University of Cambridge

MPhil MLMI

Reinforcement Learning

Candidate Number:
J902G

Date:
March 14, 2022

Word Count: 1992[a]

[a]Not including the introduction, footnotes, references, or the appendix.

1 Introduction

This report investigates the performances and efficiencies of a variety of reinforcement learning algorithms. The algorithms fall into two classes: dynamic programming methods and temporal difference (TD) methods. For the former this report investigates policy and (synchronous/asynchronous) value iteration, and for the latter it compares the three 1-step TD methods; SARSA, Expected SARSA, and Q-learning. Finally, function approximation is discussed in the context of stochastic states that change during learning.

Throughout these investigations three grid worlds are focused on, small world which is a small and simple maze, grid world which is similar to small world except with more states and barriers, and cliff world, which has the additional complexity that it is possible to step onto the cliff edge, in which case a large penalty is incurred by the agent. The rules of each are defined as follows:

1. Small world: 16 states, 3 of which are barriers. The goal state has reward of 10, and the remaining states have rewards of -1.
2. Grid world: 108 states, 9 of which are barriers. There is 1 bad cell with reward of -6, and the others have reward of -1, except for the goal state which has reward 10.
3. Cliff world: 50 states, with 7 on the bottom row having reward -100 and representing falling off of the cliff, the goal state having reward of 10, and the rest having -1.

2 Part A: Policy vs Value Iteration

For worlds with a small number of states like the ones considered in this investigation, it is feasible to iteratively explore every state in order to guarantee that the agent is not missing out on a high reward due to insufficient exploration. Policy iteration and value iteration both take this approach, and in both use the Bellmann optimality equation to update the value function each iteration according to[1]: [6]

$$V_\pi(s) = \sum_{r,s'} P(s', r|s, a) \left[r + \gamma V_\pi(s')\right] \tag{1}$$

The diversion between the algorithms is that unlike policy iteration (which evaluates a policy by calculating the value it gives to all states according to Equation (1) and then updates their policies using the argmax), value iteration directly improves the value function each iteration, without the need to explicitly define a policy. Mathematically it is equivalent to carrying out a single sweep of policy evaluation and combining both steps of policy iteration to give:

$$V_{k+1}(s) = \max_a \sum_{r,s'} P(s', r|s, a) \left[r + \gamma V_k(s')\right] \tag{2}$$

Figures 1, 2 (Left), and 3 (Left) show the learned policy on grid world using policy iteration, synchronous and asynchronous value iteration respectively. As they are the same it appears value iteration and policy iteration do indeed converge to the same result as expected. However, we needed to define when to terminate value iteration, as complete convergence of the value function requires theoretically infinite iterations, and there is no policy to check for convergence like in policy iteration. Therefore, a small threshold θ is used, and when the maximal (over states) change in value function is below it, the algorithm terminates (set to 0.0001 here).

There are two variants of value iteration, synchronous and asynchronous, and the difference between them lies in whether the update to the value function is done on using the previous iteration's 'held-out' values (synchronous), or is continuously updating and then using these updated values for other state updates within an iteration (asynchronous). Figure 2 shows the synchronous case and Figure 3 the asynchronous one. Whilst it may seem theoretically the 'incorrect' thing to do to overwrite the functions within an iteration, it reduces memory needs and can often speed up convergence as it is always using the most up-to-date information for each state update. From the right of both figures it appears that there is little difference in the convergence of the state values. Indeed as iterations tend to infinity, they will certainly both converge to the same value as each state's value function is updated an infinite number of times [4].

In terms of algorithmic cost differences between policy and value iteration, policy iteration has algorithmic complexity $\mathcal{O}((|\mathcal{A}| + k)|\mathcal{S}|^2|)$ per iteration (for k policy evaluation steps per iteration), and value iteration has complexity $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|)$ per iteration [1]. (For both, $|\mathcal{S}|^2$ is contributed by summing over all next states for each current state and $|\mathcal{A}|$ from finding the maximum over actions. The k in policy iteration comes from the multiple evaluations of the policy). Therefore, for a single iteration, policy iteration has higher algorithmic cost than value iteration. However, as value iteration never explicitly calculates the policy to gauge convergence it takes a theoretically infinite number of iterations to converge (if the threshold is 0, see Appendix 6.1 for proof), compared to policy iteration being guaranteed to converge in finite steps once the policy stops changing. In this sense then, policy iteration can be thought of as more efficient overall than value iteration, although for a reasonable threshold, value iteration may approximately converge (and the policy represented by the value function stop changing) faster than policy iteration[2].

[1]The update step below guarantees us to improve, or keep equally optimal, the value function at every new time step. [6].

[2]For example, on small world, for $k = 5$, asynchronous value iteration called Equation (1) 27 times compared to policy iteration's 77 when $\theta = 0.0001$.

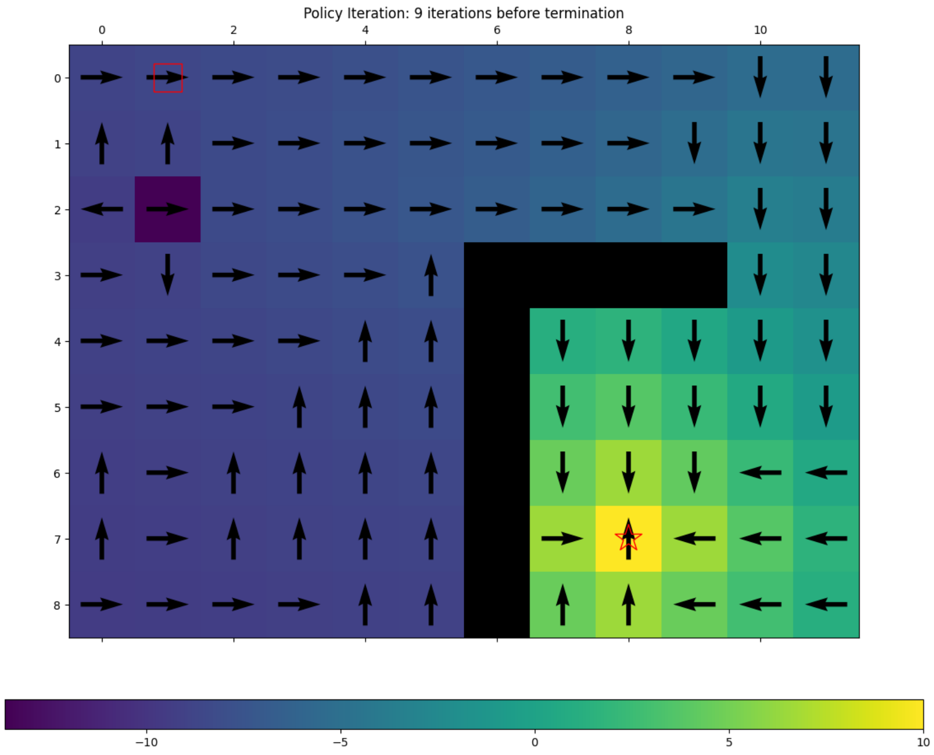
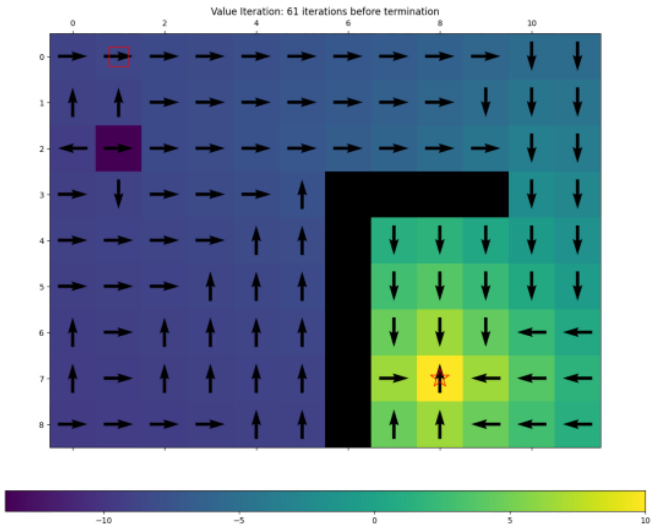


Figure 1: Plot of the learned policy and state values after 9 iterations of policy iteration on grid world. Each policy evaluation loop inside an iteration contains 5 sub-iterations.



