Solve Hill Climbing Problem Using Off-policy and On-policy Model-free Planning Algorithm

Jingpei Lu JIL360@UCSD.EDU
Department of Electrical & Computer Engineering
University of California, San Diego

1 Problem Formulation

1.1 Problem Setup

The Hill climbing problem is defined as follow: an underpowered car is placed at the bottom of a convex landscape, and it must drive up a hill. However, since the car does not have enough power to reach the goal position directly, it must oscillate backwards and forwards in order to accumulate sufficient potential energy. For any given state (position and velocity) of the car, the agent is given the possibility of driving left, driving right, or not using the engine at all. The agent receives a negative reward at every time step when the goal is not reached; the agent has no information about the goal until an initial success. Initially, the car does not know how to reach the goal, and our goal is to let the agent learns how to reach the final state using model-free algorithm.

One challenge of this problem is that it is represented by the continuous state space. As common reinforcement learning techniques such as SARSA or Q-Learning require a discrete state space. Therefore, the continuous problem must be approximated as a discrete problem such that we have finite number of states in the state space X.

1.2 Markov Decision Process (MDP) Formulation

After discretizing the continuous states, positions and velocity, into sets of discrete state P and V, we can formulate this problem as a finite-state MDP as:

- The state space is $x_t \in X \coloneqq \{P \times V\}$, which is the cartesian product between discretized position and velocity. And $x_t = (p_t, v_t) \in \Re^2$, where $p_t \in P$ and $v_t \in V$.
- The control space is $u_t \in \mathcal{U} \coloneqq \{0,1,2\}$, where 0 means driving left, 2 means driving right, and 1 means not using engine.
- The initial state x_0 can be defined randomly or it can be defined as $x_0 = (-0.5, 0)$.
- The terminal state is $x_\tau \in \mathcal{T} \coloneqq \{(\, p_t,\ v_t) |\, p_t \geq 0.6,\ p_t \in P, v_t \in V\}$.
- The planning horizon is T = 10000.
- The motion model $p_f(x_{t+1}|x_t, u_t)$ is unknown, so we want to use model-free algorithm.
- The stage cost (negative of the reward) is $l(x_t, u_t) = 1$ for $x_t \in X \backslash \mathcal{T}$.
- The terminal cost is $q(x_\tau) = 0$.

1.3 Model-free Optimal Control problem

Since in our problem, the motion model is unknow, we want to use the action-value function $Q^\pi(x, u)$ to represent the long-term reward of taking action u in state x and following policy π afterward. Our goal is to find the optimal policy that minimizes the long-term cost for every state, such as:

$$\pi^*(x) = arg \min_{u \in \mathcal{U}(x)} Q^*(x, u) \quad \forall x \in X \tag{1}$$

where, Q^* is the optimal Q function obtained by minimizing the Q-value:

$$Q^*(x, u) \coloneqq \min_\pi Q^\pi(x, u) \tag{2}$$

2 Technical Approach

2.1 State Space Discretization

Inspired by [1], I approximate the state space using the discretization technique: the state space is subdivided into a set of buckets (a regular grid), and the agent is assigned to the bucket associated with its current state, that is, the nearest one.

For my implementation, I discretize the continuous state space to a 40 by 40 grid, so there are 40 discrete states along each axis. For example, $x_c = (p_c, v_c) \in \mathcal{R}^2$ is a state in continuous state space, and $p_c \in [p_{max,c}, p_{min,c}]$, $v_c \in [v_{max,c}, v_{min,c}]$. We can assign x_c to the nearest state $x_d = (p_d, v_d)$ in the discrete state space, where

$$p_d = int((p_c - p_{\min,c})/stride_p)$$
$$v_d = int((v_c - v_{\min,c})/stride_v)$$
$$stride_p = (p_{max,c} - p_{min,c})/40$$
$$stride_v = (v_{max,c} - v_{min,c})/40$$

2.2 Techniques to ensure convergence

To ensure the convergence to the optimal action-value function Q^*, we adapt the ϵ-Greedy Exploration method to generate the action, so that we do not commit to the wrong controls too early and continue exploring the state and control spaces. The ϵ-Greedy policy is defined as:

$$\pi(u|x) = \mathcal{P}(u_t = u|x_t = x) := \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{U}(x)|}, & u = arg\min_{a \in \mathcal{U}(x)} Q(x,a) \\ \frac{\epsilon}{|\mathcal{U}(x)|}, & otherwise \end{cases} \quad (3)$$

where, $|\mathcal{U}(x)|$ is the size of the control space and ϵ is chosen so that the policy is greedy in the limit with infinite exploitation (GLIE):

- All state-control pairs are explored infinitely many times: $\lim_{k\to\infty} N(x,u) = \infty$
- The ϵ-greedy policy converges to a greedy policy:

$$\lim_{k\to\infty} \pi_k(u|x) = 1\{u = arg\min_{a \in \mathcal{U}(x)} Q(x,a)\}$$

Further, we also adapt the learning rate decay method used in [1] to ensure the convergence by satisfying the Robbins-Monro conditions. The learning rate α is adjusted by the episode number as

$$\alpha = \max(0.003, 1 * 0.85^{\#episode|100}) \quad (4)$$

2.3 Implementing an On-policy TD Policy Iteration Algorithm (SARSA)

State–action–reward–state–action (SARSA) is an algorithm for learning a Markov decision process policy and it updates the policy based on actions taken. Specifically, the updating of the Q-value depends on the current state of the agent, the action the agent chooses, the reward the agent gets for choosing this action, the state that the agent enters after taking that action, and finally the next action the agent choose in its new state. The updating rule can be written as:

$$Q^\pi(x_t, u_t) \leftarrow Q^\pi(x_t, u_t) + \alpha[\, l(x_t, u_t) + \gamma Q^\pi(x_{t+1}, u_{t+1}) - Q^\pi(x_t, u_t)] \quad (5)$$

where, α is the learning rate, γ is the discounted factor, x_{t+1} is the observation from the environment by taking action u_t at state x_t, and u_{t+1} is generated by $\pi(x_{t+1})$.

