COM3240 Adaptive Intelligence: Robot Navigation of GridWorld using Reinforcement Learning

Luca Del Basso

***Abstract*— This report discusses an implementation of the SARSA and SARSA(λ) ,on policy, reinforcement learning algorithms. The algorithms are analysed on a discrete state GridWorld problem in which an agent converges on an estimated optimal path to a singular reward. ϵ-greedy and softmax were experimented with as action-selection policies and the sigmoid function for an activation function. The appendix contains code snippets.**

I. INTRODUCTION

Reinforcement Learning is a sub-category of machine learning, in which the AI or *agent* makes predictions and performs actions in return for immediate or future *rewards*. Upon receiving a reward, or by reaching some terminal *state*, the agent updates its estimations that aid in the prediction of future rewards whilst in a specific or similar state. Through a process of trial and error, the agent is not told which action to take but discovers which actions return the most reward.

A. Environment Description

In this task, an agent is in a $N \times N$ grid world and has four actions: NORTH, EAST, SOUTH, WEST. The agent transitions from state to state by taking actions. The SARSA algorithm is such that the agent receives a reward upon moving to a new state. This reward is used to adjust the agents prediction in regards to its *expected* reward for that transition, compared to what it actually received for that transition. In this environment, there is only one state in which a positive reward is located. The agent updates its estimates for its current state-action pair by considering the expected reward for the next state-action it intends to take.

B. SARSA - Update rule

In this assignment, a single layer perceptron (SLP) is used to map an encoded form of the current state to an action. For an environment size of 10×10 states, with 4 actions, there are a total of 400 state-action pairs which will be represented by a weight matrix of the SLP. excluding for the terminal state, the update rule for the SARSA algorithm weight matrix is as follows:

$$Q(s,a) \leftarrow Q(s,a) + \eta[R + \gamma Q(s',a') - Q(s,a)]$$

where:

- $Q(s,a)$ represents the weight joining the pre-synaptic encoded state, s, to our post-synaptic output a. This value denotes the predicted or *expected* reward for taking action a in state s. Initialized randomly such that $0 \leq Q(s,a) < 1$ for all $s \in S, a \in A(s)$ and $Q(terminal-state, \cdot) = 0$
- η is the network's *learning rate* $0 \leq \eta \leq 1$. A higher learning rate makes the agent consider the most recent information more.
- R is the reward received when transitioning from state s to s' through action a.
- γ is the *discount factor* , $0 \leq \gamma < 1$ which determines the significance of future rewards. A high discount factor makes the agent consider future rewards more. This is important for this task the environment contains only 1 state in which a positive reward is received. This allows for stronger expected reward signals to sort of propagate from and around the positive terminal state and will provide a stronger incentive for the agent to transition to states closer to the reward.
- $Q(s',a')$ is the next state-action pair taken from s. For example if $Q(s',a')$ is the terminal state and it has been visited once before on the previous trial, then the neighbouring states of the terminal (including $Q(s,a)$) will receive a stronger expected reward signal and will increase the probability of the agent taking these steps in later trials.

The Update rule, when in the terminal state, is

$$(Q(s,a) \leftarrow Q(s,a) + \eta[R - Q(s,a)]$$

C. SARSA(λ) - Update Rule

As seen in the update rule, The SARSA algorithm updates the previous state-action pair value depending on the current state-action. This means only one q-value in the SLP weight matrix is updated per time step and the reward information propagates by an order of $N_{states} \times N_{actions}$. SARSA($\lambda$) introduces an eligibility trace - denoted by $e(s,a)$ - that tracks the trajectory (the states previously visited during the current trial) and updates the respective state-action pairs of the whole trajectory depending on the reward received on each time step. The update function is given by:

$$\delta \leftarrow R + \gamma Q(s',a') - Q(s,a)$$

$$e(s,a) \leftarrow e(s,a) + 1$$

$$Q(s,a) \leftarrow Q(s,a) + \eta\delta e(s,a)$$

$$e(s,a) \leftarrow \gamma\lambda e(s,a)$$

To record the eligibility trace, the SLP stores a new matrix of identical size to the weight matrix (which is storing all the state-action pair values), initialised as equal to zero. When action a is taken from state s, the corresponding location of that state-action pair in the e matrix is incremented by 1. Then, the entire weight matrix is incremented by a scaled value of the e matrix and finally the e matrix scaled by the discount factor, γ, seen previously, and a new parameter,λ, that controls the rate of decay of the trace. $0 \leq \lambda \leq 1$.