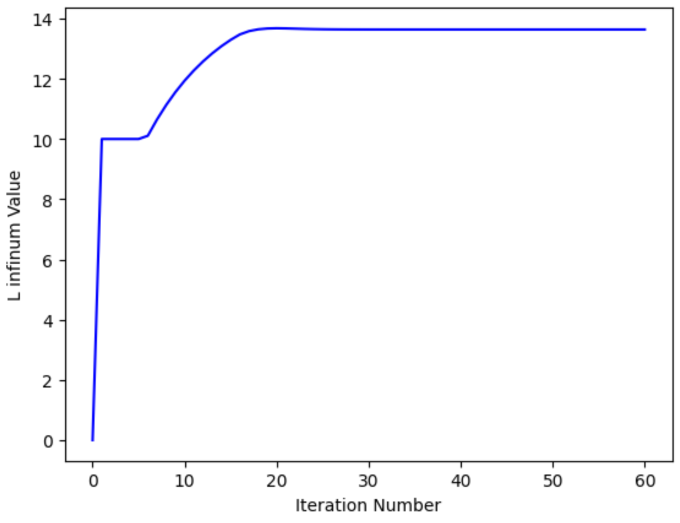


Figure 2: (Left) Plot of the learned policy and values for each state of grid world learned via synchronous value iteration. (Right) The convergence of the infinum over state values with iteration number (initialised to 0 for all states). Note that the absolute value of the mean state values does not give much information on the quality of the learned policy (as the value function values are very much dependent on the world used), but this figure still gives an idea of the speed of convergence of the synchronous value iteration.

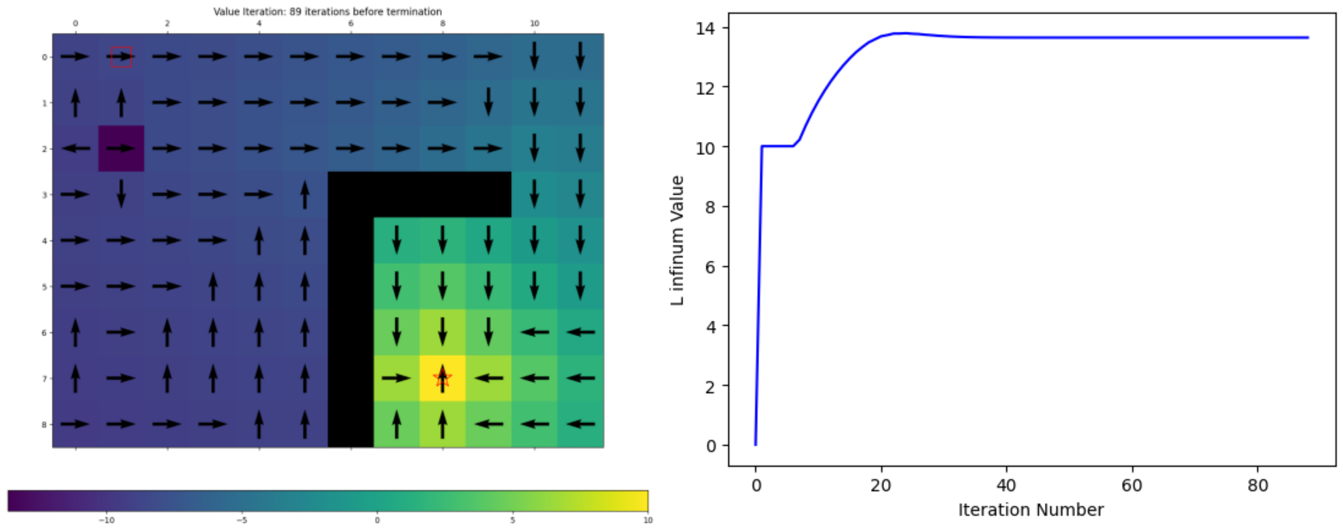


Figure 3: (Left) Plot of the learned policy and values for each state of grid world learned via asynchronous value iteration. (Right) The convergence of the infinum of state values for this learning algorithm, demonstrating the rate of convergence to an unchanging value function and similarity with synchronous value iteration.

3 Part B: SARSA and Expected SARSA

Whilst the convergence behaviours of value and policy iteration are stable and guaranteed, repeatedly summing over all states to compute the value update is cumbersome and causes agents to train slowly in large worlds. An alternative approach is to use monte-carlo sampling, where one action (or more for multi-step variants) is taken ϵ-greedily to reach a next state and the value function updated from there. These methods are accordingly called temporal difference methods due to the time lag between updates. One such method is SARSA ($\langle s, a, r, s', a' \rangle$), an on-policy method which for a current state-action pair moves to a new state stochastically according to that action, observes a reward, and then ϵ-greedily selects a new action. The update to the Q-function is then calculated via:

$$Q_{k+1}(s, a) = Q_k(s, a) + \alpha[r + \gamma Q_k(s', a') - Q_k(s, a)] \tag{3}$$

A plot of the implementation of SARSA on small world is shown in Figure 4 after 1000 episodes. Whilst it is usual to terminate learning after convergence using the same threshold method as discussed in Part A, for the TD methods in Parts B and C, episodes were continued after approximate convergence for easier comparison between the algorithms.

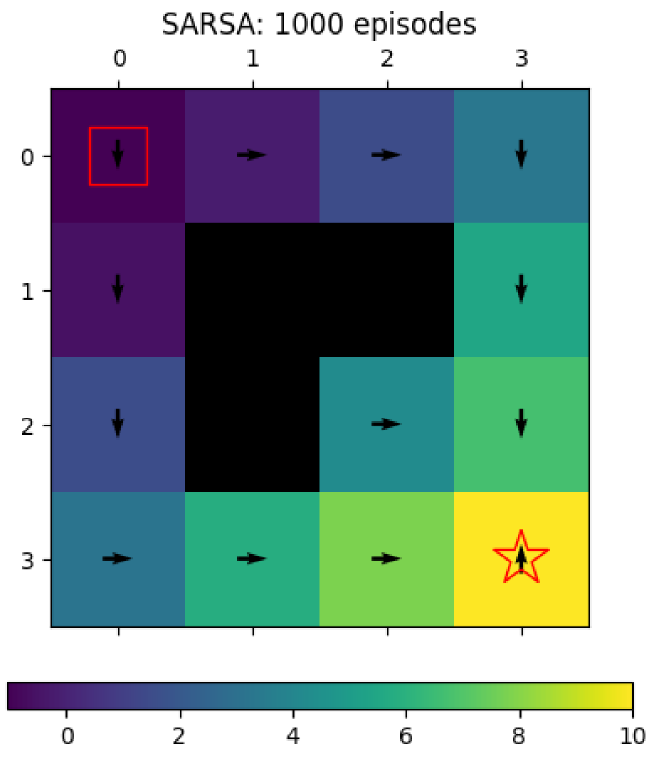


Figure 4: A plot of the learned policy and state values in small world, learned via the SARSA algorithm over 1000 episodes. It is clear that 1000 episodes is sufficient for the algorithm to learn the most direct route from the start state to the end state.