To ensure the convergence to the optimal action-value function Q^*, we also adapt the ϵ-Greedy Exploration method as described in equation (3) and set the learning rate as described in (4) to satisfy the Robbins-Monro conditions.

For my implementation the discounted factor is 1, the number of episodes is 3000, the ϵ is 1 over the number of the episodes. My implement of the SARSA algorithm is described as the pseudocode below.

Algorithm 1: SARSA Algorithm

1: Algorithm parameters: $\alpha \in (0,1]$, $\epsilon > 0$, $\gamma \in (0,1]$
2: Initialize $Q^{\pi}(x,u) = 0$, for $\forall x \in X, \forall u \in \mathcal{U}(x)$
3: Loop for each episode:
4: Initialize x_0
5: Choose u_0 from $\pi(u|x_0)$ using ϵ-Greedy policy
6: For $t = 0,1,2,\dots,T$ and the terminal state is not reached, do:
7: Take action u_t, observe $l(x_t,u_t)$ and x_{t+1}
8: Choose u_{t+1} from $\pi(u|x_{t+1})$ using ϵ-Greedy policy
9: $Q^{\pi}(x_t,u_t) \leftarrow Q^{\pi}(x_t,u_t) + \alpha[\, l(x_t,u_t) + \gamma Q^{\pi}(x_{t+1},u_{t+1}) - Q^{\pi}(x_t,u_t)]$

Figure 1: The SARSA Algorithm

After obtaining the $Q^{*}(x,u)$ using SARSA, we can extract the optimal policy using the equation (1).

2.4 Implementing an Off-policy TD Policy Iteration Algorithm (Q-Learning)

Q-learning is also a model-free reinforcement learning algorithm for learning a Markov decision process policy. Unlike the SARSA, which learns the Q values associated with taking the policy it follows itself, Q-learning updates an estimate of the optimal state-action value function Q^{*} based on the maximum reward of available actions. The updating rule can be expressed as:

$$Q^{\pi}(x_t,u_t) \leftarrow Q^{\pi}(x_t,u_t) + \alpha[\, r(x_t,u_t) + \gamma \min_{u \in \mathcal{U}(x_{t+1})} Q^{\pi}(x_{t+1},u) - Q^{\pi}(x_t,u_t)] \qquad (6)$$

where, α is the learning rate, γ is the discounted factor, x_{t+1} is the observation from the environment by taking action u_t at state x_t.

To ensure the convergence to the optimal action-value function Q^{*}, we also adapt the ϵ-Greedy Exploration method as described in equation (3) and set the learning rate as described in (4) to satisfy the Robbins-Monro conditions.

For my implementation the discounted factor is 1, the number of episodes is 3000, the ϵ is 1 over the number of the episodes. My implement of the Q-Learning algorithm is described as the pseudocode below.

Algorithm 2: Q-Learning Algorithm

1: Algorithm parameters: $\alpha \in (0,1]$, $\epsilon > 0$, $\gamma \in (0,1]$
2: Initialize $Q^{\pi}(x,u) = 0$, for $\forall x \in X, \forall u \in \mathcal{U}(x)$
3: Loop for each episode:
4: Initialize x_0
5: For $t = 0,1,2,\dots,T$ and the terminal state is not reached, do:
6: Choose u_t from $\pi(u|x_t)$ using ϵ-Greedy policy
7: Take action u_t, observe $l(x_t,u_t)$ and x_{t+1}
8: $Q^{\pi}(x_t,u_t) \leftarrow Q^{\pi}(x_t,u_t) + \alpha[\, l(x_t,u_t) + \gamma \max_{u \in \mathcal{U}(x_{t+1})} Q^{\pi}(x_{t+1},u) - Q^{\pi}(x_t,u_t)]$

Figure 2: The Q-Learning Algorithm

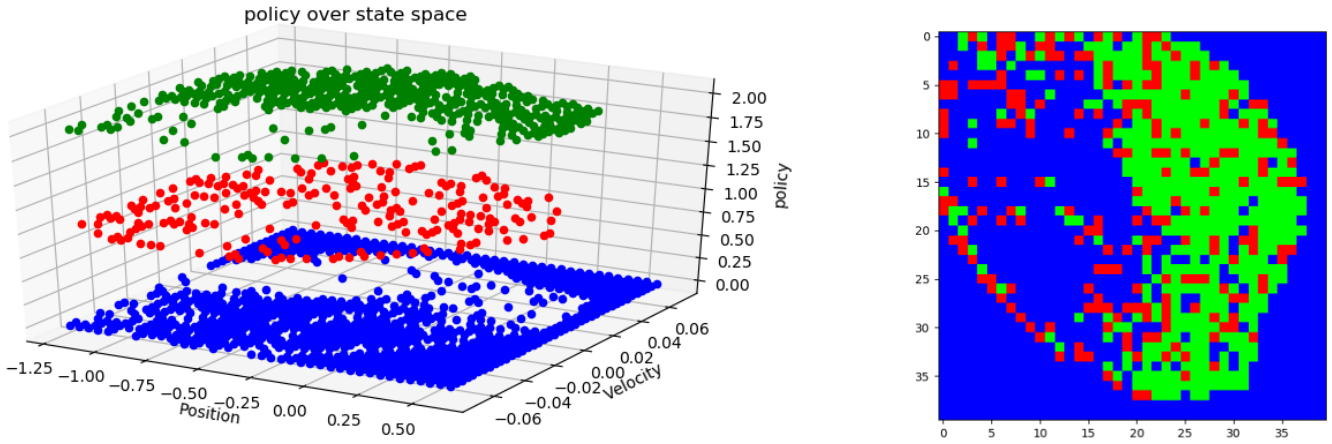
After obtaining the $Q^{*}(x,u)$ using SARSA, we can extract the optimal policy using the equation (1).

