Journal of Data Analysis and Information Processing, 2016, 4, 159-176
http://www.scirp.org/journal/jdaip
ISSN Online: 2327-7203
ISSN Print: 2327-7211

Double Sarsa and Double Expected Sarsa with Shallow and Deep Learning

Michael Ganger, Ethan Duryea, Wei Hu

Computer Science Department, Houghton College, Houghton, NY, USA
Email: michael.ganger17@houghton.edu, ethan.duryea18@houghton.edu, wei.hu@houghton.edu

How to cite this paper: Ganger, M., Duryea, E. and Hu, W. (2016) Double Sarsa and Double Expected Sarsa with Shallow and Deep Learning. *Journal of Data Analysis and Information Processing*, **4**, 159-176.
http://dx.doi.org/10.4236/jdaip.2016.44014

Received: July 26, 2016
Accepted: October 14, 2016
Published: October 17, 2016

Copyright © 2016 by authors and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/
Open Access

Abstract

Double Q-learning has been shown to be effective in reinforcement learning scenarios when the reward system is stochastic. We apply the idea of double learning that this algorithm uses to Sarsa and Expected Sarsa, producing two new algorithms called Double Sarsa and Double Expected Sarsa that are shown to be more robust than their single counterparts when rewards are stochastic. We find that these algorithms add a significant amount of stability in the learning process at only a minor computational cost, which leads to higher returns when using an on-policy algorithm. We then use shallow and deep neural networks to approximate the action-value, and show that Double Sarsa and Double Expected Sarsa are much more stable after convergence and can collect larger rewards than the single versions.

Keywords

Double Sarsa, Double Expected Sarsa, Reinforcement Learning, Deep Learning

1. Introduction

Reinforcement learning is concerned with finding optimal solutions to the class of problems that can be described as agent-environment interactions. The *agent* explores and takes actions in an *environment*, which gives the agent a reward, r, for each state, s', into which the agent transitions as a result of taking action a from the initial state s. The goal is to find an optimal policy $\pi^*(a|s)$ that maximizes the expected reward collected by the agent [1]. Often, this is described as a Markov Decision Process (MDP), which groups this sequence into an experience: (s, a, r, s'). In an MDP, the state s fully describes the environment, meaning that no other information is required to choose the next action; in other words, information from all previous states that can affect all future states is expressed in s.

There are multiple approaches to finding the optimal policy $\pi^*(a|s)$, which gives the probability of taking an action a given a state s. One set of techniques, known as *Policy Gradient* methods, directly search the space of available policies for one that maximizes the accumulated discounted reward per episode, $g = \sum_{t=0}^{T} \gamma^{T-t} r_t$, where γ is the discount rate [2]. Another set of approaches, called *Temporal Difference* (TD) methods [3], estimate the value of a particular state, $V(s)$, or state-action pair, $Q(s,a)$, and use these values to derive a policy $\pi(a|s)$ that maximizes these value functions at each step instead of maximizing g. There are other techniques that combine the ideas of Policy Gradient and Temporal Difference methods, most notably a class of algorithms called *Actor Critic* [4] [5], but in this paper we only consider algorithms that fall under the Temporal Difference category. Within this category, there are two main types of algorithms: on-policy and off-policy [4]. With off-policy algorithms, the target policy being learned is different from the behavior policy, which is the policy that the agent uses to explore the environment. For example, the behavior policy might be to choose completely random actions, while the target policy might be to always take the action with the largest expected return. In contrast to off-policy algorithms, the target and behavior policies are the same with on-policy algorithms.

One of the most popular Temporal Difference algorithms is Q-learning, first proposed in [6]. Q-learning is an off-policy algorithm that learns the greedy action-value $Q^*(s,a)$ by updating the estimate $Q(s,a)$ at every step. Although it is guaranteed to converge when the environment and rewards are deterministic, it is less robust in scenarios where these are stochastic. An extension of the Q-learning algorithm, called Double Q-learning [7], uses two action-value estimates $Q^A(s,a)$ and $Q^B(s,a)$, improving the performance of Q-learning in these stochastic scenarios. Generally, the average of $Q^A(s,a)$ and $Q^B(s,a)$ tends to be below the estimate that Q-learning makes, and is sometimes below $Q^*(s,a)$.

However, in many scenarios an off-policy algorithm is not realistic as it does not account for possible rewards and penalties that might result from an exploratory behavior policy. For example, receiving immediate returns might be more important than a true optimal policy, and while an on-policy algorithm may not converge to the optimal policy, it may still converge in fewer time steps than an off-policy algorithm to a policy which may be considered "sufficient" according to the problem domain. Often, these on-policy algorithms have stochastic policies that encourage exploration of the environment, which can also be a beneficial quality when the environment is subject to change. One such policy is called ϵ-greedy [6], which uses the parameter ϵ to control the probability that the optimal action will be taken over a random one.

A simple on-policy algorithm that is similar to Q-learning is called Sarsa [8]. Like Q-learning, it learns the action-values at each step, but unlike Q-learning, it depends solely on the states visited and actions taken. Because of this, Sarsa's action-value estimate $Q(s,a)$ never converges when the learning rate α is constant, although for a sufficiently small α, the policy can converge to one that balances exploration and exploitation. For example, if an ϵ-greedy policy is used, the policy that Sarsa converges

to will avoid states that are adjacent to other states with a large negative reward. In other words, the policy will account for the possibility of random actions and take a path which figuratively does not come "too close to the edge". A similar on-policy algorithm is called Expected Sarsa [9]; like Sarsa, this algorithm converges to a policy that balances exploration and exploitation. However, unlike Sarsa, the action-value estimate also converges, which allowing for much higher learning rates to be utilized. Notably, the policies of both of these algorithms can only converge if the reward is deterministic; if it is stochastic, the policies are much less likely to converge (unless a sufficiently small learning rate is used).