There are therefore two parameters to tune for TD methods such as SARSA: the learning rate, α, and (1 minus) greediness, ϵ. Figure 5 shows a plot of the cumulative rewards per episode for 12 combinations of α and ϵ. The optimal convergence was observed for $\alpha = 0.5$ and $\epsilon = 0.01$. Setting ϵ slightly higher seems not to make a large difference, but too high a value causes the agent to take actions at random too often, and does not sufficiently exploit the learned best actions even after many iterations (i.e. counter-productive random actions are still often selected after the policy has been learned). Similarly, higher αs cause instability in training as the Q-function struggles to stabilise at an eventual optimum.

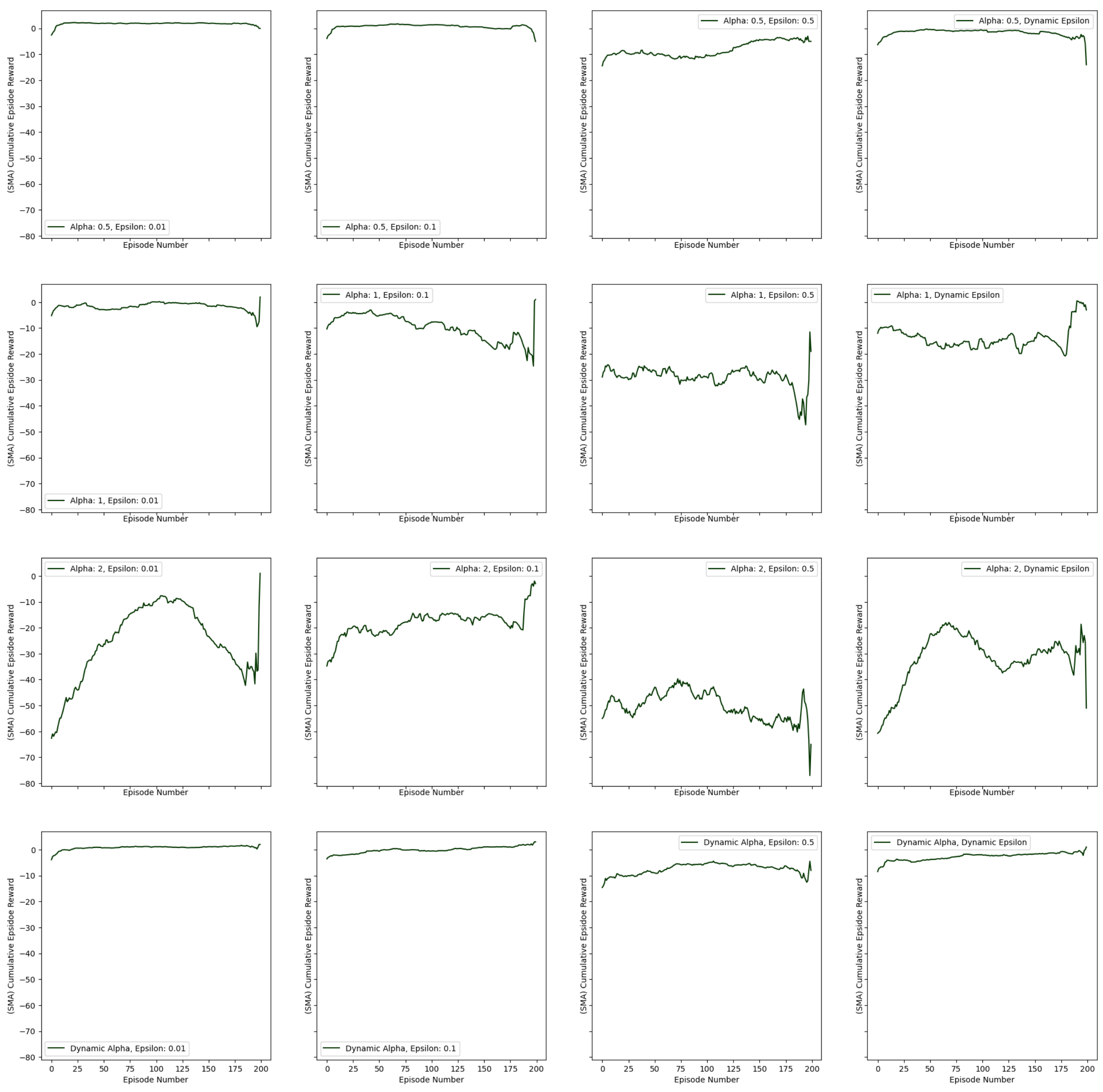


Figure 5: Grid of plots of the cumulative episode rewards for SARSA on small world, demonstrating the effect that altering the learning rate and ϵ parameter have on learning. The rightmost plots are for a dynamic $\epsilon_n = 1/n$, which can help convergence as the agent becomes more exploitative within an episode and therefore takes less greedy actions once it understands the world better (dynamic parameters are discussed in more depth in Appendix 6.2). Dynamic learning rate is also explored in the bottom row with $\alpha_n = 1/n$, which causes the agent to make smaller Q-function update steps towards the end of an episode. Dynamic parameters are expected to help SARSA in particular, as on-policy methods suffer from the fact that the target policy is still being updated with the ϵ-greedy behaviour policy even when the world is better known towards the end of an episode, meaning that Q-function updates near the end of an episode do not make the most of the information available to the agent.

If we want to guarantee convergence then dynamic α and ϵ can be used s.t. $\sum_{n=1}^{\infty} \alpha_n = \infty$, $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$, for example, with $\alpha_n = 1/n$ and $\epsilon_n = 1/n$ [6] (see Appendix 6.2 for details). This also helps with the exploitation vs exploration problem when setting parameters, as the agent now becomes more exploitative at the later stage of an episode when it has had more experience in the world, and the decreasing learning rate prevents overshooting the optima and thus helps stable convergence. However, with dynamic ϵ and α it becomes even more important to restrict the number number of iterations of SARSA per episode before episode termination, as if the iterations grow too large then both tend to 0, and the agent (a) stops updating its Q-function as the step-size is 0, and (b) takes all actions completely greedily, such that if the optimal policy had not yet been discovered, the agent can be stuck in a cycle of non-terminal states and never reach the terminal state. A suitable maximum iteration number might be 100 for small world.