3 Results

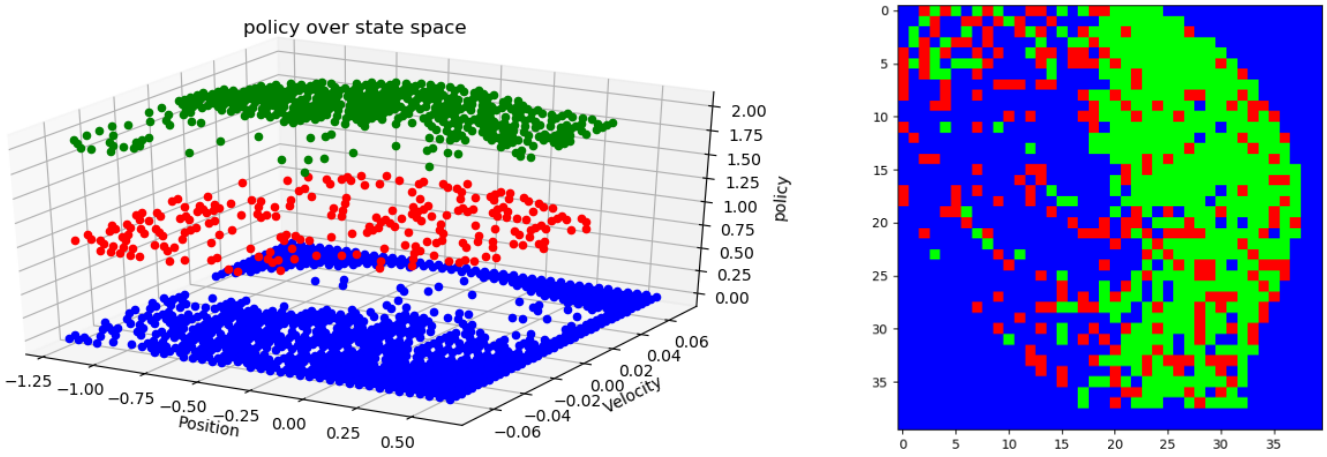
3.1 The optimized policy over the state space

Blue dot indicates action 0 (driving left), red dot indicates action 1 (no using engine), green dot indicates action 2 (driving right).

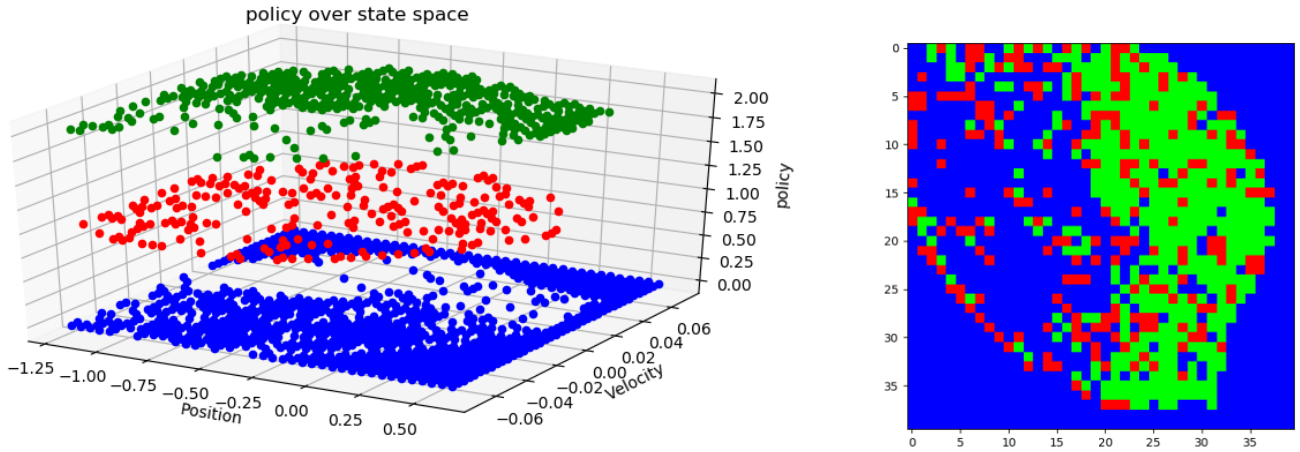
Optimized policy after 3000 episodes (SARSA)