The algorithms presented above use a tabular format to store the action-values, *i.e.* there is a single entry in the table for every s, a pair; as such, they are limited to simple problems where the state-action space is small. For many real problems, this is not the case, especially when the state-space is continuous; function approximation must be used instead. In application to Temporal Difference algorithms, it is the value functions that are approximated [10], and a variety of techniques from supervised learning are used. A more recent development is the application of deep learning [11] to Q-learning, termed a Deep Q-Network [12]. Deep learning function approximation is the term given to neural networks with many layers, and has been shown to be effective in reinforcement learning problems with large state-action spaces, such as those encountered in Atari games. Double learning has also been applied to Deep Q-Networks, which is referred to as Deep Double Q-learning [13], and has shown success in the same domain. However, recent work has shown that shallow networks can achieve similar results [14], so the advantage of deep learning over shallow learning appears to be highly domain-dependent. Additionally, deep learning has been applied to Actor Critic methods, combining Deep Q-networks with recent development in deterministic policy gradients [15] to produce a robust learning algorithm [16].

The current state-of-the-art in reinforcement learning can be seen in [17], which combined many techniques in order to learn the game of Go. This study used supervised learning to initialize a policy network, and then improved this network through self-play and generated new data. This data was then used to train a value network. During game play, a Monte Carlo Tree Search algorithm was used to simulate future moves and choose the best action. This efficient use of data to solve a problem with about 2.08×10^{170} states [18] represents a significant achievement in the field of reinforcement learning, and shows the power of combining multiple different techniques. The study used a combination of supervised learning and reinforcement learning to train both a policy and a value network, and combined real data with simulated data to improve their training. Additionally, although during training many CPUs and GPUs were used, the final rollout action selection was very efficient and ran on a single machine in a short period of time.

2. Algorithms

In this paper, we present two new algorithms that extend from the Sarsa and Expected

Sarsa algorithms, which we refer to as Double Sarsa and Double Expected Sarsa. The concept of doubling the algorithms comes from Double Q-learning, where two estimates of the action-value $Q(s,a)$ are decoupled and updated against each other in order to improve the rate of learning in an environment with a stochastic reward system. Although Q-learning and Double Q-learning are off-policy, this concept extends naturally to the on-policy algorithms of Sarsa and Expected Sarsa, producing a variation of each algorithm that is less susceptible to variations in the reward system. In addition, the ideas of Double Sarsa and Double Expected Sarsa can be extended with function approximation of the action-values, in the same way that Q-learning can be extended to Deep Q-networks through function approximation.

2.1. Double Sarsa

The update rule for Double Q-learning is what makes it unique from standard Q-learning. In Q-learning, the action-value is updated according to

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\left[r + \gamma \max_{a'} Q_t(s',a') - Q_t(s,a)\right], \tag{1}$$

where s is the initial state, a is the action taken from that state, r is the reward observed from taking action a, and s' is the next state the agent reaches resulting from s, a. In Double Q-learning, the update is decoupled using two tables, Q^A and Q^B:

$$Q^A_{t+1}(s,a) = Q^A_t(s,a) + \alpha\left[r + \gamma Q^B_t\left(s', \operatorname*{argmax}_{a'} Q^A_t(s',a')\right) - Q^A_t(s,a)\right]. \tag{2}$$

The key idea is the replacement of the maximum action-value, $\max_{a'} Q(s',a') = Q\left(s', \arg\max_{a'} Q(s',a')\right)$, with the value in a second table. This serves to decouple the two tables, tending to reduce susceptibility to random variation in r and stabilize the action-values. Additionally, the roles of the two tables Q^A and Q^B are periodically switched, meaning that each table is only updated using half of all the experiences and that there is only a marginal increase in computational cost over having a single table.

The update rule for Double Sarsa is very similar to that used for Double Q-learning. However, because it is on-policy, a few modifications are necessary. First, we use an ϵ-greedy policy that uses the average of the two tables to determine the greedy action,

$$\pi(a \mid s) = \begin{cases} 1-\epsilon, & \text{if } a = \arg\max_{a'}\left(Q^A(s,a') + Q^B(s,a')\right) \\ \dfrac{\epsilon}{N_a - 1}, & \text{otherwise} \end{cases}, \tag{3}$$

where $\pi(a \mid s)$ is the probability of taking action a from state s, and N_a is the number of actions that can be taken from state s. In general, any policy derived from the average of Q^A and Q^B can be used, such as ϵ-greedy or softmax [19]. The update rule then becomes

$$Q^A_{t+1}(s,a) = Q^A_t(s,a) + \alpha\left[r + \gamma Q^B_t(s',a') - Q^A_t(s,a)\right] \tag{4}$$

Because Sarsa does not take the maximum action value during the update rule, but

does so instead during the computation of the greedy policy, there is a weaker decoupling of the two tables. However, Q^A is still updated using the value from Q^B for the state-action pair s', a', which helps to reduce the variation in the action-value.

Figure 1 shows the Algorithm for Double Sarsa, using a generic policy π that balances exploration and exploitation. Unlike Double Q-learning, where the algorithm updates Q^A or Q^B with equal probability, in Double Sarsa Q^A and Q^B are instead *swapped* with equal probability to simplify implementation. This algorithm is very similar to the original Sarsa algorithm [4], except for the addition of the second action-value table and the swapping of the two tables.

2.2. Double Expected Sarsa

Expected Sarsa is a more recently developed algorithm that improves on the on-policy nature of Sarsa. Because Sarsa has an update rule that requires the next action a', it cannot converge unless the learning rate is reduced ($\alpha \to 0$) or exploration is annealed ($\epsilon \to 0$), as a' always has a degree of randomness. Expected Sarsa changes this with an update rule that takes the expected action-value instead of the action-value of s', a':

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\left[r + \gamma \sum_{a'} \pi(a' \mid s') Q_t(s',a') - Q_t(s,a)\right]. \tag{5}$$

Because the update no longer depends on the next action taken, but instead depends on the expected action-value, Expected Sarsa can indeed converge; [9] notes that for the case of a greedy policy π, Expected Sarsa is the same as Q-learning. In order adapt this to Double Expected Sarsa, we change the summation to be over Q^B instead of Q^A:

$$Q^A_{t+1}(s,a) = Q^A_t(s,a) + \alpha\left[r + \gamma \sum_{a'} \pi(a' \mid s') Q^B_t(s',a') - Q_t(s,a)\right]. \tag{6}$$

Although Expected Sarsa can be both on-policy and off-policy, here we discuss only

