Tabular Methods

July 13, 2018

1 Introduction to Reinforcement Learning: Tabular Methods

The goal of reinforcement learning is to achieve goal-directed learning from interactions with an environment. At each time step, t, the agent recieves a state observation S_t and a reward R_t, and performns an action A_t, like so:

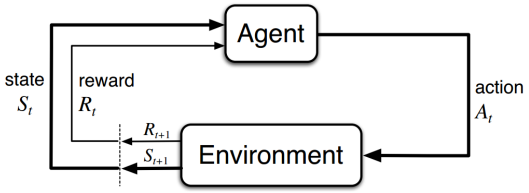


Figure 3.3. from Sutton Barto [1]

The agent must learn to pick actions that maximize the total expected future reward, called the **return**, $R_{t+1} + R_{t+2} + R_{t+3} + ...R_T$, where T is the timestep at which the episode terminates. In practice we often use the discounted return instead of the actual return. **Discounted returns** weights imminent rewards higher than rewards in the far future. This is controlled by the discounting factor $\gamma \in [0, 1])$, like so:

$$G_t = \sum_{k=0}^{T} \gamma^k R_{t+k+1}$$

For simple cases, where the actions and observation space are small and discrete it is possible to use **tabular approaches**, where each possible state-action pair is enumerated (in a table). Tabular approaches aren't applicable to many real world problems, but they can be useful for illuminating the fundamental principles of reinforcement learning.

This notebook describes tabular versions of two of the classical reinforcement learning algorithms: SARSA and Q-learning.

2 The Algorithms

Most reinforcement learning mehtods involve estimating a **value function** — a function that estimate how good a given state or state-action pair is in terms of the expected return. The return of a state naturally depends on the agents behavior in the future, value functions are therefore defined with respect to a policy. So given the state-action pair, (s, a), the **state-action value function** for following policy π is denoted as

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$$

Similarly the **state value function** is

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$

In the tabular case the value function is represented as a table, with one entry for each unique state-action pair. The subsript is often omitted when it clear which policy is used. Value functions are nice to work with, as they automatically take into account what will happen in the future, thus greedily optimizing the value function is equivalent to maximizing the longterm expected reward. Both Q-learning and SARSA estimate the state-action value function.

In this notebook, as is commonly done, the **ϵ-greedy** policy is used. The policy selects the action with the highest value: $\pi(s) = \max_a Q(s, a)$ with probability $1 - \epsilon$, and a random action with probability ϵ. Using a stochastic policy is desirable as it ensures that every state has a non-zero chance of being selected. This is necessary for the convergence proof (see Sutton and Barto [1] for details). ϵ-greedy is one of the simplest (and quite naive) solutions to the **exploration-exploitation dillemma**, but it works well for small to medium problems (e.g. it is sufficent for many, but not all Atari games).

2.1 Temporal Difference Methods

One of the most important ideas witihin reinforcement learning is that of **temporal difference** (TD) methods. TD methods combine elements of dynamic programming and Monte Carlo methods (see Sutton and Barto [1] chapters 4 and 5), resulting in a methods that can solve environments with unknown dynamics in an online manner. TD methods work by minimizing the **TD-error**, δ_t, a measure of the difference between the current estimate and a better estimate. The exact formulation of δ_t depends on the algorithm. Below we will see two such formulations. When using the state-action value function the updates are simply:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\delta_t$$

where α is the step-size parameter. Note that we use q to denot the true value function, and Q to denote the approximate value function, similar to Sutton and Barto [1].

Below we will look at how the TD methods SARSA and Q-learning defines the TD-error, and the consequences of this.

2.2 SARSA

SARSA **on-policy** method, meaning that the value function estimates the policy that the agent follows. The TD-error for SARSA is given by:

$$\delta_t = \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

For terminal state S_{t+1} the value function $Q(S_{t+1}, A_{t+1})$ is defined as zero.

$R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ is an approximation of the return, G_t, and is called the **target** of the update. Note that the value function appears in the target. I.e. the update is performed using **bootstrapping**, because the update of the value function is done inpart based on current estimates of the value function. Using bootstrapping generally speeds up training time and reduces memory requirements, but it also adds some instability to the system. This algorithm is called **1-step SARSA**, the target consists of one observed reward, R_{t+1}.