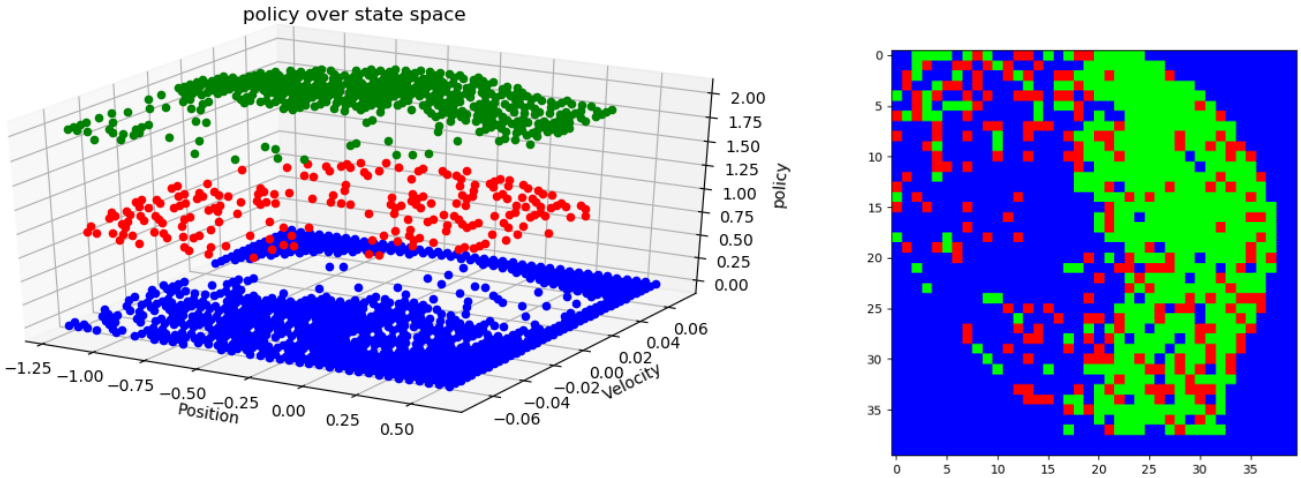
Optimized policy after 3000 episodes (Q-Learning)



Optimized policy after 5000 episodes (SARSA)



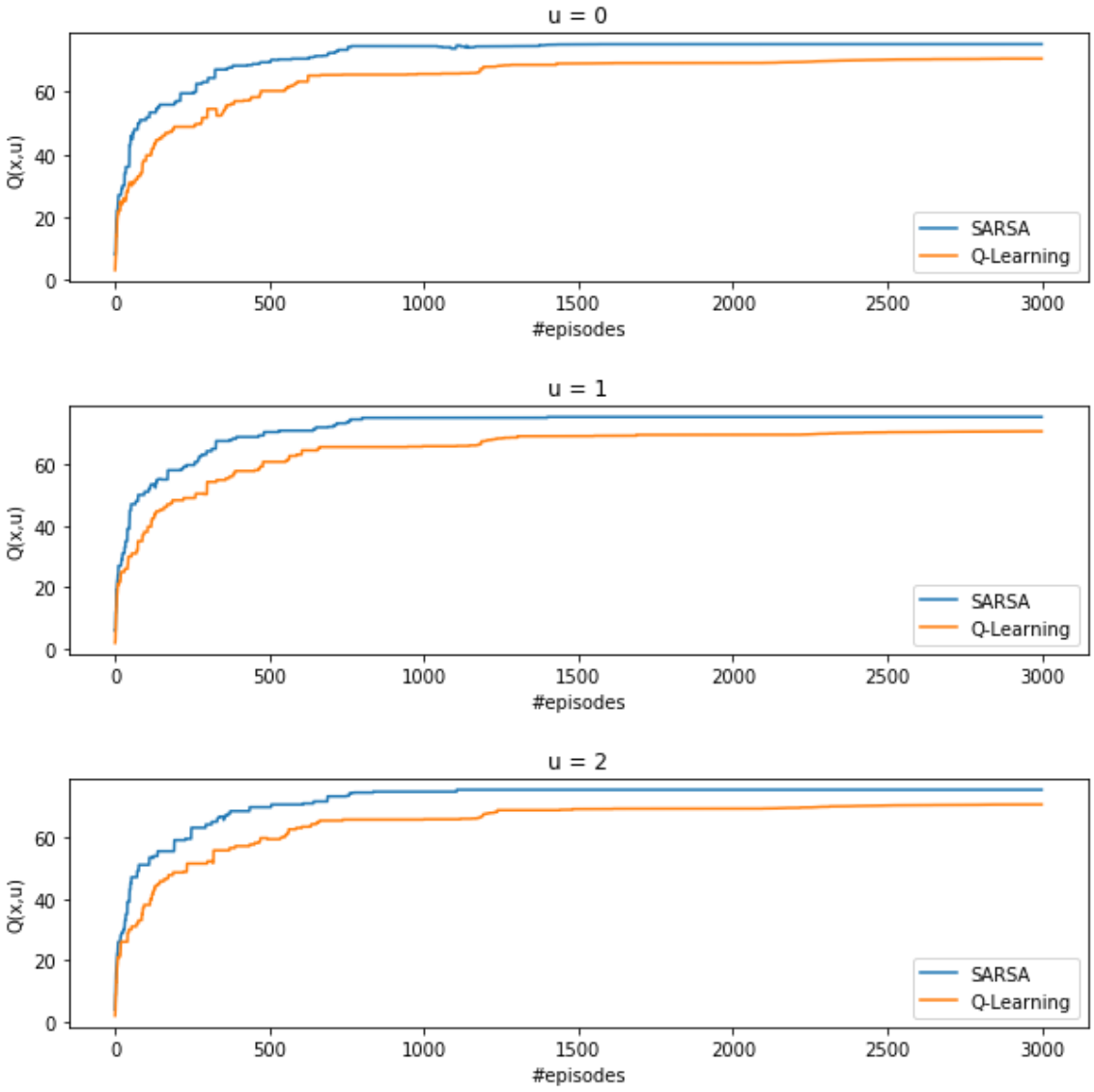
Optimized policy after 5000 episodes (Q-Learning)