Double Sarsa

1. Initialize $Q^A(s,a)$ and $Q^B(s,a)$ arbitrarily
2. **loop** { over episodes }
3. Initialize s
4. Choose a from s using arbitrary policy
5. **repeat** { for each step of the episode }
6. Take action a, observe r, s'
7. Choose a' from s' using policy π derived from average of Q^A and Q^B
8. $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha\left[r + \gamma Q^B(s',a') - Q^A(s,a)\right]$
9. $s \leftarrow s'; a \leftarrow a'$
10. **with** probabilty 0.5:
11. swap$\left(Q^A, Q^B\right)$

Figure 1. Double Sarsa algorithm, with tabular representation of the action-values. Lines 10 and 11 swap the references to Q^A and Q^B, meaning each table is updated using half of the experiences each. Note that $\gamma = 0$ if the next state s' is terminal, otherwise it is the discount rate.

the on-policy version as it often has more utility; in Expected Sarsa, $Q(s,a)$ represents the estimated action-value under target policy π, which is the same as the behavior policy when it is on-policy. If the behavior policy and target policy are different (*i.e.* it is off-policy), it is usually more desirable for the target policy to be greedy, and not stochastic, in which case Expected Sarsa degenerates to Q-learning. The on-policy Double Expected Sarsa algorithm is shown in Figure 2, with lines 8 and 9 being the only differences from Double Sarsa. The two tables are again decoupled, this time in calculating the expected value $V_{s'}^{B}$ under the current policy. Although the action a' is chosen in line 7, it is not needed until the next iteration (it is shown as such in order to be consistent with the Double Sarsa algorithm in Figure 1).

2.3. Neural Network Approximation of *Q*(*s*, *a*)

Often, it is advantageous to represent the action-value function $Q(s,a)$ with a form of function approximation, especially when the state space is large or continuous. The simplest representation is a linear combination of the state-action features, $\phi(s,a)$, using a vector of weights, θ. In other words,

$$Q(s,a;\theta)=\sum_i \theta_i \phi_i(s,a). \tag{7}$$

If $\phi(s,a)$ is a one-hot encoding for each s,a pair, this degenerates to the tabular form discussed above. However, it is often beneficial to introduce non-linearities into the function approximator; one set of functions that do so are known as neural networks. The action-value function can be written more generally to accommodate this change of form:

$$Q(s,a;\theta)=f_a(\phi(s),\theta), \tag{8}$$

Double Expected Sarsa

1. Initialize $Q^A(s,a)$ and $Q^B(s,a)$ arbitrarily
2. **loop**{over episodes}
3. Initialize s
4. Choose a from s using arbitrary policy
5. **repeat**{for each step of the episode}
6. Take action a, observe r,s'
7. Choose a' from s' using policy π derived from average of Q^A and Q^B
8. $V_{s'}^{B}=\sum_{b'}\pi(b'\mid s')Q^B(s',b')$
9. $Q^A(s,a)\leftarrow Q^A(s,a)+\alpha\left[r+\gamma V_{s'}-Q^A(s,a)\right]$
10. $s\leftarrow s';a\leftarrow a'$
11. **with** probabilty 0.5:
12. swap(Q^A,Q^B)

Figure 2. Double Expected Sarsa algorithm, with tabular representation of the action-values. Lines 11 and 12 swap the references to Q^A and Q^B, meaning each table is updated using half of the experiences each. Note that $\gamma=0$ if s' is terminal, otherwise it is the discount rate.

where $\phi(s)$ is a feature vector that represents the state s, θ is a vector that represents the parameters of the network, and f_a is the component of the vector-valued function f that corresponds to action a. It is important to note that this function approximation allows for a continuous state-space, but a discrete action-space; the approximation can be extended further to continuous action-spaces as well, especially in actor-critic algorithms [16], but in this paper we only discuss the former approximation. In order to update the Sarsa network, we use a target similar to the target used in the tabular form,

$$Y = r + \gamma Q(s', a'; \theta) - Q(s, a; \theta), \tag{9}$$

and for Expected Sarsa,

$$Y = r + \gamma \sum_{a'} \pi(a' \mid s') Q(s', a'; \theta) - Q(s, a; \theta). \tag{10}$$

Deep Double Sarsa and Deep Double Expected Sarsa use two different neural networks that have the same structure; we represent these two networks by their parameters θ^A and θ^B. Similar to the tabular update rules, the target used for Deep Double Sarsa is

$$Y^A = r + \gamma Q(s', a'; \theta^B) - Q(s, a; \theta), \tag{11}$$

and the target for Deep Double Expected Sarsa is

$$Y^A = r + \gamma \sum_{a'} \pi(a' \mid s') Q(s', a'; \theta^B) - Q(s, a; \theta^A). \tag{12}$$

As in the tabular algorithms, the policy is derived from the average of $Q(s, a; \theta^A)$ and $Q(s, a; \theta^B)$. The algorithms for Deep Double Sarsa and Deep Double Expected Sarsa are shown in **Figure 3** and **Figure 4**, respectively.

Deep Double Sarsa

1. Initialize θ^A and θ^B arbitrarily
2. **loop**{over episodes}
3. Initialize s
4. Choose a from s using arbitrary policy
5. **repeat**{for each step of the episode}
6. Take action a, observe r, s'
7. Choose a' from s' using policy π derived from average of $Q(s, a; \theta^A)$ and $Q(s, a; \theta^B)$
8. $Y^A = r + \gamma Q(s, a; \theta^B) - Q(s, a; \theta^A)$
9. Train neural network using θ^A, $\phi(s)$, and Y^A
10. $s \leftarrow s'; a \leftarrow a'$
11. **with** probabilty 0.5:
12. swap(θ^A, θ^B)

Figure 3. Deep Double Sarsa algorithm, with neural network representation of the action-values. Lines 11 and 12 swap the references to θ^A and θ^B, meaning each table is updated using half of the experiences each. Note that $\gamma = 0$ if s' is terminal, otherwise it is the discount rate.