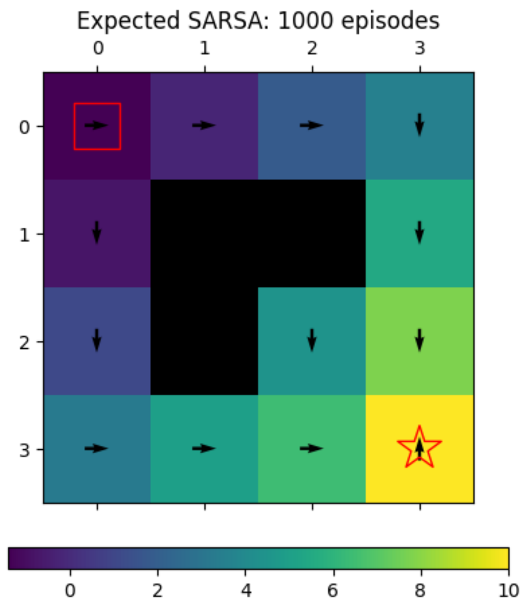


Figure 6: A similar plot to in Figure 4, but this time for the Expected SARSA algorithm. The learned policy is almost the same as that of SARSA (except for initially moving right rather than down, but this does not effect reward), and the state values are also the same after 1000 episodes. This is to be expected for a world with few states such as small world when there are a high number of episodes. However, as is confirmed below in Part D, we expect Expected SARSA to converge faster and more smoothly than SARSA for larger worlds.

A variation of SARSA is to alter Equation (3) to instead update using the expectation over the Q-function of the next state-action pair, thus more efficiently accounting for how likely each action is under the current policy (see Equation (4) below):

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha[r + \gamma \underset{a'}{\mathbb{E}}\left[Q_k(s',a')|s'\right] - Q_k(s,a)] \tag{4}$$

We expect Expected SARSA to converge more smoothly than SARSA as the Q-function update is now independent of the ϵ-greedily chosen next action (whose stochasticity can cause larger variances in rewards and Q-function during learning) - in other words, SARSA 'moves'[3] from the next state on average in the way that Expected SARSA 'moves' deterministically [6]. Figure 6 shows the learned policy of Expected SARSA on small world, and is similar to SARSA's policy, except for moving down first and not right (both have equal value though due to symmetry of the world).

Figure 7 shows the cumulative rewards per episode plots for the same parameters as for SARSA above. Comparing the two it can be seen that the optimal parameters are the same, and Expected SARSA achieves marginally better rewards. What is more, it seems more robust to the learning rate, and also ϵ as now

[3]I use the quotation marks to signify that the actual next step a' is taken ϵ-greedily in both, but as far as the Q-function update is concerned, it is as if we have moved with that action from the next state. I.e. the behaviour policy and target policy are different for Expected SARSA but the same for SARSA.

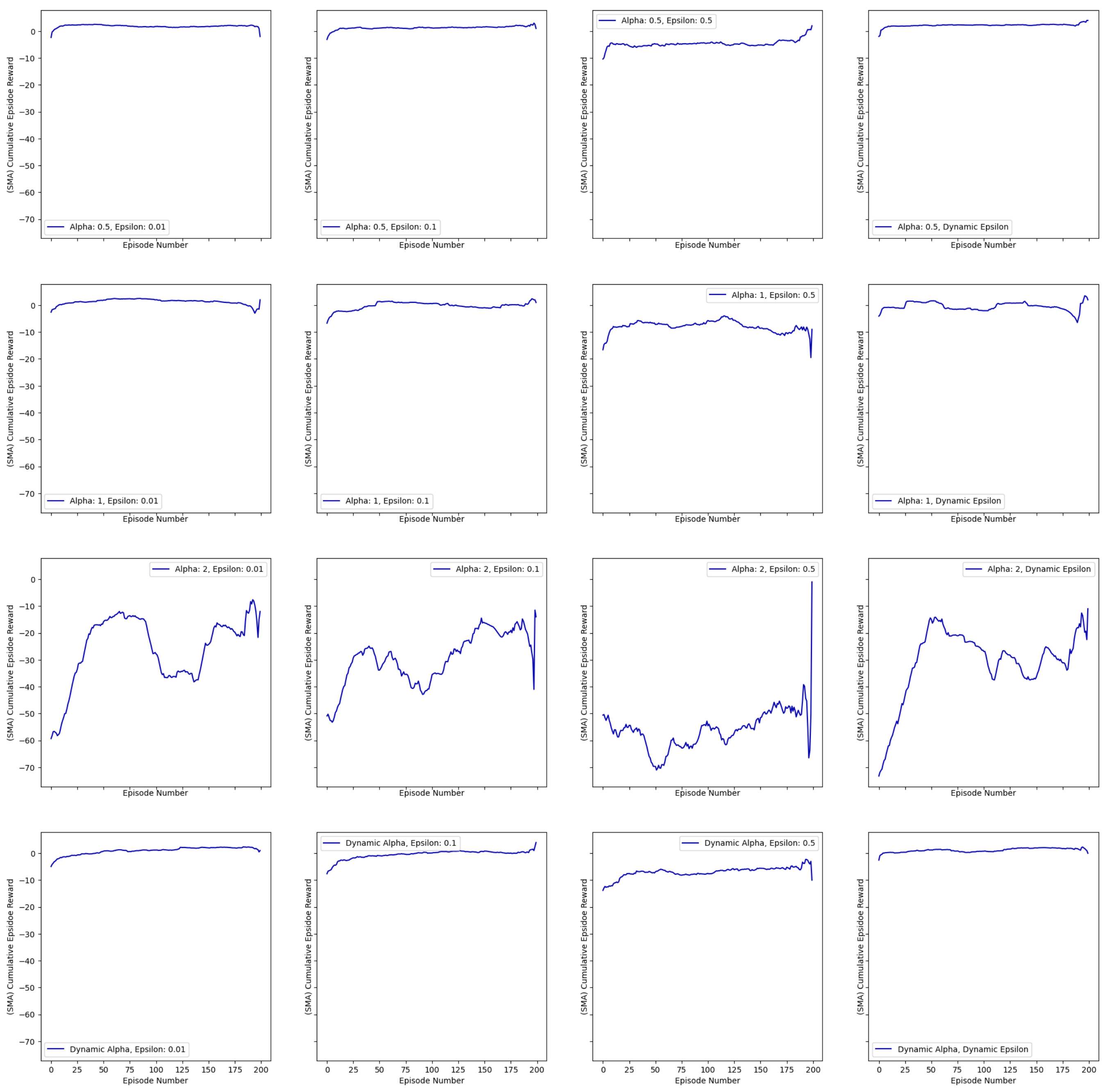


Figure 7: Grid of plots again demonstrating the effects of varying α and ϵ when using the Expected SARSA algorithm. The rightmost plots are for a dynamic $\epsilon_n = 1/n$ and the final row $\alpha = 1/n$ again.

the Q-function update depends less explicitly on it (now that the update step ignores the next chosen action a'). Too high parameters still break convergence though as Expected SARSA is not completely independent from them.

To conclude therefore, whilst Expected SARSA requires explicit computation of the expectation value on each iteration and is therefore more computationally expensive, it converges faster and more predictably than SARSA, and so its overall efficiency and performance is better.