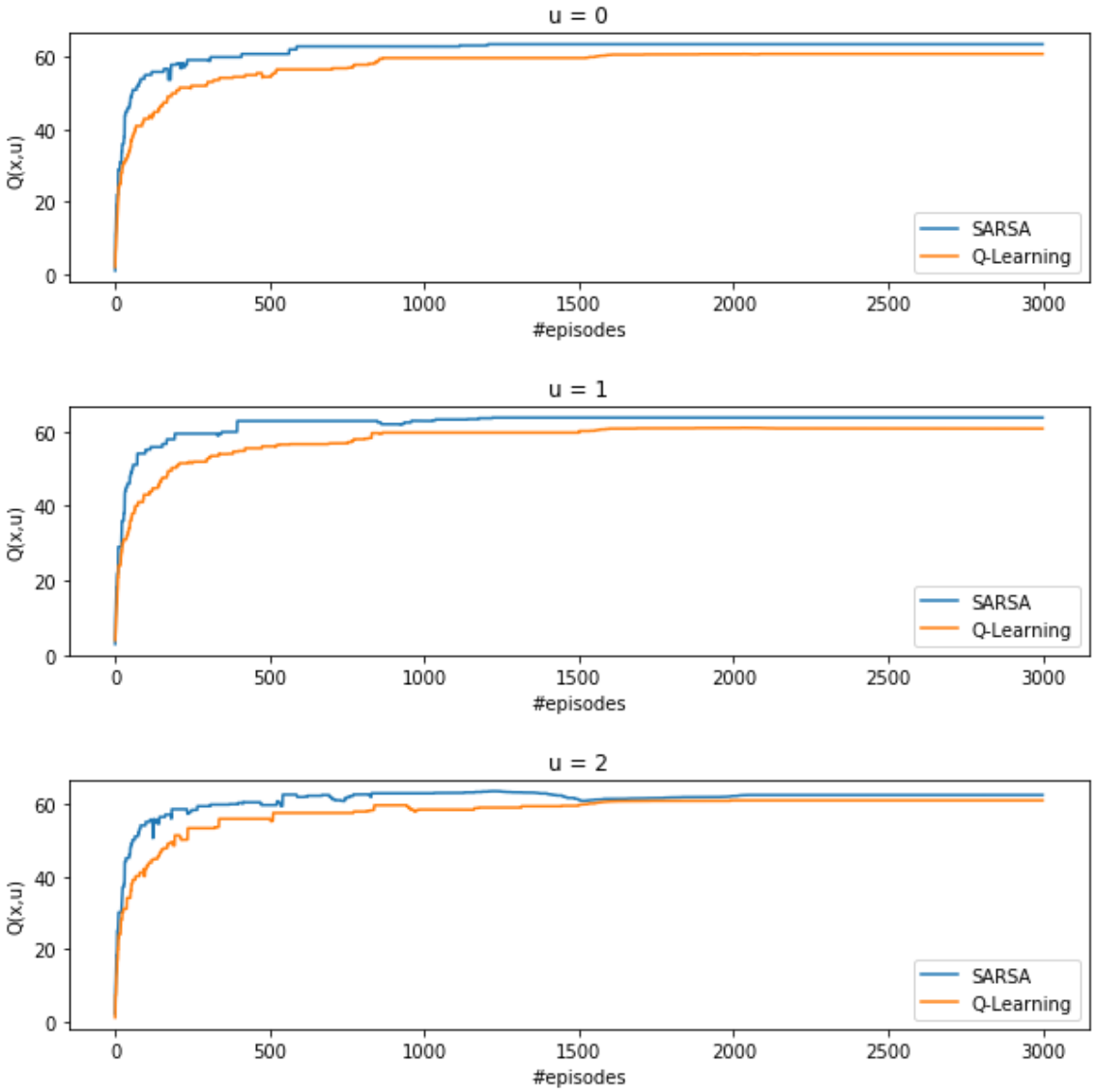
3.2 $Q(x,u)$ over a number of episodes for a set of states

Learning rate decay with discount factor $\gamma = 1$.