2.3 Q-learning

Q-learrning is an **off-policy** method, meaning that the agent is able to follow a different policy, the **behavior policy**, than the one that it learns to estimate, the **target policy**. For Q-learning the TD-error is:

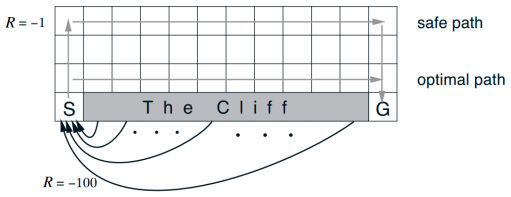
$$\delta_t = \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

The difference between SARSA and Q-learning lies in the difference in the TD-error, δ_t, and how it affects the updates. SARSA updates current estimates using observed state-action pairs, where as the Q-learning update uses hypothetical 'best guesses' updates (as indicated by the max). This is an important difference as the SARSA agent learns to optimize the policy it follows, and Q-learning learns the optimal policy, even though it doesn't follow it.

In the following we will compare the two algorithms using the classical **Cliff Walking** example, that demonstrates the difference between on-policy (SARSA) and off-policy (Q-learaning) methods.

3 Cliff Walking

The Cliff Walking environment (Sutton and Barto [1], Example 6.6) is a simple 4×12 grid. The agent starts in one corner of the gird, and must move to another. At each step the agent chooses a direction to go, up, right, down, or left, and moves one step in that direction. In all states the agent recieves a reward of -1, except *the cliff*, which gives the agent a reward of -100, and sends the agent back to start, and the *terminal state* which terminates the episode, and gives a reward of 0. The agent maximizes the reward by getting to the terminal state as quickly as possible.



Example 6.6 from Sutton Barto [1]

```
In [1]: ## Useful Jupyter setup commands
        %load_ext autoreload
        %autoreload 2
        %matplotlib inline

In [2]: from __future__ import absolute_import
        from __future__ import division
        from __future__ import print_function

        import numpy as np
        import matplotlib.pyplot as plt
        from IPython.display import clear_output
```

```
import utils
from cliff import Cliff
from agents import TabularNStepSARSA, TabularNStepQLearning
```

4 Experiments

Below we train tabular 1-step SARSA and 1-step Q-learning agents on the Cliff Walking environment. In this notebook we will use an ϵ-greedy policy, with $\epsilon = 0.1$ held constant, unless noted otherwise.

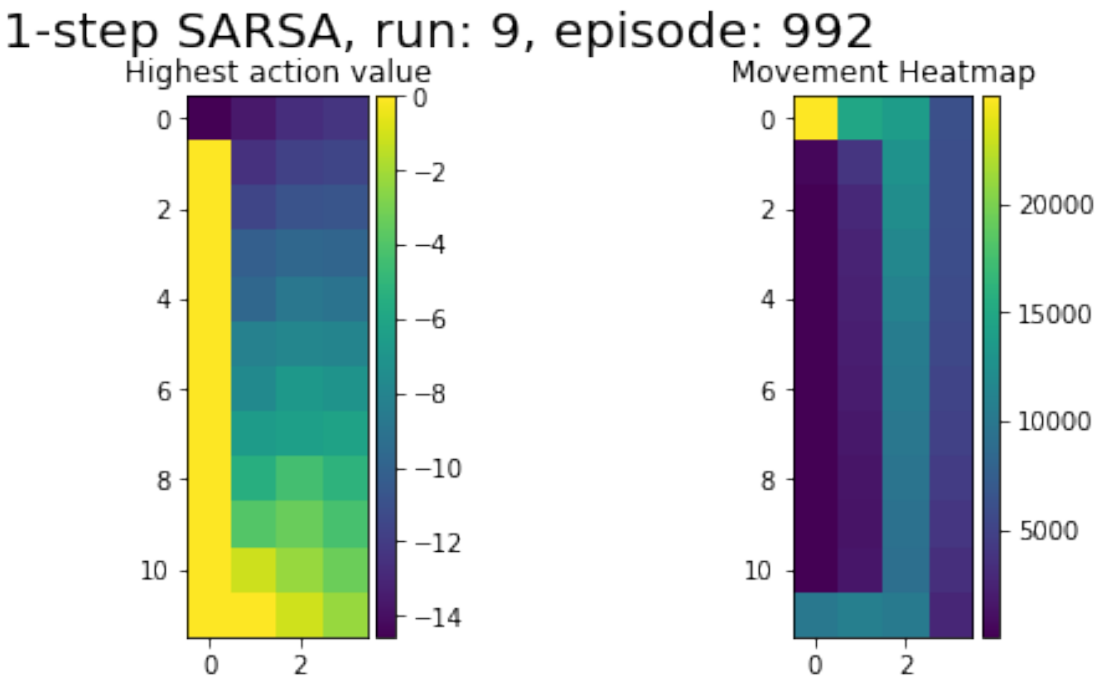
During training we monitor

- the highest action value for each possible state (**left plot**), i.e. value of the greedy action of the agent. (Note the cliff has value 0. This is the initial value, and since the agent never actually performs any actions here it is never changed.)
- the movement as heatmap (**right plot**) i.e. the number of times the agent has visited each state.

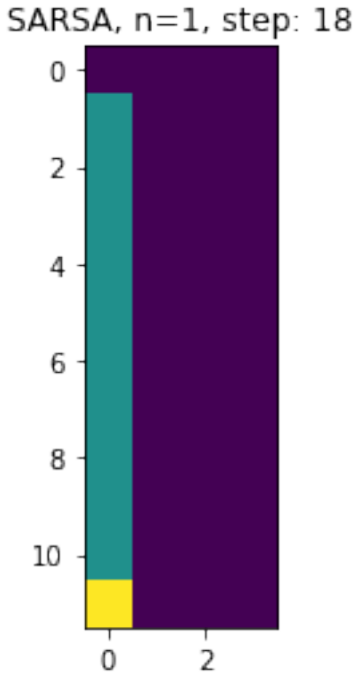
In [3]:
```
## Run settings
num_runs = 10  # Number of runs to average rewards over
eps_per_run = 1000  # Number of episodes (terminations) per run
n = 1  # n parameter in n-step Bootstrapping
```

In [4]:
```
## SARSA
TN_SARSA_rewards = []
env = Cliff()
for i in range(num_runs):
    TN_SARSA = TabularNStepSARSA(env.state_shape, env.num_actions, n=n)
    _, rewards = utils.run_loop(env, TN_SARSA, str(n)+'-step SARSA, run: ' + str(i), ma
    TN_SARSA_rewards.append(rewards)

TN_SARSA_rewards = np.array(TN_SARSA_rewards)
```



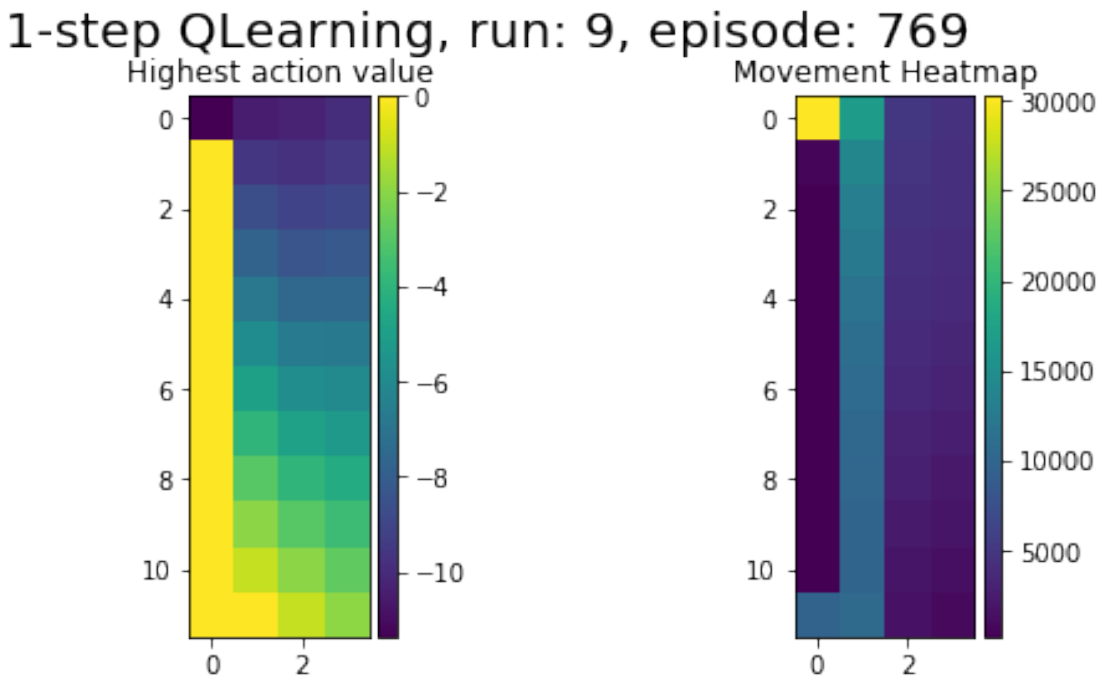
In [5]:
```
# Run the last SARSA agent using visualizations.
# Try running this a couple of times
utils.run_loop(env, TN_SARSA, 'SARSA, n='+str(n), max_e=1, render=True)
```



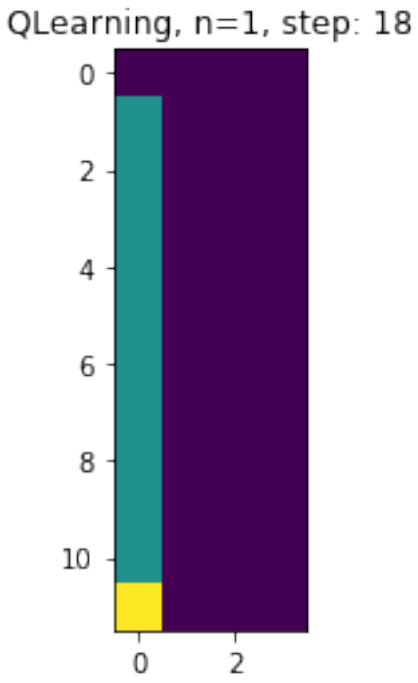
Out[5]: ([18], [-17])

In [6]:
```
## Q-learning
TN_QLearning_rewards = []
env = Cliff()
for i in range(num_runs):
    TN_QLearning = TabularNStepQLearning(env.state_shape, env.num_actions, n=n)
    _, rewards = utils.run_loop(env, TN_QLearning,  str(n)+'-step QLearning, run: ' + s
    TN_QLearning_rewards.append(rewards)

TN_QLearning_rewards = np.array(TN_QLearning_rewards)
```



In [7]:
```
# Run the last QLearning agent using visualizations.
# Try running this a couple of times
utils.run_loop(env, TN_QLearning, 'QLearning, n='+str(n), max_e=1, render=True)
```



Out[7]: ([18], [-116])

5 Results and Discussion

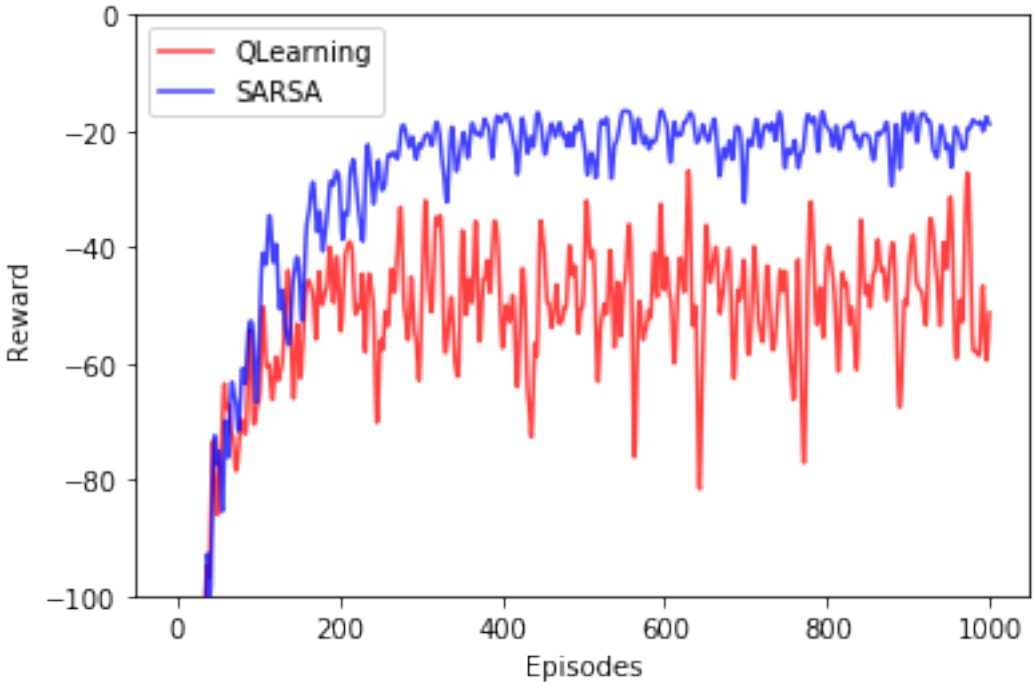
Looking at the *Movement Heatmaps* we see that the Q-learning agent learns the 'optimal path' along the cliff, where as the SARSA agent learns a safer path, further from the cliff.