4 Part C: Q-Learning

An off-policy[4] variation of SARSA is Q-learning, which still explores the world by having an ϵ-greedy behaviour policy, but now has a greedy target policy, meaning that the Q-function updates every step on the greediest action a' regardless of the behaviour policy action. The update step is summarised by:

$$Q_{k+1}(s, a) = Q_k(s, a) + \alpha[r + \gamma \max_{a'} Q_k(s', a') - Q_k(s, a)] \tag{5}$$

We still have the two parameters to tune, and Figure 9 shows the cumulative rewards per episode for different parameter settings under the Q-learning algorithm. The optimal combination was now $\alpha = 0.5$, $\epsilon = 0.1$. Compared to SARSA, Q-learning is much more robust to the value of ϵ, as the Q-function update always takes the greediest action for a' according to Equation (5), and only depends on ϵ in the behaviour policy (although the dependence of Q-learning on ϵ would be expected to be greater in higher-risk worlds like cliff world, as Q-learning does not learn to stay clear of high-risk states, and so a higher ϵ means a higher chance of stepping into a very bad reward). Notably, Q-learning is also more robust to learning rate than SARSA, but once again becomes unstable and does not converge if learning rate is too high. We can also again guarantee convergence if dynamic α and ϵ are used according to the stochastic approximation convergence rules discussed in Part B and Appendix 6.2.

In terms of the policy learned, whilst SARSA learns a near-optimal policy (for small ϵ) whilst exploring, Q-learning learns the optimal policy directly and is therefore less conservative. This can lead to higher variance as initially Q-learning has not got the experience to accurately decide on the optimum policy, so it makes more frequent bad decisions (discussed in more details for cliff world in Part D). For the small world however, it can be seen in Figure 8 that this does not affect the learned policy.

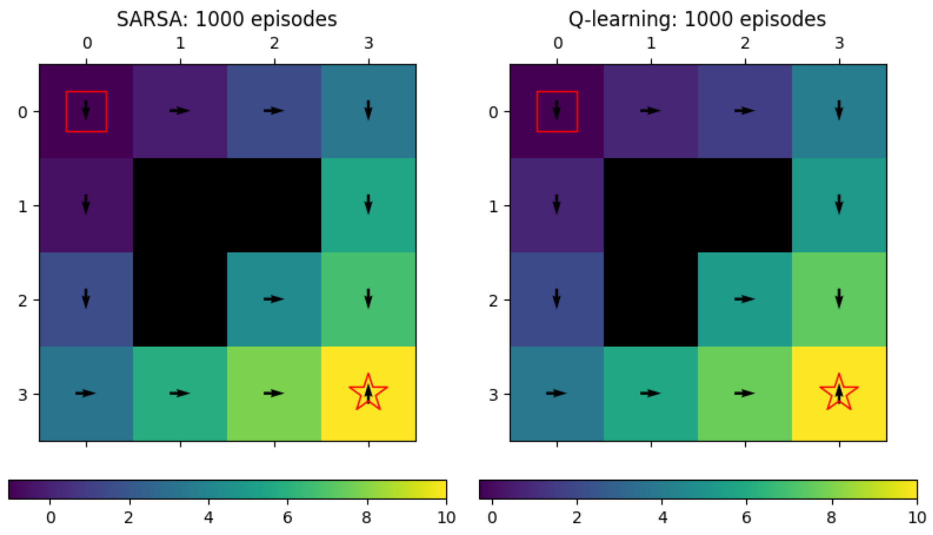


Figure 8: A comparison of the learned policies after 1000 iterations for small world, using Q-Learning and SARSA. Similarly to the comparison between the policies of SARSA and Expected SARSA, the Q-Learning policy is also the same.

[4]Off-policy because it updates the Q-function using a greedy policy, even though it's not using the greedy policy to explore the world. In comparison, SARSA's update depends on the current policy, and is therefore an on-policy method.

When comparing efficiency, it may seem more efficient to directly learn the optimal policy as in Q-learning, as it aims to move more directly towards the goal state. However, in practice it is less efficient than SARSA, as Q-learning's tendency to take high-risk strategies means that large negative rewards are more often incurred and learning is less stable - hence convergence often takes longer with Q-learning than it does with SARSA, particularly for worlds with high risks/large negative rewards. This is supported as, on average, the number of iterations per episode over all episodes is greater for Q-learning than (Expected) SARSA, with an average of 14.0 and (11.5) 12.7 iterations respectively on small world. In general, on-policy methods such as (Expected) SARSA are faster converging than their off-policy counterparts.

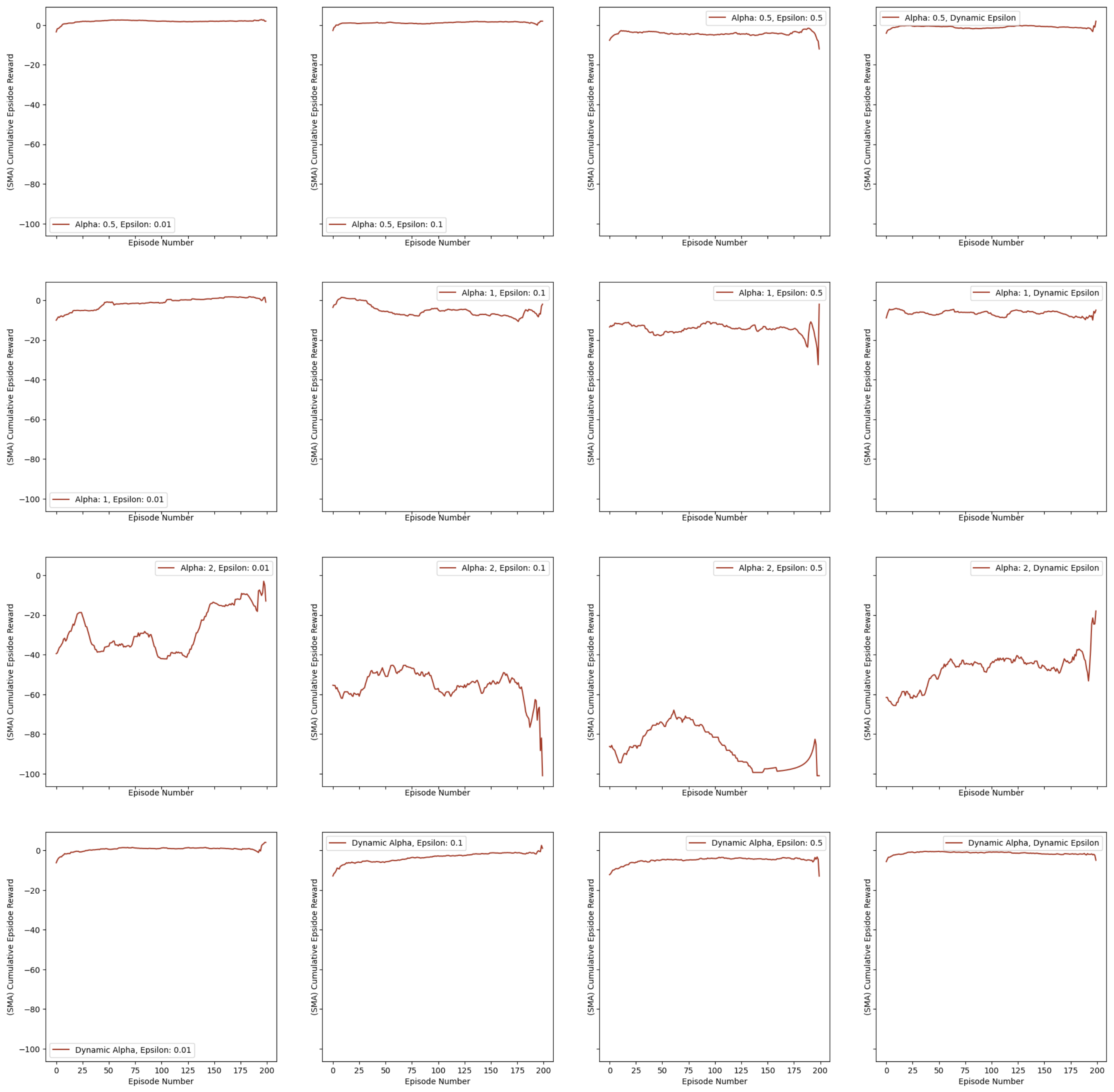


Figure 9: The effect of α and ϵ on learning under the Q-Learning algorithm. The final column uses again dynamic $\epsilon_n = 1/n$ and the final row $\alpha_n = 1/n$.