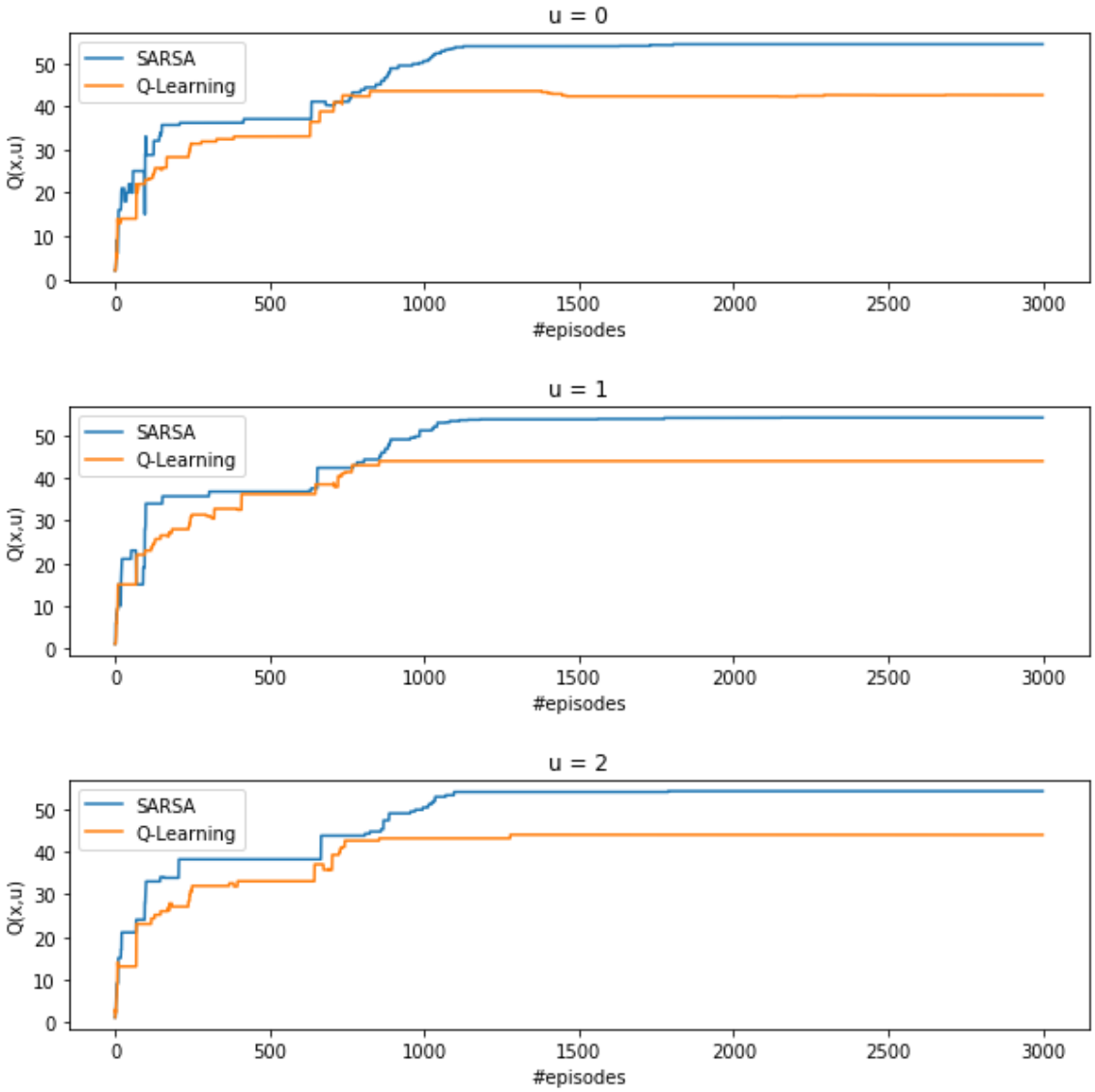
State x = (0,0)



State x = (-1,0)


u = 0
Q(x,u)
#episodes
SARSA
Q-Learning
u = 1
Q(x,u)
#episodes
SARSA
Q-Learning
u = 2
Q(x,u)
#episodes
SARSA
Q-Learning

State x = (0.25,0)



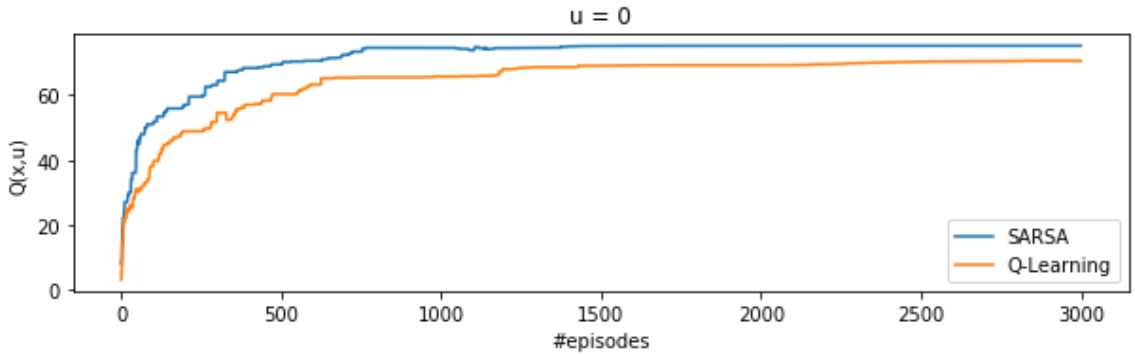
From the figures above, we can see that SARSA is converging faster than Q-Learning in general.