Deep Expected Double Sarsa

1. Initialize θ^A and θ^B arbitrarily
2. **loop**{over episodes}
3. Initialize s
4. Choose a from s using arbitrary policy
5. **repeat**{for each step of the episode}
6. Take action a, observe r, s'
7. Choose a' from s' using policy π derived from average of $Q(s,a;\theta^A)$ and $Q(s,a;\theta^B)$
8. $V_{s'}^B = \sum_{b'} \pi(b' \mid s') Q(s', b'; \theta^B)$
9. $Y^A = r + \gamma V_{s'}^B - Q(s,a;\theta^A)$
10. Train neural network using θ^A, $\phi(s)$, and Y^A
11. $s \leftarrow s'; a \leftarrow a'$
12. **with** probabilty 0.5:
13. swap(θ^A, θ^B)

Figure 4. Deep Double Expected Sarsa algorithm, with neural network representation of the action-values. Lines 12 and 13 swap the references to θ^A and θ^B, meaning each table is updated using half of the experiences each. Note that $\gamma = 0$ if s' is terminal, otherwise it is the discount rate.

3. Results

The experiment used to test the difference between Sarsa, Expected Sarsa, and their respective doubled versions was a simple grid world (see **Figure 5**) with two terminal states, one with a positive reward of 10 and the other with a negative reward of −10. Additionally, a blocking "wall" was placed in between the terminal states. Every time the agent moves a step in the environment, it receives an average reward r with mean μ and standard deviation σ. The state feature vector was represented by the concatenation of four one-hot encodings of the position of each of the objects,

$$\phi(s) = \left(\phi^{\mathbf{A}}, \phi^{\mathbf{P}}, \phi^{\mathbf{W}}, \phi^{\mathbf{G}}\right) \text{ and } \phi_i^j(s) = \begin{cases} 1 & \text{if } s^j = i \\ 0 & \text{if } s^j \neq i \end{cases}, \tag{13}$$

where $\phi_i^j(s)$ is the ith element of the one-hot encoding of the position of object $j \in \{\mathbf{A}, \mathbf{P}, \mathbf{W}, \mathbf{G}\}$, s^j is the position of that object, and $\phi(s)$ is the concatenation of all the $\phi^j(s)$ vectors. The number corresponding to each position can be seen in **Figure 5**, as well as the object positions, which were inspired from [20].

For comparison, we show the difference between the algorithms for rewards with both a deterministic distribution, where $p(r = \mu \mid s) = 1$ for non-terminal s, and a stochastic distribution of two values, where $p(r = \mu + \sigma \mid s) = p(r = \mu - \sigma \mid s) = 0.5$ for arbitrary σ for non-terminal s. In both the deterministic and stochastic cases, the negative terminal state **P** had a reward of −10 and the positive terminal state **G** had a reward of +10. An environment with both a positive and negative terminal state is ideal for testing the robustness of on-policy algorithms because they must learn a policy that minimizes the number of steps to the positive terminal state while avoiding states that may lead to the negative terminal state, due to the stochastic nature of the policies.

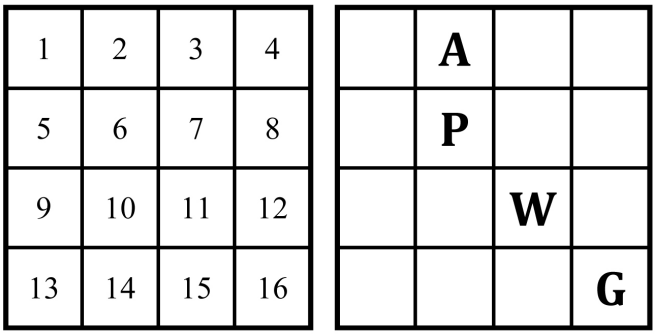

Figure 5. Grid world used to test the four algorithms discussed in this paper, left grid shows the number corresponding to the position and right grid shows the initial position of each object. **A** is the agent's starting position, **W** is the "wall", **P** is the terminal state with a reward of −10 (the "pit"), and **G** is the second terminal state with a reward of +10 (the "goal"). **A** is the only position allowed to change throughout the course of an episode.

In this paper, we first compare Sarsa, Expected Sarsa, Double Sarsa, and Double Expected Sarsa in tabular form, where $Q(s,a)$ is represented by a single table entry for each *s*, *a* pair, varying different parameters of exploration, learning, and rewards. Then, we discuss the extension of these algorithms to Q-Networks and Deep Q-Networks, using neural networks to approximate $Q(s,a)$ in a few scenarios that highlight the advantage of applying double learning to Sarsa and Expected Sarsa.

3.1. Tabular Representation of *Q*(*s*, *a*)

A comparison of Sarsa, Expected Sarsa, Double Sarsa, and Double Expected Sarsa under a deterministic reward system can be seen in Figure 6(a), showing the average return was over 100,000 episodes. Expected Sarsa and Double Expected Sarsa appear to have almost identical performance, although for small learning rates Expected Sarsa tends to perform marginally better; presumably, this is because the doubled version must train two tables and consequently takes longer to converge than the single version. In the first 1000 episodes under the same reward system, the average return collected by Double Expected Sarsa was about 6.4% less than the reward received by Expected Sarsa (not shown), which supports this hypothesis.

However, unlike the Expected algorithms, there is a clear performance difference between Sarsa and Double Sarsa for a deterministic reward. Like Expected Sarsa, Sarsa performs marginally better than Double Sarsa when the learning rate is small, although this is difficult to see in Figure 6(a). However, for learning rates greater than about 0.25, Double Sarsa shows a clear performance improvement over the standard Sarsa algorithm, especially as $\alpha \to 1$. When $\alpha = 1$, the average return collected by Sarsa quickly drops off to below 0, while Double Sarsa stillcollects an average return of about 3.5. This improvement in performance is likely a consequence of the Sarsa update rule, which uses the value of the next action $Q(s',a')$ to update the value at the current state and action $Q(s,a)$. This can introduce a substantial amount of variation in $Q(s,a)$,