5 Part D: Comparing Temporal Difference Methods

To compare algorithms it is useful to plot their rewards on the same plot. This is done in Figure 10 for Q-learning, SARSA and Expected SARSA on cliff world. Expected SARSA pulls away from the other two methods and converges more smoothly, as it contains the advantages of both Q-learning and SARSA: like SARSA, it is more conservative and not as greedy as Q-learning, but in comparison to SARSA, the deterministic nature of the its Q-function updates reduces cumulative reward variance. Comparing SARSA and Q-learning, SARSA performs better with slightly lower variance. The reasoning behind this is as follows and can be seen from the grids in Figures 11 to 13.

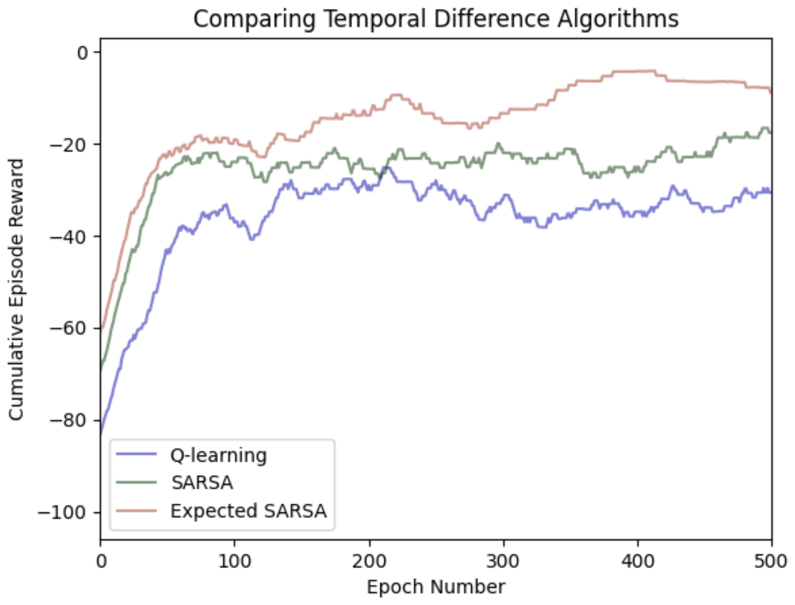


Figure 10: A comparison of the improvement of cumulative episode rewards on cliff world as a function of episode number for the TD algorithms. The rewards contain a high amount noise which made the behaviour harder to see, so a simple moving average with a window size of 100 has been applied to all three curves.

Q-learning learns the optimal policy, which after the first few episodes coincides with following the cliff edge closely to minimise step number (see Figure 13). However, as the next action selection is still ϵ-greedy, the agent will fall off the cliff with probability approximately given by $\epsilon/4$ each step (true for the non-edge states next to the cliff).

In comparison, SARSA updates its Q-function according to its ϵ-greedy behaviour policy, and therefore favours moving away from the cliff edge quickly as it learns sooner that staying near the cliff can lead to very negative rewards (its on-policy learning means that if it falls off the cliff then it updates its Q-function based on that fall and gives the cliff-state a low value, favouring moving away from the next-to-cliff state for future iterations). This makes SARSA more conservative, and leads to better performance in worlds with high-risk states such as cliff world. This explains the differences in cumulative return seen in Figure 10, as Q-learning more often receives very negative reward to SARSA and so the smoothed curve is lower despite it eventually taking fewer steps. This can also be seen from the cliff states having value further from 0 (indicating that they have been visited more often) in Q-learning compared to SARSA/Expected SARSA.

The policy for Expected SARSA behaves similarly to SARSA, also moving away slightly from the cliff edge due to its Q-function update being less exploitative and more explorative than Q-learning.

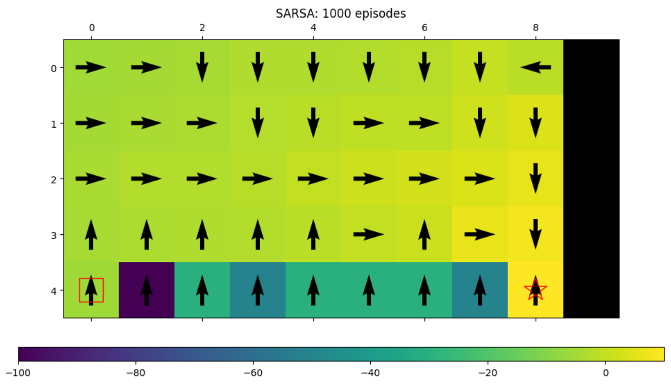


Figure 11: Plot of the learned policy on cliff world after 1000 episodes of the SARSA algorithm. The learned policy appears to be conservative, moving away from the cliff edge, which is in agreement with the theoretical predictions [6]. The parameters were set to $\alpha = 0.5$, $\epsilon = 0.1$ for this plot.

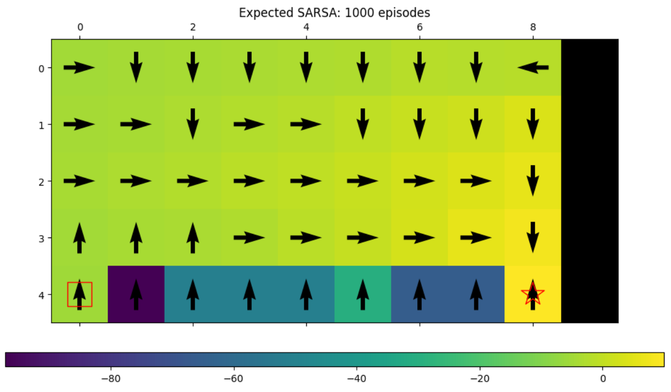


Figure 12: Plot of the learned policy on cliff world after 1000 episodes of the Expected SARSA algorithm. The policy for Expected SARSA also moves two steps away from the cliff edge when crossing left-to-right. The parameters were set to $\alpha = 0.5$, $\epsilon = 0.1$ for this plot.