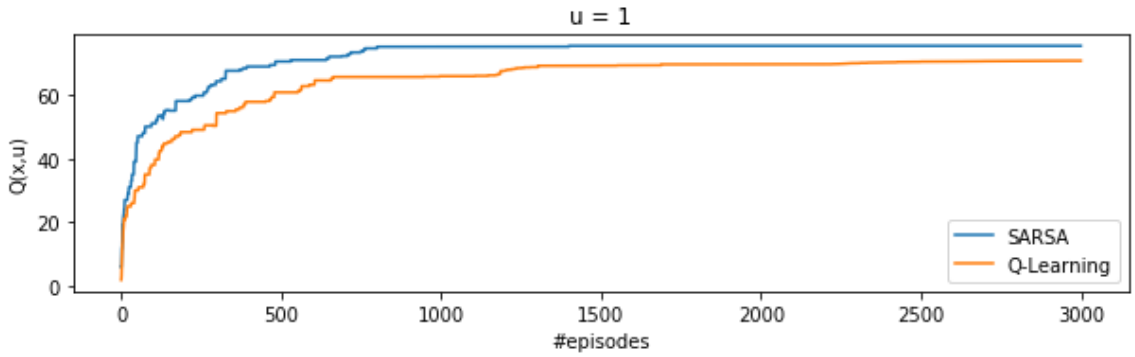
3.3 Hyperparameters analysis

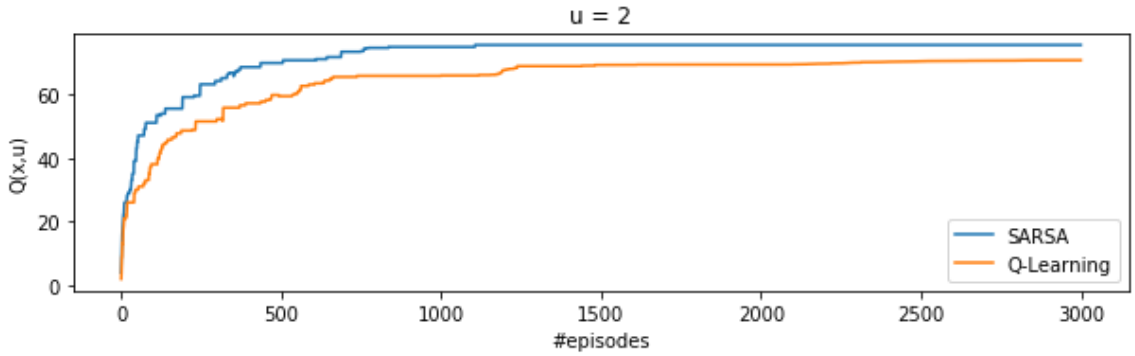
I ran few experiments with different discount factors and fixed learning rate. From my observation, with learning rate decay, lower discount factors (0.9 and 0.8) will make the Q-value converges faster but the policy converges slower (see figures below). If I fixed the discount factor, the learning rate should be set carefully. If it is too small, the algorithm will converge too slow, if it is too big, the algorithm is hard to converge. Therefore, I think learning rate decay is a good method to overcome such problems.

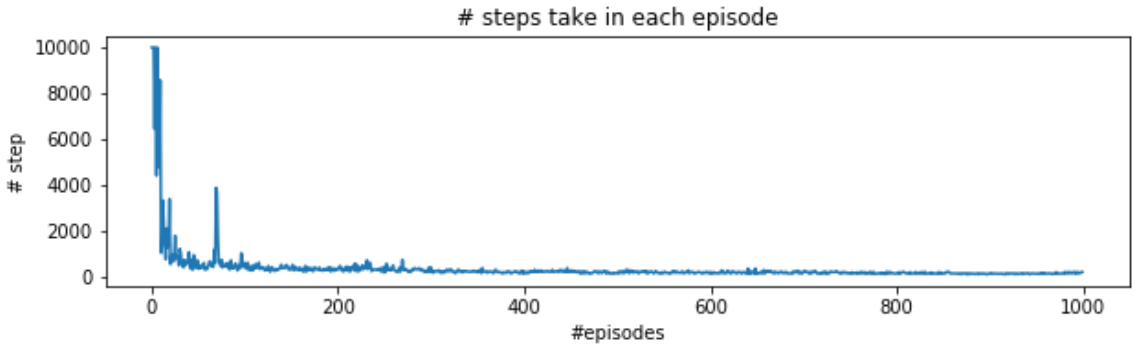
Learning rate decay with discount factor $\gamma = 1$

State x = (0,0)