II. Action Selection Policy

A. ϵ-greedy

In reinforcement learning a balance that has to be found is between exploration and exploitation. When is it better to explore in order to potentially discover higher rewards, compared to when is it better to exploit actions of (somewhat) known rewards. One of the simplest action selection policy (ASP) is ϵ-greedy. Where ϵ is a parameter and bounded by $0 \leq \epsilon \leq 1$. The action that is expected to return the highest reward is chosen with probability $1 - \epsilon$, an action is chosen at random with probability.

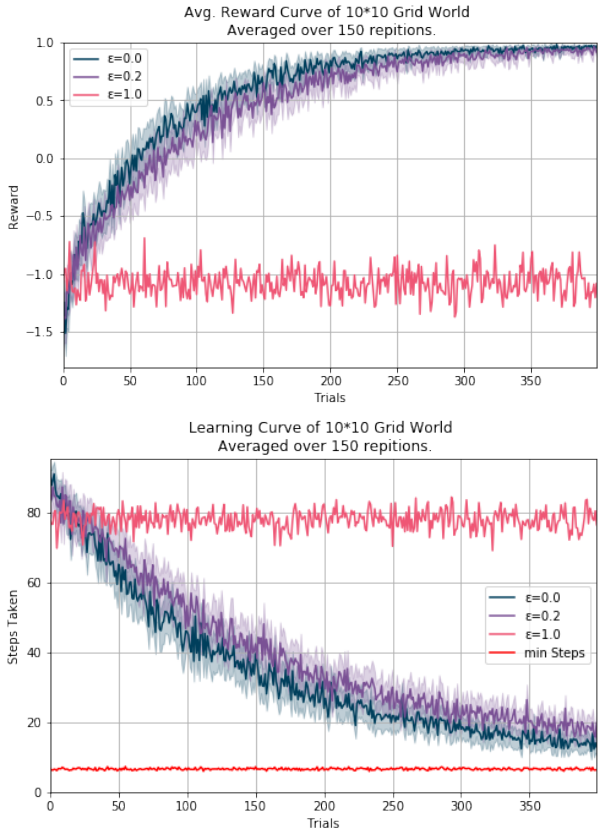


Fig. 1. convergence on maximum reward and minimum step for ϵ-greedy. parameters: η: 0.3 , γ:0.9.

ϵ-greedy as an ASP, converges well and is a popular choice. In its simplicity, it does not consider that in a lot of environments exploration would be favourable initially but not towards the end of the trial. It can often be assumed that a lot of the q-values have been converged upon to some degree and exploration may not be needed as much. As a result exploitation should take precedence later on.

In this narrow test case, $\epsilon = 0$ converges faster and generally performs better than the others, apart from taking slightly more steps initially to find the terminal positive reward state. This is because it will only select the terminal state when it has the highest $Q(s, a)$ value whereas the other agent has a 20% probability of selecting randomly.As ϵ increases you can see the agent explores more because the average steps for each agent increases as ϵ increases, with ϵ = 1 showing a completely random policy. Notice that the error range does not decrease for the average steps taken, this is because the likelihood of exploration remains constant throughout the entire experiment for each agent. Whereas the errorbars start to converge for the average reward, this is because the agent has learnt to not turn into walls (which return a reward of -0.1) and does not run out of steps (which returns a reward of -1) thus the reward range narrows to 1 as this is the only reward the agent receives. Again, greedy performs better than ϵ=0.2 because the latter will always have a 20% chance of turning into a wall, even if its expected reward for those $Q(s, a)$ pairs are significantly worse than the possible $max(Q(s, a)$.