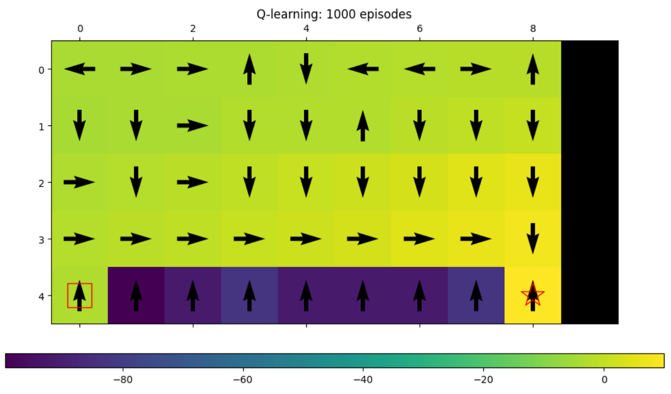


Figure 13: Plot of the learned policy on cliff world after 1000 episodes of the Q-Learning algorithm. The learned policy is exploitative as expected, and stays close the cliff edge to minimise the distance travelled to arrive at the end state. The parameters were set to $\alpha = 0.5$, $\epsilon = 0.1$ for this plot.

6 Part E: A Stochastic Small-World

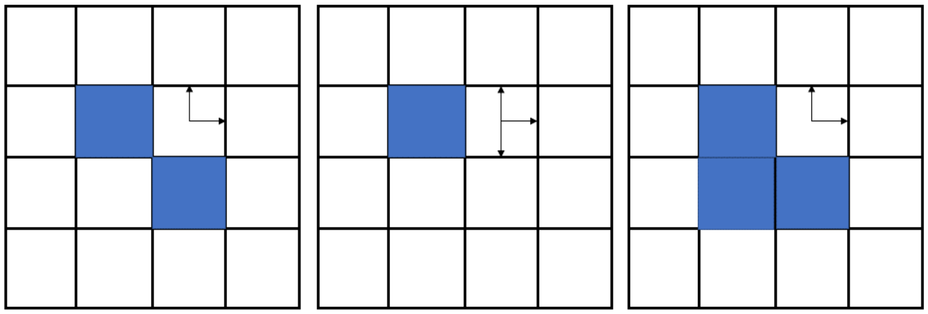


Figure 14: Plot of examples of the new small world, the barriers in the centre of the world now appear stochastically, increasing the number of states 8-fold. It is worth noting that a barrier changing does not affect all states, only the ones adjacent, and even these do not change for all barriers changing (for example see the left and right configurations, the shown state next to the barrier has the same actions available despite the world being in a different state).

The adaptations of this task now introduce a problem of generalisation to the previously considered models. In the DP algorithms, the value functions were represented in a tabulated form over states, and so the 3 stochastic barrier states in the new problem now present an issue of increasing by 8 times the size of the state space[5]. For large worlds this therefore leads to computational intractability, particularly if more states were to be made stochastic. This is also a problem for the TD methods, as they can only practically converge to the best value if they explore each state a sufficient number of times, and this can again become computationally very expensive.

Therefore an approach is to take a method with greater generalisation that does not require this tabulated value function form. This is done via approximating the value function by a parameterised function described by learned weight vectors [6]. The advantage of this formulation is that there are commonly fewer weights than states, so a change of states like considered here will update the weights in a way that then influences many states' values simultaneously and allows for fixed computation requirements that are independent to the size of the state-space (for a fixed number of weights).

The simplest form of function approximation is linear-in-the-weights approximation with the relation: [6]

$$\widetilde{V}(s;\vec{w}) = \vec{w}^T \vec{\phi}(s) \tag{6}$$

where $\vec{\phi}(s)$ is a vector representing basis functions acting on features of state s, e.g. RBF functions with different means acting on a value that encodes the current position. It may also improve learning to use a more complex encoding than simply s, as although distinct, the states representing the same location in a different arrangement will have similar values, and so it makes sense to encode this information before inputting to the function to speed up learning. For example, the input for each state could be a vector with positional information and information on which barriers it borders/whether the state is itself a barrier in its world arrangement.

The complexity of the rules in the world and the number of states dictates how complex this value-function approximation should be, and for large complex worlds deep neural networks are often used [5]. In the case of this world however, the size means that a linear-in-weights function would be sufficient, and has the advantage of being faster to train and less likely to overfit due to fewer parameters. The basis functions would have still to be non-linear though in order to give sufficient flexibility to allow learning with barriers.

[5]There are $2^3 = 8$ combinations of barrier/not-barrier for 3 states, causing the number of states to increase 8-fold.

References

[1] *Berkeley AI Materials: Markov Decision Processes I*. URL: http://ai.berkeley.edu/lecture_slides.html.

[2] Avi Pfeffer; Revised by David Parkes and Ryan P. Adams. *Markov Decision Processes*. URL: https://canvas.uw.edu/files/46146037/.

[3] Eyal Even-Dar and Yishay Mansour. "Learning Rates for Q-Learning". In: *J. Mach. Learn. Res.* 5 (Dec. 2004), pp. 1–25. ISSN: 1532-4435.

[4] Tom Mitchell Mitchell. *10703 Deep Reinforcement Learning*. Sept. 2018.

[5] David Silver. *Lectures on Reinforcement Learning*. URL: https://www.davidsilver.uk/teaching/. 2015.

[6] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

6.1 Convergence of Value/Policy Iteration Proof

Value Iteration Convergence:

The basis of the proof that value iteration converges regardless of the initialisation revolves around the fact that the Bellman equation is a *contraction mapping*. Let $\mathcal{B}(\cdot)$ be the Bellman operator performing on a value function. We see from Equation (2) that for any initialisation V_0 of the value function:

$$\begin{gathered} V_{i+1} = \mathcal{B}(V_i) \ \forall i \in [0, \infty) \\ ||\mathcal{B}(V_{i+1}) - \mathcal{B}(V_i)||_\infty \leq \gamma ||V_{i+1} - V_i||_\infty \ \forall i \in [0, \infty) \end{gathered} \tag{7}$$

where $||f||_\infty$ is the supremum of f. Therefore, as $|\gamma| < 1$, the difference between successive values decreases with every iteration. Hence, after an infinite number of iterations:

$$\lim_{i \to \infty} ||\mathcal{B}(V_{i+1}) - \mathcal{B}(V_i)||_\infty = 0 \tag{8}$$

and the value is guaranteed to converge [2].

Policy Iteration Convergence:

The policy iteration algorithm gaurantees two things:

1. If for the ith iteration $\pi^{i+1} = \pi^i$ then π^i was an optimal policy
2. $V^{\pi^{i+1}}(s) \geq V^{\pi^i}(s) \ \forall s$ by Equation (2)

Between these two points, we are therefore guaranteed to converge to the optimal policies in a **finite** number of iterations.

This is also guaranteed to be the same or fewer iterations than value iteration, which theoretically takes an infinite number of iterations to converge (although the policy can converge much sooner without us realising as it is not explicitly calculated in the algorithm). This suggests policy iteration is more efficient computationally than value iteration [2] despite the higher algorithmic cost per iteration. However, as explained in Part A, in practice value iteration converges in a similar number of iteration when sensible thresholds are used, and so as the computational cost of each iteration is less for value iteration, it becomes practically more efficient.

6.2 Convergence of TD Methods Under Decaying α

As proven in [3], for any $\omega \in (0.5, 1)$, we can select a dynamic learning rate $\alpha_n = 1/n^\omega$ such that:

$$\begin{gathered} \sum_{n=0}^{\infty} \alpha_n = \sum_{n=0}^{\infty} \frac{1}{n^\omega} = \infty \\ \sum_{n=0}^{\infty} \alpha_n^2 = \sum_{n=0}^{\infty} \frac{1}{n^{2\omega}} = 0 \end{gathered} \tag{9}$$

where this is a consequence that the infinite sum of $1/n^k$ only for positive k diverges if k is between 0 and 1 (hence ω is restricted to be between 0.5 and 1 here so that the first sum diverges and the second converges above).