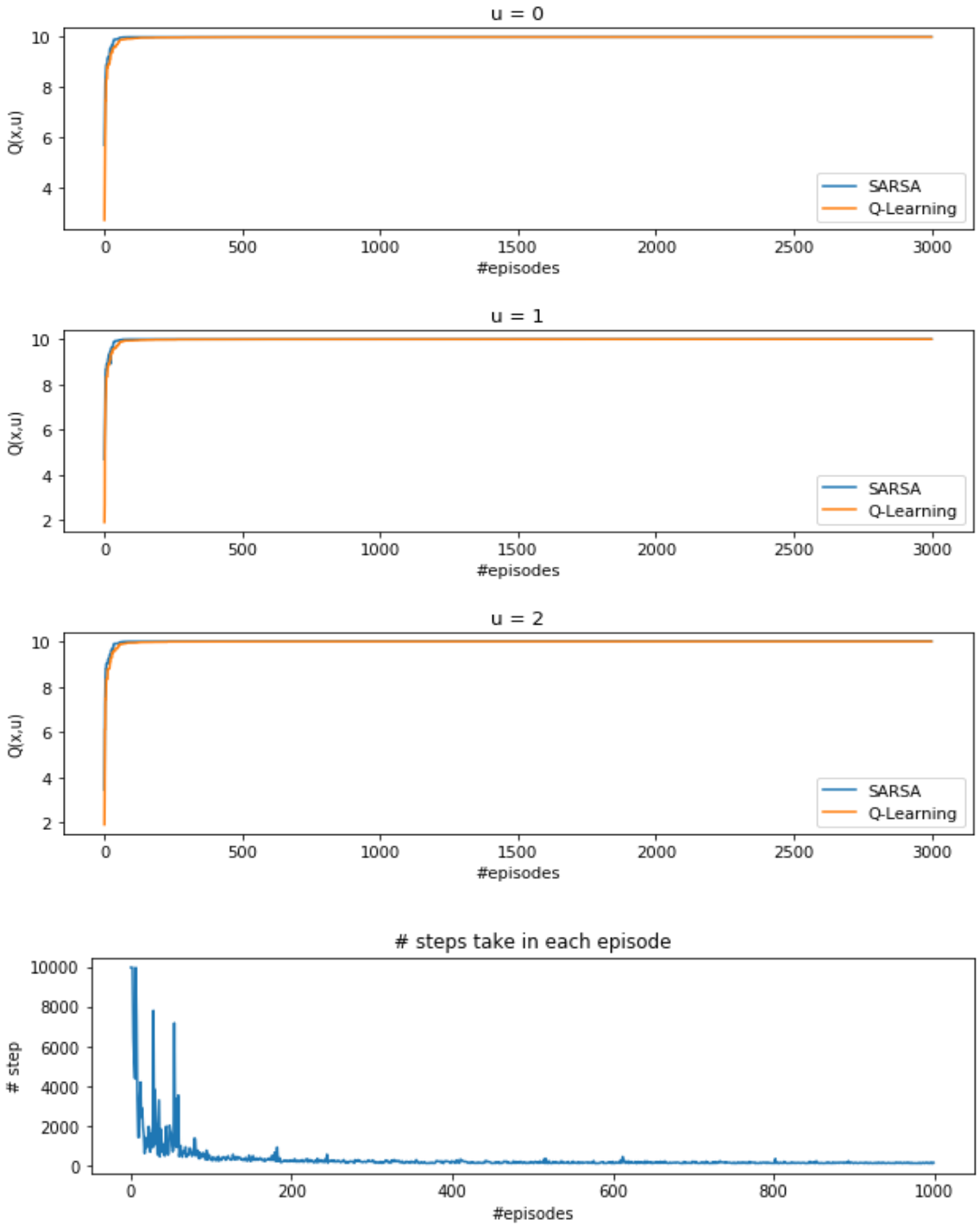






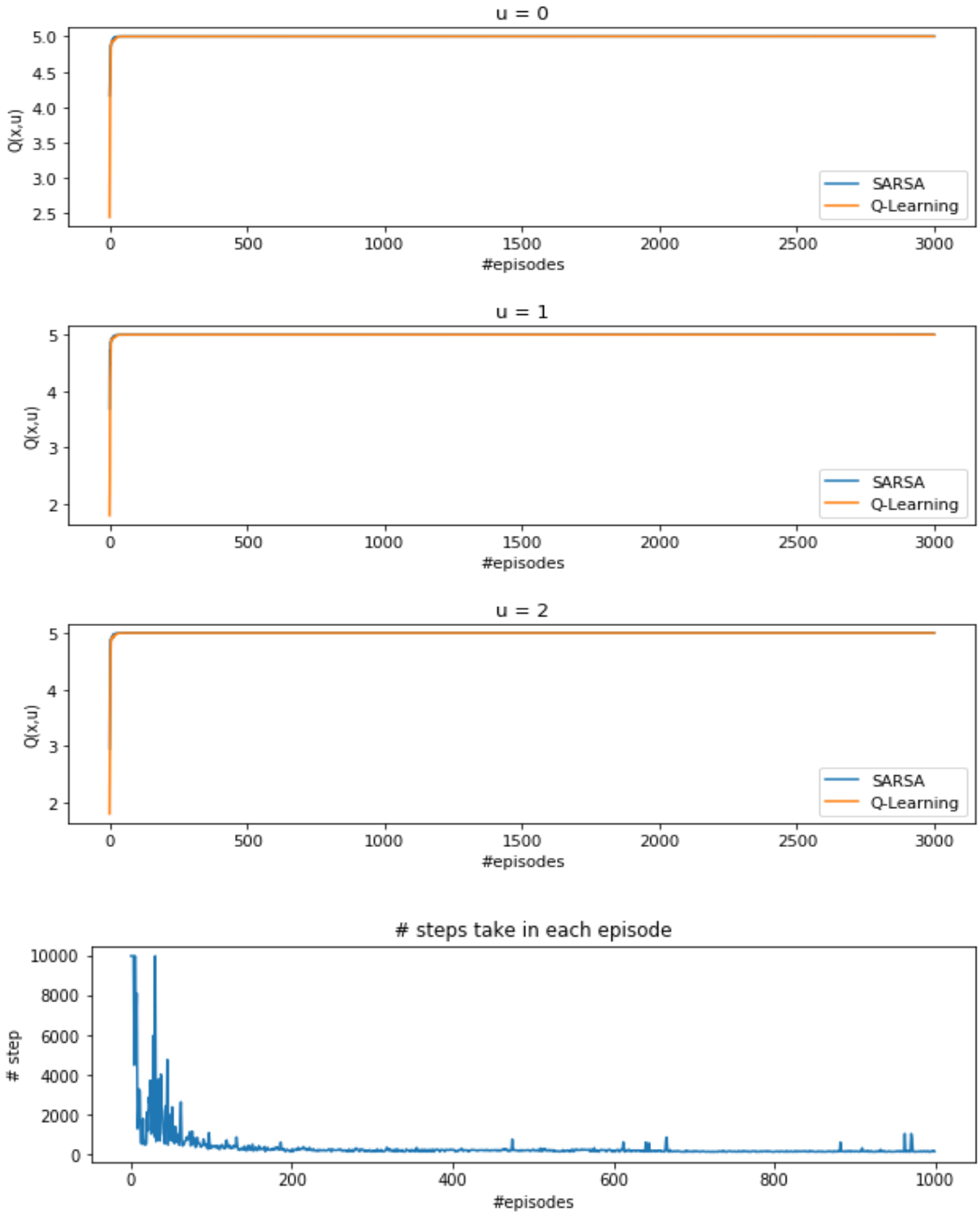
Learning rate decay with discount factor $\gamma = 0.9$

State x = (0,0)



Learning rate decay with discount factor $\gamma = 0.8$

State x = (0,0)



References

[1] https://medium.com/@m.alzantot/deep-reinforcement-learning-demysitifed-episode-2-policy-iteration-value-iteration-and-q-978f9e89ddaa