B. Softmax - Boltzmann Distribution

I also investigated the Softmax function, using a Boltzmann distribution, as a potential ASP. Where:

$$Pr\{a_t = a|s_t = s\} = \frac{e^{\frac{Q(s,a)}{\tau}}}{\sum_b e^{\frac{Q(s,b)}{\tau}}}$$

The function takes the $Q(s, a)$ values and turns them into a probability distribution that sums to one. The action that is taken is then drawn from that probability distribution. It is controlled by a parameter, τ, that determines how distributed the probabilities are. As $\tau \to 0$ the policy becomes greedy, pushing the $max(Q(s, a))$ value to have a probability that essentially becomes 1. On the otherhand, as $\tau \to 1$, the policy more evenly distributes the probabilities, becoming a random selection. τ is implemented in such a way that it initially is close to 1, meaning the policy is explorative, and as the trial progresses, τ approaches zero meaning it becomes a greedy policy. This was achieved by, $\tau \leftarrow \frac{1}{t} \times c$. Where t is the current trial number and c is a scaling factor to control the rate in which the agent takes greedier actions.

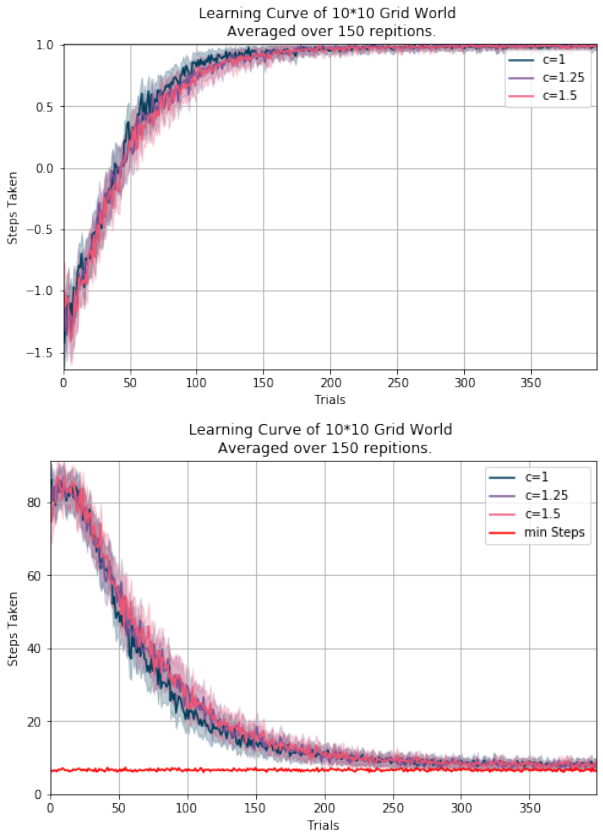


Fig. 2. convergence on maximum reward and minimum step for softmax action selection. parameters: η: 0.3 , γ:0.9. (see appendix for more)

Compared to ϵ-greedy, softmax as an ASP converges on its estimate of the optimal policy much quicker. It initially sees an

increase in steps taken early in the runs because τ is closer to 1 meaning the agent initially favoured an exploraitve approach instead of exploitative. The c parameter was included to investigate if the explorative behaviour could be significantly increased/decreased. The method clipped τ between 0 and 1 in hopes that it would remain at 1 for longer and thus explore for longer. See the appendix to more clearly see the error differences. Notice how the error range also converges, as the probability for greedy selection increases as the trial number increases. The results show that varying c does not impact the results that much, with $c = 1$ performing the best. Whilst softmax takes longer computationally, it converges much quicker than ϵ-greedy. As a result, softmax is used as the ASP for all future experiments. See the Appendix for separate plots of different parameter tests thus far.
For the rest of the report, all results shown will be using a softmax ASP,with $c = 1$. Whilst Softmax takes longer to execute per time step, it also converges quicker and closer to the minimum required steps per trial.