With this learning rate, and discounting factor $0 < \gamma < 1$, we are guaranteed to converge with a rate that is polynomial in $1/(1-\gamma)$, and exponential in $1/(1-\gamma)$ if $\omega = 1$ [3].

6.3 Part A Code

6.3.1 Asynchronous Value Iteration

```
def Asynchronous_value_iteration(model: Model, maxit: int = 100, threshold: float = 0.0001):

    ## Initialise:
    V = np.zeros((model.num_states))
    pi = np.zeros((model.num_states))

    ## Compute the value of being in the current state with the current action by taking
        expectation of value over next states that the action can take you to:
    def compute_value(s, a, reward: Callable):
        return np.sum([model.transition_probability(s, s_next, a) * (reward(s, a) + model.gamma
            * V[s_next]) for s_next in model.states])

    ## For each current state, find the value of each action using the compute_value, and
        greedily take the action with highest expected value:
    def update_values():
        for s in model.states:
            action_values = [compute_value(s, a, model.reward) for a in Actions]
            V[s] = np.max(action_values)

    for i in trange(maxit):
        V_old = V.copy()
        update_values()

        diff = max(abs(V_old-V))
        if diff < threshold:
            break

    return V
```

Listing 1: The code used for asynchronous value iteration. Note how the policy is never explicitly calculated (although it can easily be computed from the value function each iteration if required). Note also how in 'update_values()' the current value function is both used to compute the update and overwritten, rather than held out as in synchronous value iteration.

6.3.2 Synchronous Value Iteration

```
def SynchronousValueIteration(model: Model, maxit: int = 100, threshold: float = 0.0001):

    ## Initialise:
    V = np.zeros((model.num_states))
    pi = np.zeros((model.num_states))

    ## Compute the value of being in the current state with the current action by taking
        expectation of value over next states that the action can take you to:
    def compute_value(s, a, reward: Callable):
        return np.sum([model.transition_probability(s, s_next, a) * (reward(s, a) + model.gamma
            * V[s_next]) for s_next in model.states])

    ## For each current state, find the value of each action using the compute_value, and
        greedily take the action with highest expected value:
    def update_values():
        V_new = V.copy()
        for s in model.states:
            action_values = [compute_value(s, a, model.reward) for a in Actions]
            V_new[s] = np.max(action_values)
        return V_new

    for i in trange(maxit):
        V_new = update_values()
        diff = max(abs(V_new-V))
        V = V_new

        if diff < threshold:
            break

    return V
```

Listing 2: The only difference between this code and the code for Synchronous Value Iteration is in the ‘update_values()’ function and the loop over iterations at the end, where now the value update is done on held-out values for all states before being copied over to *value_new* at the end of the iteration.

7 Part B Code

7.0.1 SARSA

```python
def SARSA(model: Model, num_episodes: int = 100, maxit: int = 1000, alpha: float = 1, epsilon:
    float = 0.1, dynamic_epsilon: bool = False, dynamic_alpha: bool = True):

    ## Initialise Q-func values and policy:
    Q = np.zeros((len(model.states), len(Actions)))
    V = np.zeros((len(model.states),))
    pi = [Actions(0)] * len(model.states) # intialise policy actions from all states to 0
        arbitrarily

    def choose_action(s: int, epsilon):
        if np.random.uniform(0,1) < epsilon:
            return np.random.randint(Q.shape[1])
        else:
            return np.argmax(Q[s])

    def update_Q(s, a, r, s_next, a_next, alpha):
        Q[s][a] += alpha * (r + model.gamma*Q[s_next, a_next] - Q[s][a])

    def update_pi(s): # For the purpose of plotting
        optimal_action_index = np.argmax(Q[s, :])
        pi[s] = Actions(optimal_action_index)

    threshold = 0.0001
    for i in trange(num_episodes):

        pi_old = pi.copy()
        Q_old = Q.copy()
        episode_over = False
        s = model.start_state
        action = choose_action(s, epsilon=1) # first initialisation step arbitrarily
        j = 0

        while not episode_over and j <= maxit:
            if dynamic_epsilon:
                eps = 1 / (j+1)
            if dynamic_alpha:
                alpha = 1 / (j+1)

            immediate_reward = model.reward(s, action)
            cumulative_reward += immediate_reward

            # Move to next state by sampling s_next from P_Transition(s_next, s, a)
            s_next = np.random.choice(model.states[:-1], p=[model.transition_probability(s, s_,
                action) for s_ in model.states[:-1]])
            a_next = choose_action(s_next, epsilon=eps)

            update_Q(s, action, immediate_reward, s_next, a_next, alpha=alpha)
            update_pi(s)

            s = s_next
            action = a_next
            if s == model.goal_state:
                episode_over = True
                Q[s][action] += alpha*(model.reward(s, action) - Q[s][action])
                cumulative_reward += model.reward(s, action)

            j += 1

        if max(abs(Q_old - Q).flatten()) < threshold:
            break

     # Record the value of each state from the Q-function for the purpose of plotting:
     for s in model.states:
            V[s] = np.max([Q[s][a] for a in Actions])

     return pi, Q, V
```

Listing 3: Code snippet for the SARSA algorithm. Note that each iteration the algorithm looks forward one step to see the next state and action (chosen ϵ-greedily), and directly uses this pair to update the Q-function in ‘update_Q()’.

7.0.2 Expected SARSA

```
...

    def update_Q(s, a, r, s_next, alpha, epsilon):

        # Expected SARSA doesn't need the new action as it takes the expected q-function value
            over all actions given next state.
        # However, the next action is defined below for purposes of moving the exploration of
            the agent forward.

        expQ = 0
        maxQ = np.max(Q[s_next, :])

        chose_greedy = 0
        for i in range(len(Actions)):
            if Q[s_next][i] == maxQ:
                chose_greedy += 1

        prob_non_greedy = epsilon / len(Actions)
        prob_greedy = ((1 - epsilon) / chose_greedy) + prob_non_greedy

        for i in range(len(Actions)):
            if Q[s_next][i] == maxQ:
                expQ += Q[s_next][i] * prob_greedy
            else:
                expQ += Q[s_next][i] * prob_non_greedy

        Q[s][a] += alpha * (r + model.gamma*expQ - Q[s][a])

    ...
```

Listing 4: Code for the Expected SARSA algorithm. The only difference between the code above for SARSA and the code for Expected SARSA is how to handle the expectation value. This is done in ‘update_Q()’.

7.1 Part C Code

```
...

   def update_Q(s, a, r, s_next, alpha):
          # Q-Learning always takes the greedy approach for a_next when judging current state
              value, and therefore this doesn't need the a_next chosen below in the loop:
          Q[s][a] += alpha * (r + model.gamma*max(Q[s_next, :]) - Q[s][a])

...
```

Listing 5: Code for the Q-learning algorithm. Again the only difference to SARSA is in the 'update_Q()' function, which is now changed to update the target policy with the greedy action selection independent of the behaviour policy, via: