includes updating the Q-value, calculating the reference covariance matrix, and selecting the optimal action.
- Loop iteration: Loop iteration is performed using the Grid World environment and the S4Q-LEARNING algorithm according to the provided algorithm

pseudo-code. In each iteration step, the starting state is sampled, the action is selected, the covariance matrix is updated, the trigger value is calculated, etc. When the trigger value satisfies the stop condition, the iterative loop is exited.

3 Code part

This part is a kinda code record and analysis: implement from pseudo code:

```
            import numpy as np

# Parameters
H = 3  # Number of policies
c = 0.1  # Parameter for S3Q-LEARNING
K = 10  # Number of trajectories to simulate
   = 0.1  # Update trigger parameter

# Initialize policy replay memory
policy_replay_memory = []

# Define bonus function b
def bonus_function(s, a):
    # Define your bonus function here
    return 0  # Placeholder bonus function

# Main loop for each phase p
for p in range(1, K + 1):
    mtot = np.sum([mj for pij, mj in policy_replay_memory[:p-1]])
# Calculate mtot

    # Define  Control  function
    def pi_control():
        j = np.random.choice(range(p - 1), p=[mj / mtot for pij, mj in policy_r
        return policy_replay_memory[j][0]  # Return selected policy

    # Apply S3Q-LEARNING
    def s3q_learning(pi_control, c, mtot, b):
        # Implementation of S3Q-LEARNING algorithm
        Q = np.zeros((H, num_actions))  # Q-values initialization
```

```
        Sigma_ref = np.eye(num_features)  # Reference covariance matrix initial

        # Implement S3Q-LEARNING algorithm here and update Q and Sigma_ref

        return Q, Sigma_ref

    Q, Sigma_ref = s3q_learning(pi_control, c, mtot, bonus_function)
# Obtain optimistic Q values

    # Extract greedy policy
    policy = np.argmax(Q, axis=1)

    # Initialize trajectory count and trigger value
    m = 0
    Th = 0

    # Repetition loop
    while Th < (c * np.log(p * K /   )):
        s = sample_start_state()  # Sample start state

        for h in range(H):
            a = policy[h, s]  # Choose action based on policy
            m += 1  # Increment trajectory count

            # Update trigger value
            Th += np.linalg.norm(feature_vector(s, a)) ** 2 / Sigma_ref[h, h]

            # Update accumulator and covariance
            Sigma_h += np.outer(feature_vector(s, a), feature_vector(s, a))

            s = transition(s, a)  # Transition to the next state

        # End of repetition loop

    policy_replay_memory.append((policy, m))  # Update policy replay memory

# End of main loop for each phase
```

- sample start state() is used to sample the starting state, transition(s, a) is used for state transfer, and feature vector(s, a) is used to extract the feature vector of the state-action pair.
- Set the grid size, action space size, the dimensionality of the feature vector, and number of iterations.

4 Problem

- What is the difference between iteration and grid world?
- How to implement the replay policy