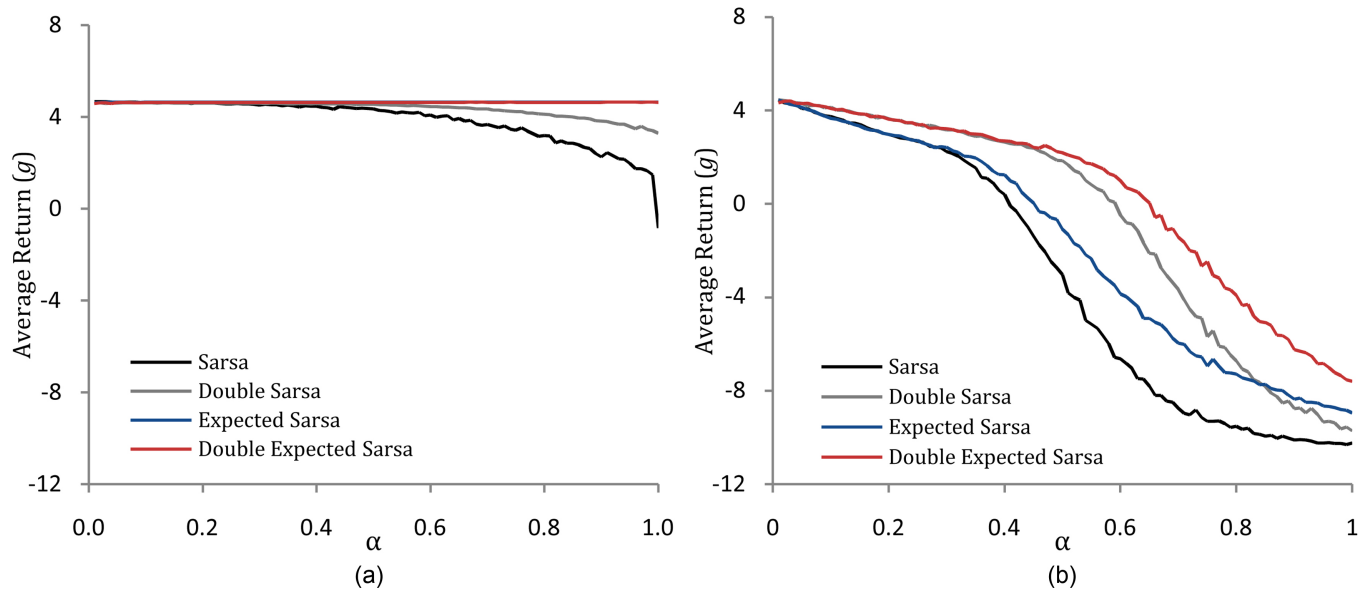

Figure 6. (a) Average return per episode vs. learning rate α for 100,000 episodes, deterministic reward with $\mu = -1$ and $\sigma = 0$, and an ϵ-greedy policy with $\epsilon = 0.1$. Expected Sarsa and Double Expected Sarsa are overlapping due to their convergence to very similar average returns. (b) Average return per episode vs. learning rate α for 100,000 episodes, stochastic reward with $\mu = -1$ and $\sigma = 7$, and an ϵ-greedy policy with $\epsilon = 0.1$.

especially if α is not annealed over time. Double Sarsa reduces this variation by decoupling the two tables, preventing against large changes in $Q(s,a)$, which tends to produce a more stable policy and increase the amount of reward collected.

Figure 6(b) shows the same comparison between the four algorithms with a stochastic reward, where $\mu = -1$ and $\sigma = 7$. For most of the learning rates tested, the doubled versions of the algorithms performed better than their respective single version. Unlike the deterministic case, Expected Sarsa does not have a clear advantage over Double Expected Sarsa in the first 1000 episodes (not shown), and like the deterministic case both exhibit the same trend over 100,000 episodes (Figure 6(b)), although the trend is significantly different.

Figure 7 shows the learning rate below which returns are positive and above which they are negative, comparing the learning rate which produces the same average return. These results indicate that the double estimators employed by Double Sarsa and Double Expected Sarsa allow for faster learning rates under the same stochastic reward conditions. As shown in Figure 7, around a 40% increase in the learning rate can be applied before Double Sarsa and Double Expected Sarsa collect rewards equivalent to Sarsa and Double Expected Sarsa, respectively. In real world applications, this can be a significant advantage, allowing greater returns to be collected earlier on in the learning process.

A comparison of the path length distributions between the four algorithms in the stochastic case is shown in Figure 8. The path length L is the number of steps that it took the algorithm to reach a terminal state in a given episode. Although all four algorithms reach the negative terminal state in $L = 1$ steps with approximately equal probability, it is apparent that Double Sarsa and Double Expected Sarsa tend to reach

Category	Metric	Algorithm			
		Sarsa	Double Sarsa	Expected Sarsa	Double Expected Sarsa
Learning Rate	Zero Crossing (α)	0.4095	0.5849	0.4535	0.6466
	Increase by Doubling		42.8%		42.6%
	Increase from Sarsa		42.8%	10.7%	57.8%
Computation Time	100,000 Episodes	228.4 s	249.7 s	238.0 s	248.0 s
	Increase by Doubling		9.3%		4.2%
	Increase from Sarsa		9.3%	4.2%	8.5%
Computational Efficiency	Increase by Doubling		33.5%		38.4%
	Increase from Sarsa		33.5%	6.6%	49.3%

Figure 7. Comparison of the performance of each algorithm with $\alpha = 0.1$, a stochastic reward system of $\mu = -1$ and $\sigma = 7$, and an ϵ-greedy policy of $\epsilon = 0.1$. The zero-crossing was determined by fitting a line $g = m \cdot \alpha + b$ to each of the curves in **Figure 6(b)**, using at least 7 points very close to the line $g = 0$, and finding α such that $m \cdot \alpha + b = 0$. In all cases, the R^2 value was greater than 0.9. The computation time was averaged over 100 runs; a single run includes 100,000 episodes after initialization of the algorithm, where an episode completed when the agent reaches the terminal state. *Increase by Doubling* was calculated by taking the ratio of the two metrics for Double Sasrsa to Sarsa and Double Expected Sarsa to Expected Sarsa. *Increase from Sarsa* was calculated by taking the ratio of the two metrics for Double Sarsa, Expected Sarsa, and Double Expected Sarsa to Sarsa. *Computational Efficiency* was computed by subtracting the percentage increase of computation time from the percentage increase of the zero crossing learning rate.

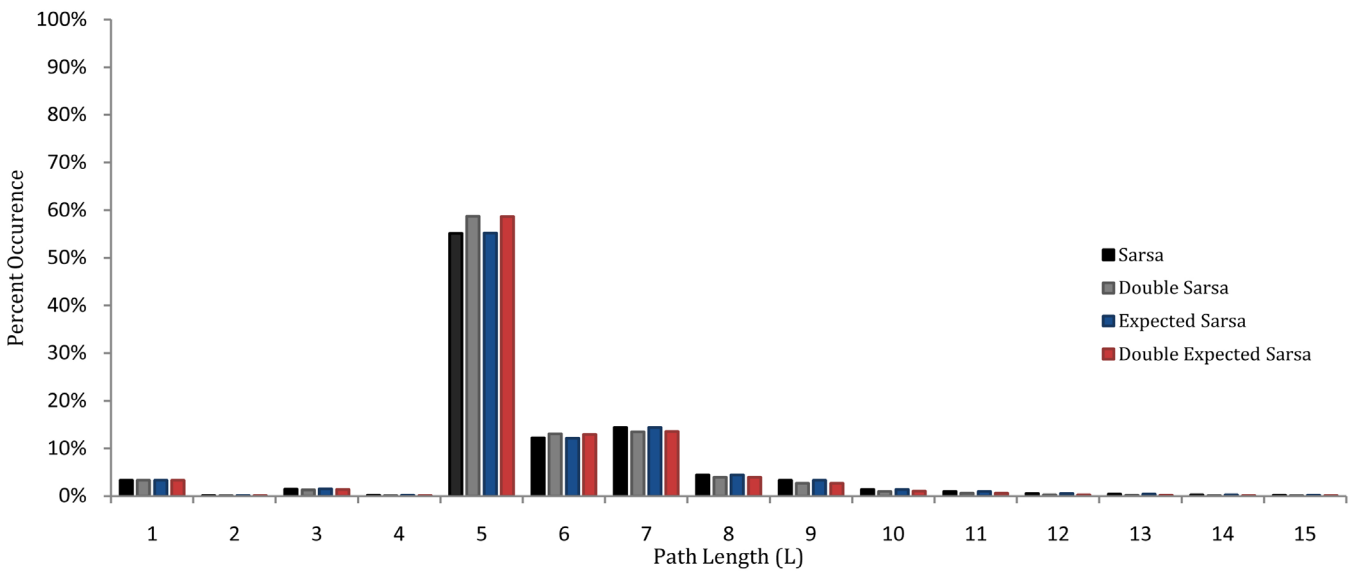

Figure 8. Path length distributions for all four algorithms, accumulated over 100 runs with 100,000 episodes each, truncated to a path length of $L = 15$ to show the most frequently occurring lengths. The path length is the number of steps that were needed to reach a terminal state. An ϵ-greedy policy of $\epsilon = 0.1$ was used, with a learning rate of $\alpha = 0.1$. The reward system was stochastic, with $\mu = -1$ and $\sigma = 7$.

the positive terminal state in fewer steps than Sarsa and Expected Sarsa. This is likely due to the double versions having a more stable policy as a result of having decoupled action-value estimates, preventing against large changes in the action-value, as well as the policy.

Also shown in the table is the average computation time for 100,000 episodes, with $\alpha = 0.1$. As can be seen in the table, the extra computational expense of Double Sarsa, Expected Sarsa, and Double Expected Sarsa is marginal, with all three algorithms taking less than 10% more time than Sarsa. This is in contrast to the increase in the zero crossing learning rate (α where $g = 0$), which in all cases is significantly greater than the original algorithm. This indicates that there is a significant advantage of using the doubled versions of Sarsa and Expected Sarsa when the reward is stochastic.

As shown with the Double Q-Learning algorithm, Double Sarsa and Double Expected Sarsa initially tend to have a lower estimate of the action-value than Sarsa and Expected Sarsa, respectively. **Figure 9(a)** shows the maximum action value, $\max_a Q(s,a)$, for the initial state of the agent, averaged over 1000 runs, with the doubled versions converging to the true value slower than the single versions. In addition, this plot shows the increased stability of the action-values that doubling the Sarsa and Expected Sarsa algorithms imparts. Interestingly, unlike Double Q-Learning, the Double Sarsa and Double Expected Sarsa tended to converge to a higher maximum action value (see **Figure 9(a)**) than Sarsa and Expected Sarsa, and approached the true value which was converged to in the deterministic case. This is likely due to the increased stability provided by the two decoupled action-value tables, instead of a single action-value table, which improves the quality of the policy and consequently increases the total reward.

This stability is especially important for on-policy algorithms, as a more stable behavior policy tends to reduce variation the distribution of states visited by the agent, as

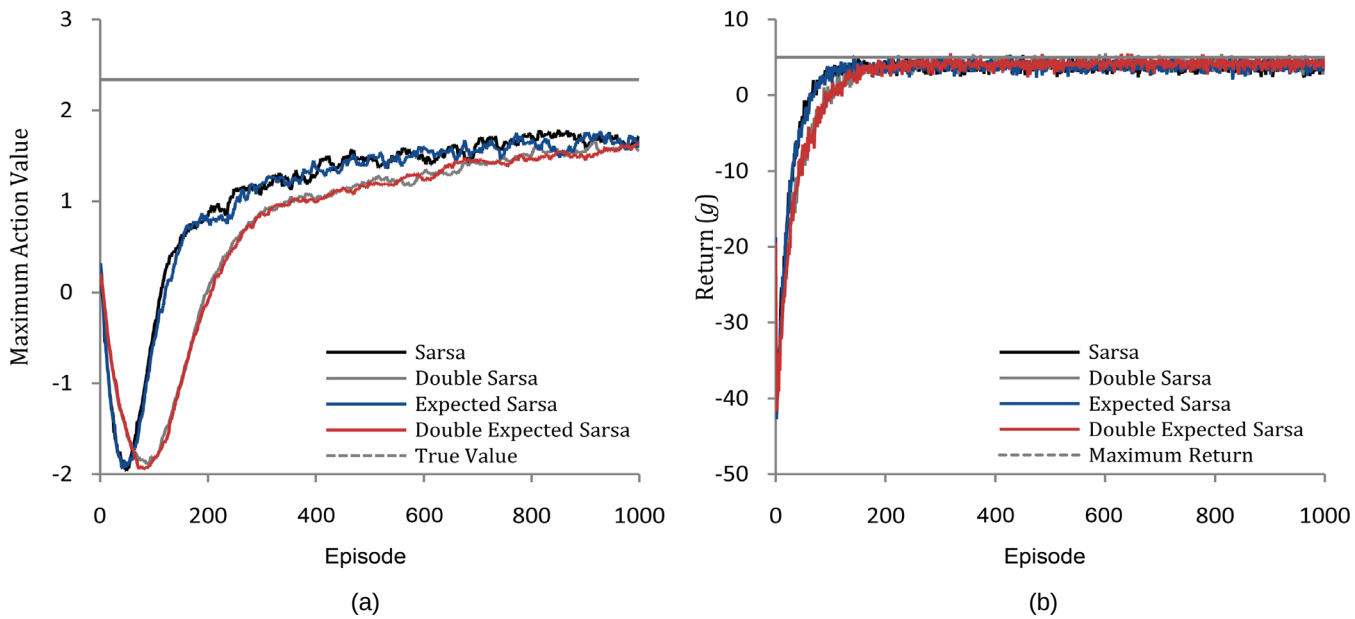

Figure 9. (a) Maximum action value for the initial state $\max_a Q(s,a)$ for each algorithm, averaged over 1000 runs. Reward was stochastic with $\mu = -1$ and $\sigma = 7$, with ϵ-greedy policy of $\epsilon = 0.1$ and a learning rate of $\alpha = 0.1$. The true value is the value which Expected converged to in the deterministic case. (b) Average returns for the same experiment of 1000 runs. The return g is the sum of the rewards in a given episode, or $g = \sum_{epsiode} r$. The convergence of the returns occurs much faster than the estimated maximum action value of the initial state.

well as the actions taken, making them significantly more predictable. For comparison, in the experiment shown in Figure 9(a), the average variance of the maximum action-value over all $T = 1000$ episodes, $\overline{\sigma^2} = \left(\sum_{t=1}^{T} \sigma_t^2\right) / T$ (where σ_t^2 is the variance of $Q(s,a)$ at episode t over 1000 runs) was computed. For Sarsa, $\overline{\sigma^2}$ was 4.51, 2.44 for Double Sarsa, 4.36 for Expected Sarsa, and 2.32 for Double Expected Sarsa. This is a significant reduction in variation, given the small difference in the average return curves shown in Figure 9(b).

Figure 10(a) and Figure 10(b) show similar results from an experiment with the same parameters, except that the number of episodes was increased to 100,000 and the number of runs decreased to 100. As can be seen, the average return collected by Double Sarsa and Double Expected Sarsa quickly surpasses that of Sarsa and Expected Sarsa, and the maximum action-value increases accordingly. Once again, this is likely due to the reduction in variation provided by double learning. It is also interesting to note that this is different than what was shown in [7] for Double Q-learning. That study found that the double estimator should, on average, underestimate the single estimator; this is clearly not the case in Figure 9(a). Likely, this is due to the fact that Q-learning is off-policy and takes the max in its update rule, while Sarsa is on-policy and often has a stochastic behavior (and target) policy.

The degree of effectiveness of Double Sarsa and Double Expected Sarsa is highly dependent on the distribution of rewards. Figure 11(a) shows the average return per algorithm against the standard deviation σ of the two-value stochastic distribution,

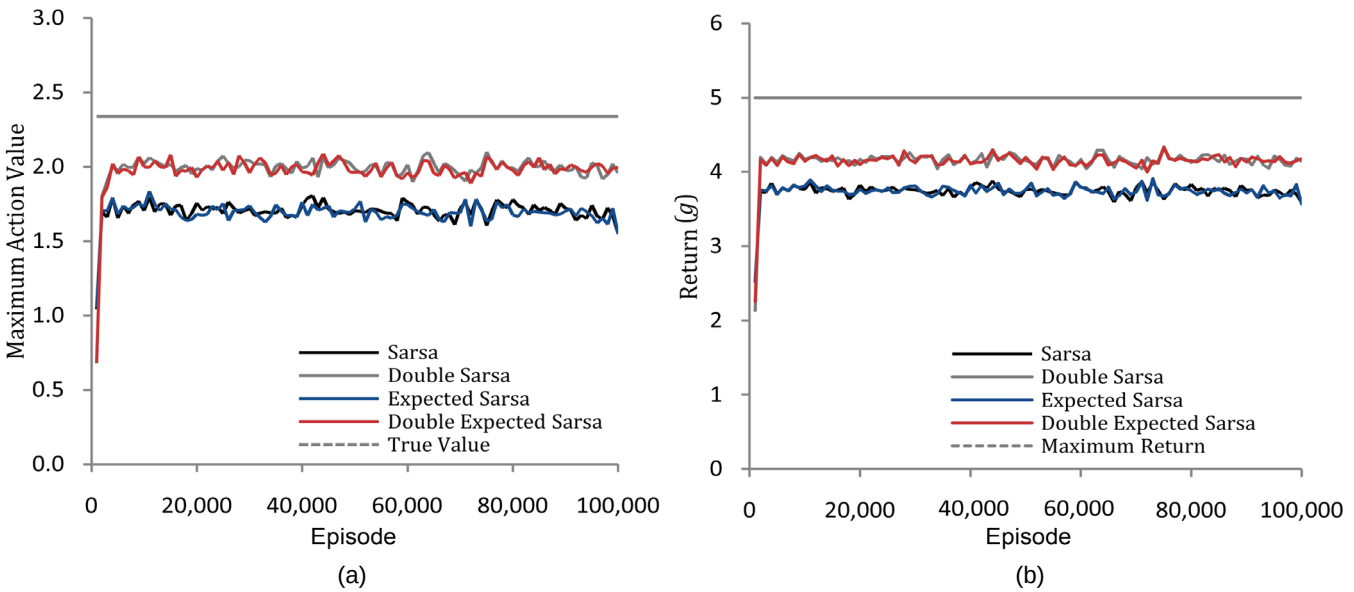

Figure 10. (a) Maximum action value for the initial state $\max_a Q(s,a)$ for each algorithm, averaged over 100 runs. Reward was stochastic with $\mu = -1$ and $\sigma = 7$, with ϵ-greedy policy of $\epsilon = 0.1$ and a learning rate of $\alpha = 0.1$. The graph shows the average value every 1000 episodes, averaged over the previous 1000 episodes. The true value is the value which Expected Sarsa converged to in the deterministic case. (b) Average return for the same experiment as Figure 9(a), averaged over the same 1000 episode intervals, where the return g is computed the same way as in Figure 9(b).