III. Finding Optimal Values

A. Action Selection Policy

As seen in the previous chapter the parameter controlling softmax, τ, changes on every trial and adding a scaling parameter $c \geq 1$ did not seem to have much affect. I experimented with values for c less than one (see appendix) but saw no real change in results.

B. Learning Rate

The learning rate is a factor that determines how much of the most recent information is considered when updating q-values. Here are the results.

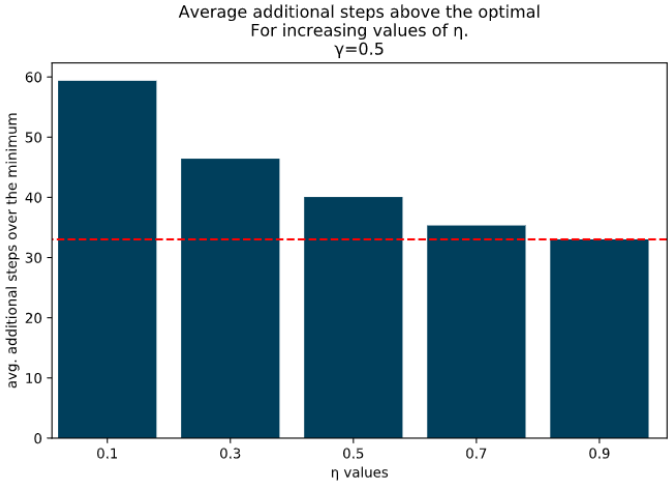
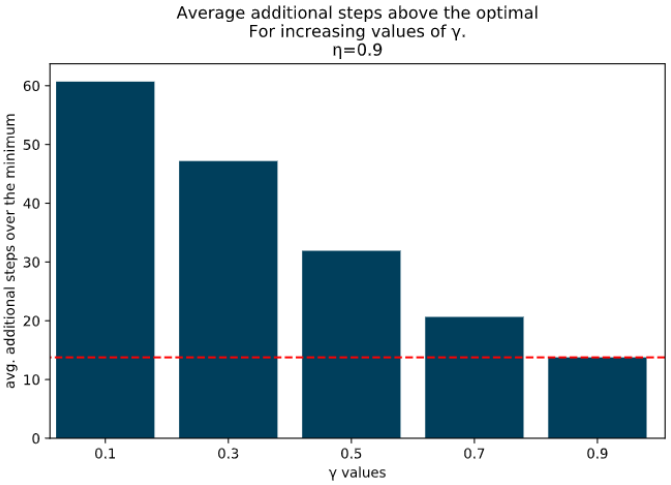


Fig. 3. averaged over 10 runs

The figure shows that as η increases, the average number of additional steps above the optimal path decreases. See the appendix for learning curves, they show that the time it takes to converge generally decreases as η increases. The optimal value selected for η will be 0.9. as it produced the lowest amount of additional steps needed. Having η set to 0.1 shows that a learning rate that is too close to zero means the agent will learn at a very slow rate, shown by the much slower convergence on the optimum (see appendix). I have used an average of the optimal steps for each trial as a measurement of success. It shows how each agent converges but due to parameter tweaks, some converge faster than others and closer to the optimal.

C. Discount Factor

The discount factor determines how much the agent considers future reward. In this environment, as there is only one reward location and $N_{states} \times N_{actions}$ state-action pairs it is clear that having a high discount factor (and thus really taking into consideration future rewards) has a significant impact on the success of the agent (see appendix).



The data shows that as the discount factor approaches one, the success and speed of convergence of the agent increases. As a result, a discount factor of 0.9 will be used for all future experiments.

IV. SARSA(λ)

Implementing an eligibility trace allows the agent to strengthen its trajectory for the current trial. Prior to this, the agent only updated a single state-action pair on every time step. With an eligibility trace, the agent can update the most recent state-action pair with the highest weighting (as it gets incremented by 1 and applied to the weight matrix *before* being scaled down by parameters) and then all previous state-action pairs along that trajectory get updated by a smaller and smaller amount depending on how close they are to the current, in terms of time steps.

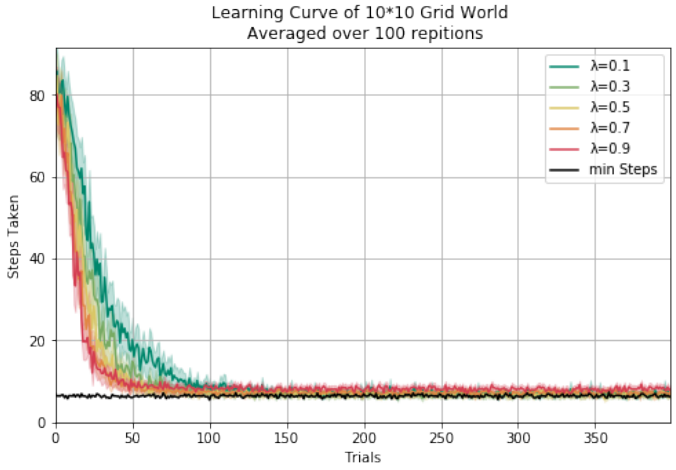


Fig. 4. convergence on minimum step for softmax action selection, measuring a change in the eligibility trace parameter λ. η:0.9 , γ:0.9.

Figure 5 shows the agents converge quicker with an eligibility trace, than without it. and a λ value of 0.9 converges the fastest, at approximately the 50th trial. This is because for a bigger λ, all previous states on the trajectory get updated by a larger value.

V. Q-Value Heat Map

A common way to show information regarding the weights/$Q(s, a)$ values for discrete and relatively small reinforcement learning problems is to use a heatmap. The heatmaps below plot the $max_a(Q(s, a))$ and the direction that it points to.

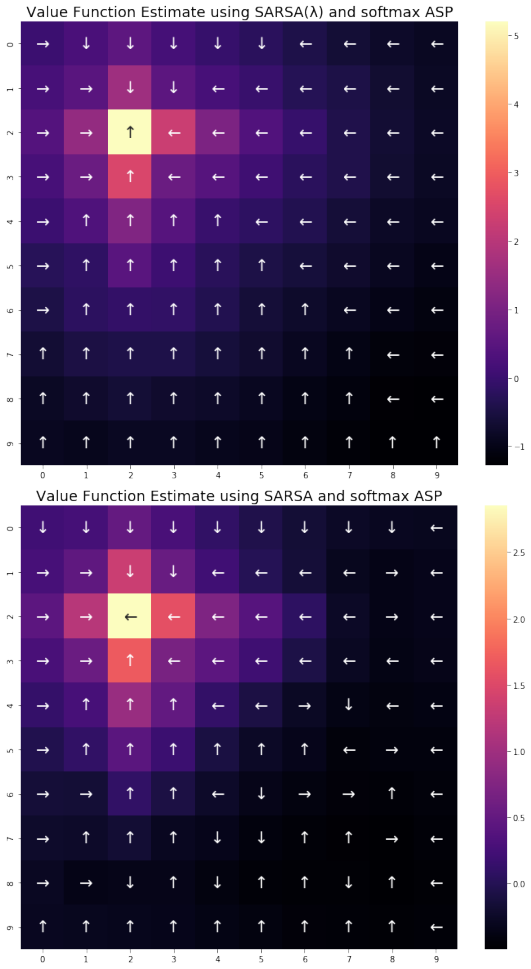


Fig. 5. Heatmaps showing the agents estimate for an optimal policy. The colour indicates the strength of the expected highest reward whilst the arrow indicates what action will lead to this reward. λ :0.9. (only applicable for the top) η:0.9 , γ:0.9. Averaged over 100 runs and using the same terminal state.

The top heatmap reinforces the concept of the eligibility trace, as the expected rewards are more concentrated around the terminal state and with a stronger correlation. Whereas without an eligibility trace, the values are updated still with a correlation pointing towards the reward, but a weaker one.

VI. Increasing the Search Space

As the area of the search space increases, so does the combination of state-action pairs. An area of size 1000×1000 would require the SLP to have a weight matrix with 4 rows and 1,000,000 columns. The agent would need to spend significantly more time steps per trial exploring the environment before finding the 1 state with a positive reward. Methods to increase convergence time can be implemented, such as the softmax boltzmann method or eligibility trace as seen in this report. More advanced approaches such as using a Bayesian Neural Network that uses a probability distribution over possible weights, instead of having a single set of fixed weights.

VII. Extra Walls

As described in the assignment specification, walls were added at the described locations that return a reward of -0.1 if the agent turns into it, the agent is also returned to its original location if this happens. Likewise for if the agent tries to leave the environment. These heatmaps show that the agent can navigate the space and avoide negative impacting obstacles, with SARSA(λ) providing stronger correlating directions towards the terminal positive reward state.

Interestingly, the maximum expected values increased significantly when obstacles were implemented, even more so when using an eligibility trace, as seen by the key on the right of the maps. Overflow warnings occurred when the trial number grew larger and thus when the agent became greedier. I suspect that as the location of the reward is in a corner, once the agent becomes greedy the update rule can make the weight matrix grow unbounded. I experimented with ensuring that the agent does not initialise at the terminal state and this achieved faster convergence but still fairly high (yet lower than previous) q-values. See appendix.

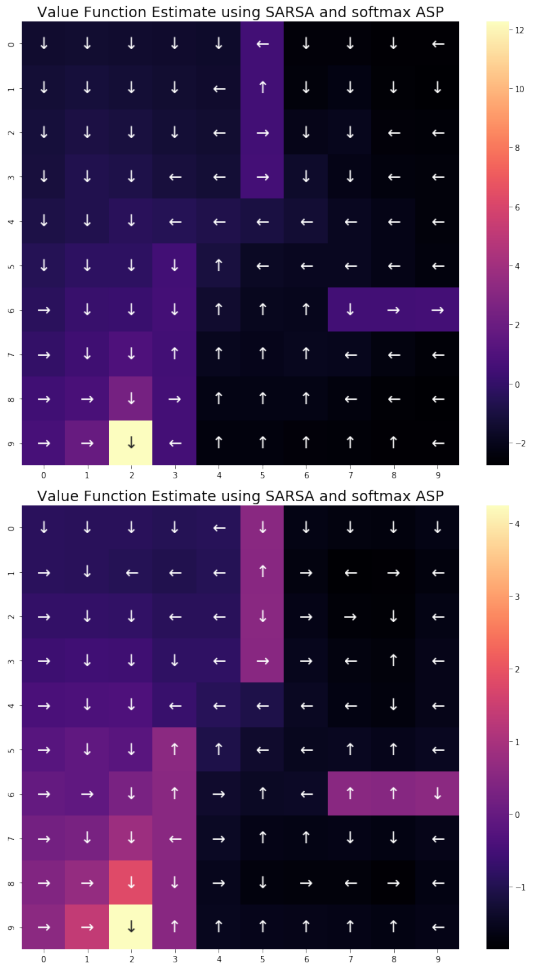


Fig. 6. Top figure is SARSA(λ).Heatmaps showing the agents estimate for an optimal policy when walls are introduced inside the environment. The colour indicates the strength of the expected highest reward whilst the arrow indicates what action will lead to this reward. λ :0.9. (only applicable for the top) η:0.7 , γ:0.9. Averaged over 100 runs and using the same terminal state.

Appendices

A. Action Selection Policy

```python
import numpy as np
import matplotlib.pyplot as plt
import time

def safe_sigmoid(signal):
    '''a sigmoid function that prevents overflow or underflow'''
    sig = signal.copy()
    sig = np.clip( sig, -500, 500)
    return 1.0/(1+np.exp(-sig))

def epsilon_greedy(signal,eps):
    '''classic epsilon greedy
       inputs:
            signal: the array to be chosen from
            eps: the epsilon value '''
    p = np.random.rand()
    if p < eps:
        return np.random.randint(0,len(signal))
    else:
        return np.argmax(signal)

def softmax(qs,ts,c):
    '''a softmax function turning q values into probability
       distributions
       inputs:
            qs: the q(s,a) valeus
            ts: the current trial iteration
            c: a scaling factor used for experimentation -> default is 1'''
    q=qs.copy()
    temp = 1/ts * c
    temp = np.clip(temp,0,1)
    e = np.exp((q-np.max(q))/temp)
    if np.isnan((np.sum(e))):
        print("hi")
    k = e /np.sum(e)
    return k
```

Fig. 7. The code for the action selection policies.

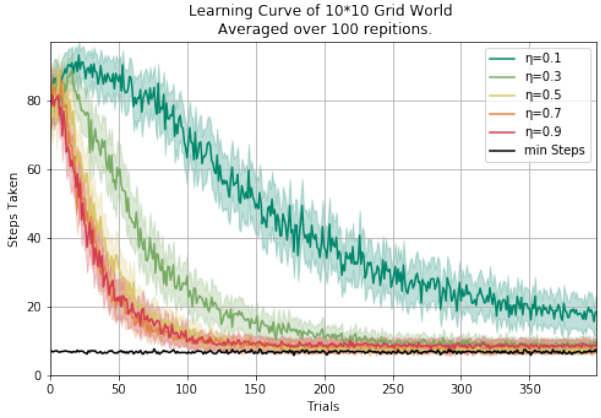


Fig. 8. convergence on minimum step for softmax action selection, measuring a change in the learning rate. γ:0.9.

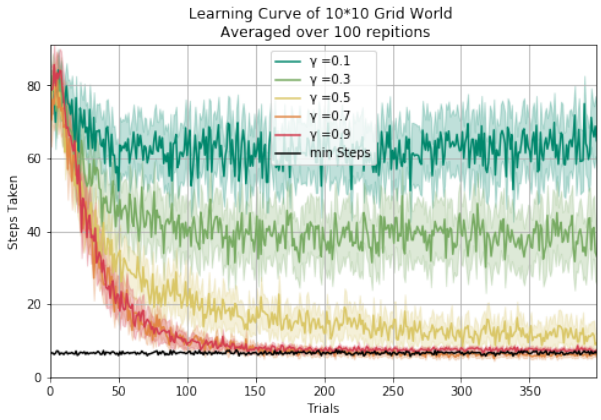


Fig. 9. convergence on minimum step for softmax action selection, measuring a change in the discount factor. η:0.9.

Here is the code used for my action selection policy experiments. In some cases I received overflow errors so as a result I implemented the clip code (in safe sigmoid) to prevent this. I am not sure if this is considered a 'quick fix' or a genuine solution to similar problems in reinforcement learning.

B. softmax with wall

The graphs on the right shows a delay in convergance when obstances are introduced that provide a negative reward. As expected SARSA(λ) performs better as it can update its whole trajectory.