The code cell below plots the (smoothed) average reward for Q-learning and SARSA as a function of episodes. The SARSA agent achieves a higher reward during training, despite the Q-learning agent learns the optimal path. This is with probability ϵ the agent moves in a random direction, and might fall off the cliff. The SARSA agent optimizes the behavior policy, and takes this into account. The Q-learning agent on the other hand learns the optimal policy given the presented dynamics of the environemnt without accounting for the behavior policy that it is currently following.

```
In [8]: plt.figure()
        include_sd = False # include standard deviation in plot
        utils.reward_plotter(TN_QLearning_rewards, 'QLearning', 'r', include_sd=include_sd, sm
        utils.reward_plotter(TN_SARSA_rewards, 'SARSA', 'b', include_sd=include_sd, smooth_fac

        axes = plt.gca()
        axes.set_ylim([-100, 0])

        plt.show()
```



However if we change the epsilon to zero and test the agents again we see the benifit of the Q-learning agent. Since both the environment and the agents are deterministic in this case we only have to run the environment once in order to determine their performance.

```
In [9]: TN_QLearning.min_eps = 0
        TN_SARSA.min_eps = 0

        _, TN_QLearning_rewards_no_eps = utils.run_loop(env, TN_QLearning, 'QLearning, n='+str
        _, TN_SARSA_rewards_no_eps = utils.run_loop(env, TN_SARSA, 'SARSA, n='+str(n), max_e=1
        clear_output()

        print("Q-learning rewards with epsilon = 0:", TN_QLearning_rewards_no_eps[0])
        print("SARSA rewards with epsilon = 0:     ", TN_SARSA_rewards_no_eps[0])
```

```
Q-learning rewards with epsilon = 0: -12
SARSA rewards with epsilon = 0:      -14
```

Without randomness the Q-learning agent will consitently beat the SARSA agent. On-policy and off-policy methods have different advantages and dissadvantages, and which one is better will depend on the problem at hand.

If ϵ was slowly annealed to 0 both agents would learn to follow the optimal path.

6 N-step bootstrapping

The agents above use 1-step boot strapping. That is to say that the target is computed based on 1 observed reward. The target (estimate of the return) can be generalized to n-steps like so:

$$G_{t:t+n} = \sum_{k=0}^{n} \gamma^k R_{t+k+1} + \gamma^n Q(S_{t+n}, A_{t+n}) \quad 0 \leq t \leq T - n$$

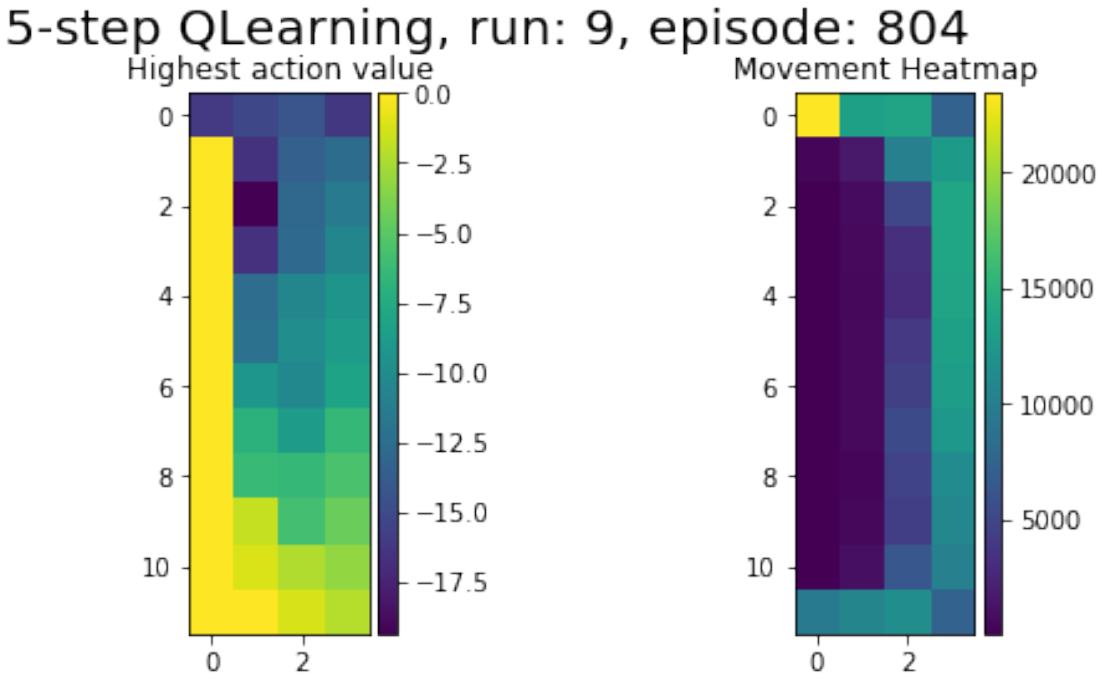
This target can be replaced with little effort with previous target, resulting in n-step versions of SARSA. For $t > T - n$ the actual returns are used (i.e. without bootstrapping).

n-step methods generally train faster than 1-step methods, as the reward signal can be propagated faster through the network. n-step versions of SARSA and Q-learning are demonstrated below. n-step Q-learning is however not a fully off-policy method any more, as the target uses n rewards from the behavior policy.

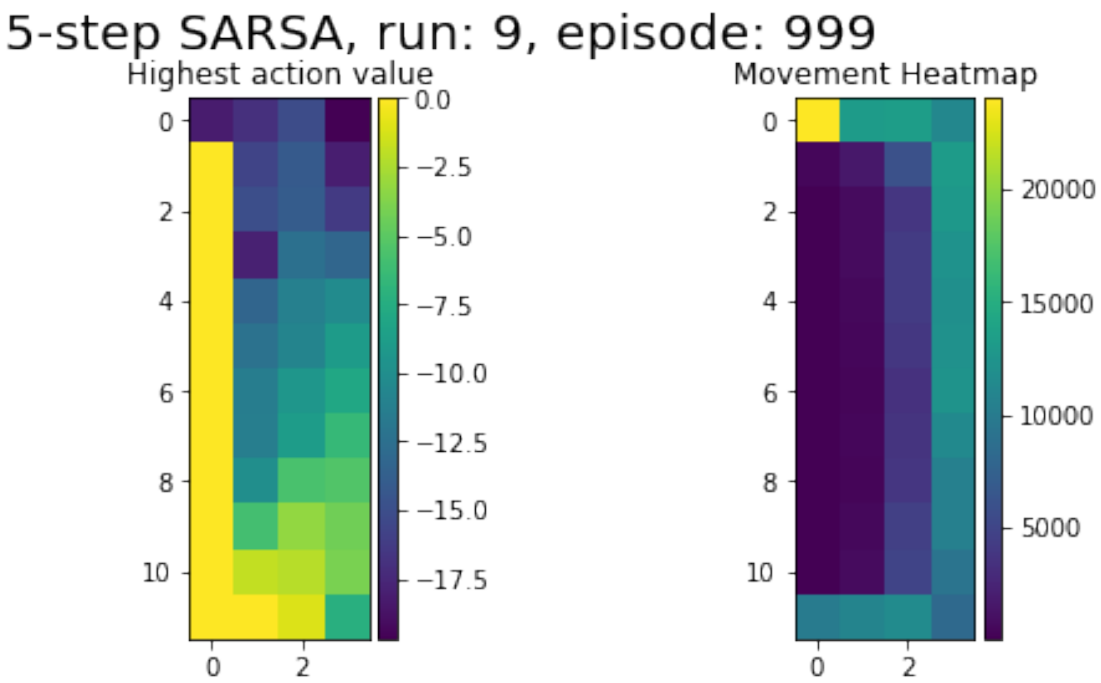
In order to demonstrate this the experiment is re-run below, but using $n = 5$.