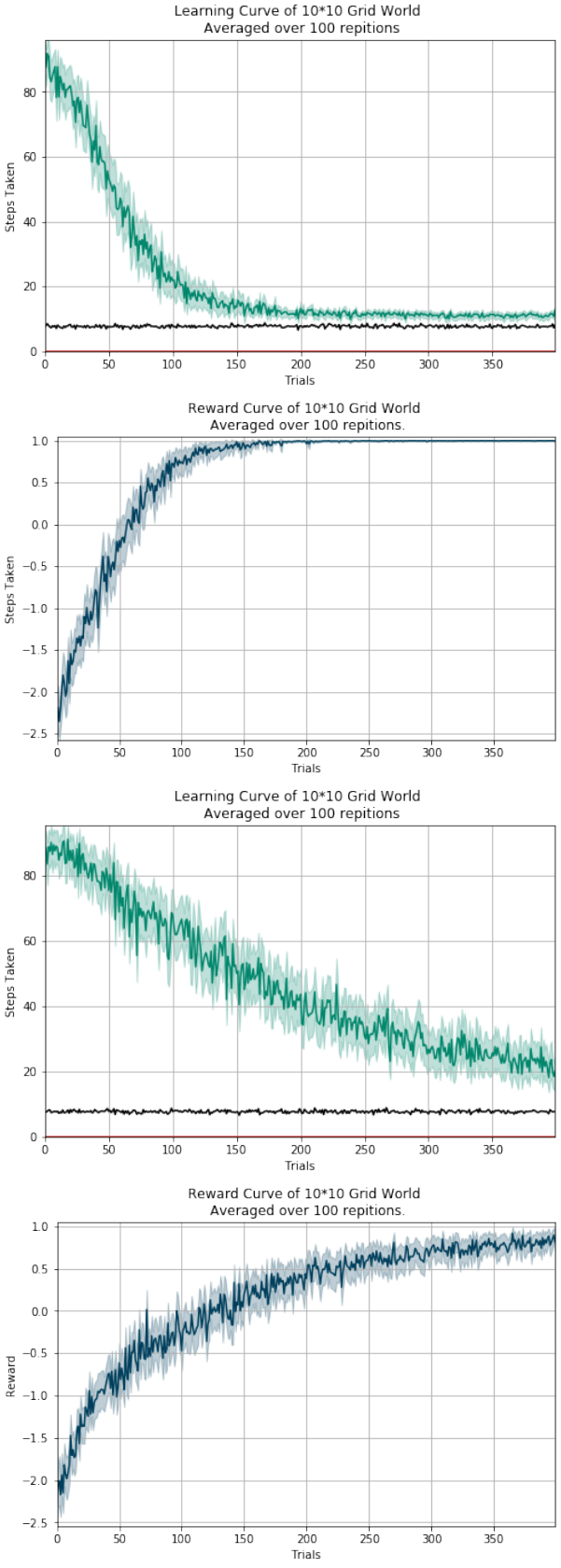


Fig. 10. top two: learning curve/reward curve with obstacles for SARSA(λ). Using optimal values previously found. bottom two: learning curve/reward curve with obstacles for regular SARSA. Using optimal values previously found.

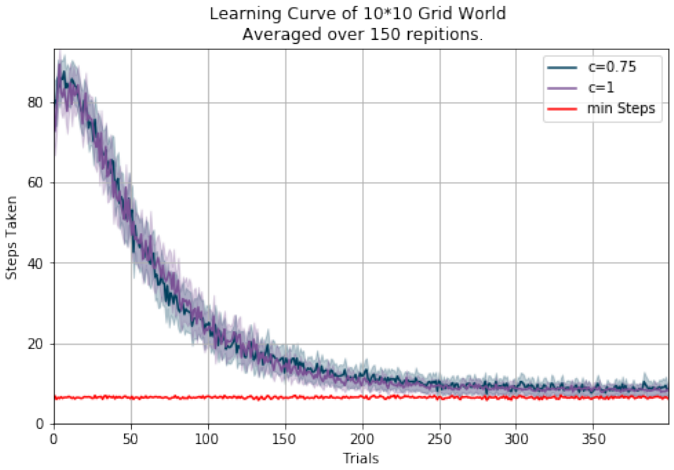


Fig. 11. As you can see there is not much variation when varying the parameter c

C. *softmax supplementary*

Below you can see more detailed plots of the softmax experimentation. The top one consists of experimenting with $c \leq 1$. The below plots are the same data as the one in the main report but split onto their own axis for easier viewing.