In [10]:
```
## Run settings
n = 5
# We leave the other settings as before
```

In [11]:
```
TN_QLearning_rewards = []
env = Cliff()
for i in range(num_runs):
    TN_QLearning = TabularNStepQLearning(env.state_shape, env.num_actions, n=n)
    _, rewards = utils.run_loop(env, TN_QLearning,  str(n)+'-step QLearning, run: ' +
    TN_QLearning_rewards.append(rewards)
TN_QLearning_rewards = np.array(TN_QLearning_rewards)
```



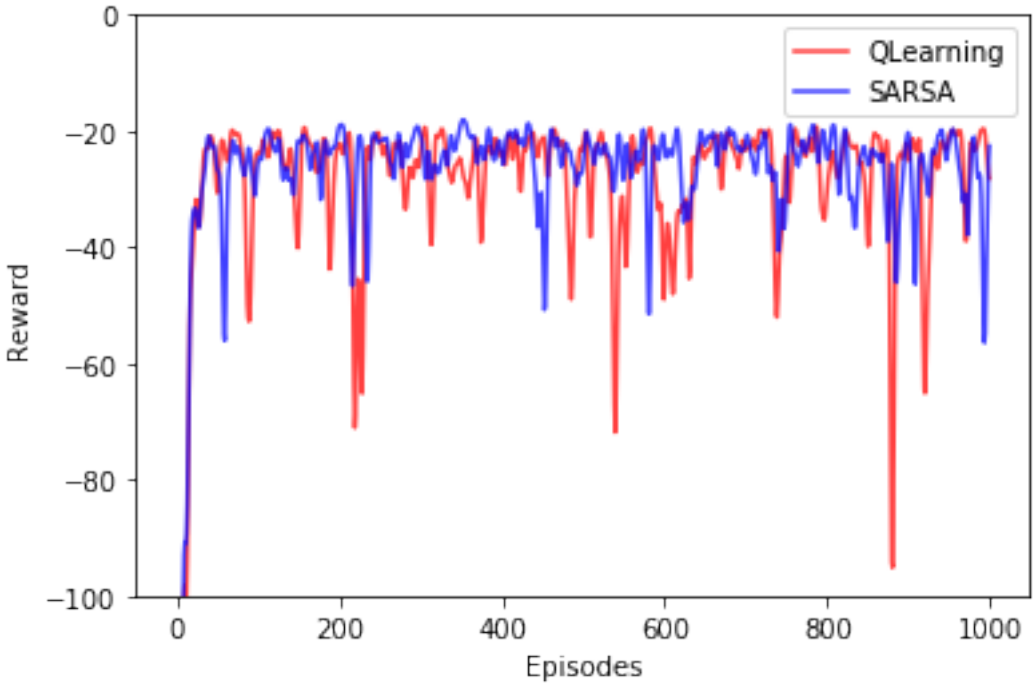
In [12]:
```
TN_SARSA_rewards = []
env = Cliff()
for i in range(num_runs):
    TN_SARSA = TabularNStepSARSA(env.state_shape, env.num_actions, n=n)
    _, rewards = utils.run_loop(env, TN_SARSA,  str(n)+'-step SARSA, run: ' + str(i),
    TN_SARSA_rewards.append(rewards)
TN_SARSA_rewards = np.array(TN_SARSA_rewards)
```



In [13]:
```
plt.figure()
include_sd = False # include standard deviation in plot
utils.reward_plotter(TN_QLearning_rewards, 'QLearning', 'r', include_sd=include_sd, sm
utils.reward_plotter(TN_SARSA_rewards, 'SARSA', 'b', include_sd=include_sd, smooth_fa

axes = plt.gca()
axes.set_ylim([-100, 0])

plt.show()
```



We see that using $n = 5$ makes both networks learn the safe route, and their training performance is indistinguishable, as both of them take the behavior policy into account.

7 Bibliographic Notes

[1] Richard S. Sutton and Andrew G. Barto. 1998. Introduction to Reinforcement Learning (1st ed.). MIT Press, Cambridge, MA, USA.