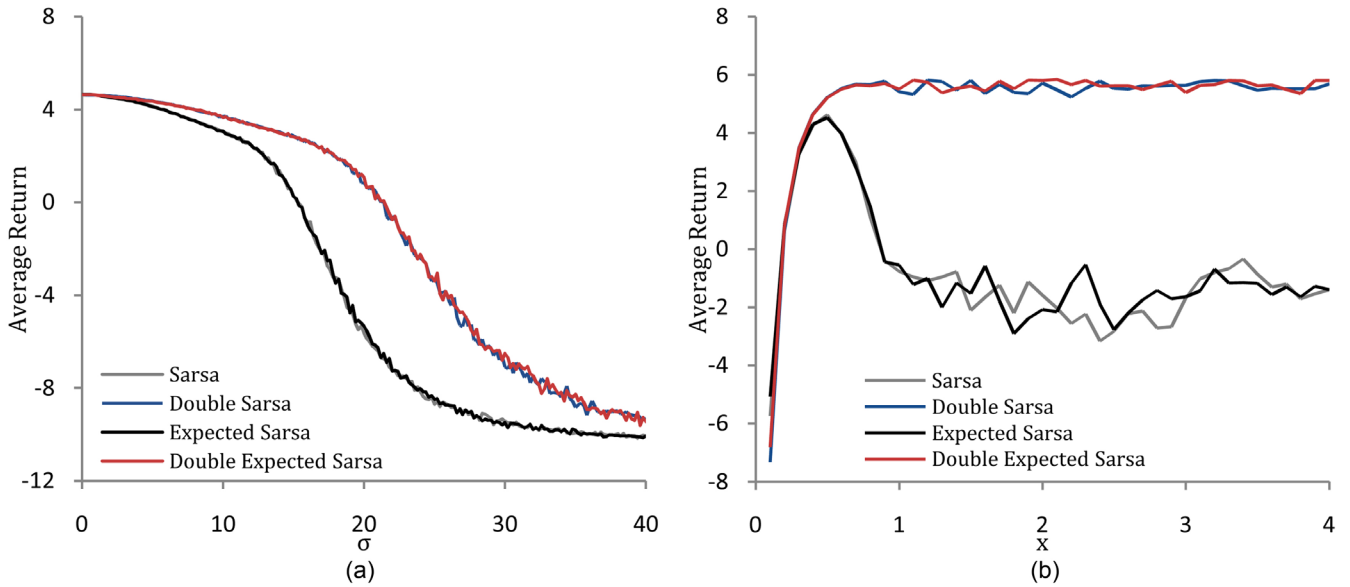

Figure 11. (a) Average return over 100,000 episodes with varying standard deviation of the reward system, σ. The reward distribution had two values, with $p(\mu-\sigma|s)=p(\mu+\sigma|s)=0.5$ for non-terminal s. For all cases, $\mu=-1$, $\alpha=0.1$, and an ϵ-greedy policy of $\epsilon=0.1$. Expected Sarsa had very similar average returns, as did Double Sarsa and Double Expected Sarsa. (b) Average return with varying power x for ϵ-decreasing policy of $\epsilon=1/n(s)^x$, taken over 10,000 episodes and averaged over 100 runs. A stochastic reward system was used, with $\mu=-1$ and $\sigma=7$, and the learning rate was kept constant at $\alpha=0.3$ for all four algorithms.

with $\mu=-1$, $\alpha=0.1$, and $\epsilon=0.1$. It appears that the doubled versions are significantly more robust with respect to variations in the reward distribution. For example, when $\sigma=20$, Double Sarsa and Double Expected Sarsa still net positive rewards, while Sarsa and Expected Sarsa are significantly negative. Note that this means the reward that $r\in\{-21,19\}$ for non-terminal s, which covers a range that is about double the range of the terminal rewards, $r\in\{-10,10\}$.

The advantage of doubling Sarsa and Expected Sarsa can also be seen in **Figure 11(b)**, which compares the average return collected by each algorithm over 10,000 episodes with an ϵ-decreasing [21] policy and $\alpha=0.3$. For the ϵ-decreasing policy, ϵ was calculated according to $\epsilon=1/n(s)^x$, where $n(s)$ is the number of times s was visited and x is an arbitrary exponent used to control how quickly $\epsilon\to 0$. For the same learning rate, a faster decreasing ϵ (a larger x) can be used with Double Sarsa and Double Expected Sarsa than with Sarsa and Expected Sarsa before the returns collapse, meaning a greedy policy can be more quickly achieved and greater returns can be collected; in situations where exploration is highly undesirable (e.g. it is expensive), this can be a significant advantage.

3.2. Neural Network Representation of $Q(s, a)$

In order to test the robustness of each algorithm, we tested each of them with neural network function approximation of $Q(s,a)$. All neural networks were implemented using the Keras library [22], and backpropagation was performed using the RMS Prop

technique [23]. A comparison of different neural network architectures applied to each algorithm can be seen in **Figure 12**. The parameter n represents trials from a range of values; typically, $n \in \{5,10,\cdots,100\}$. The returns were averaged over 16 runs in order to reduce natural variations in performance from random initialization of the network parameters, θ, and the maximum average return for each architecture was taken over n according to $g_{\max} = \max_n g_n$, where g_n is the average return for a given architecture with parameter n. As can be seen in the figure, a variety of trends are apparent. First, as the network architecture transitions from shallow to deep, the average return collected generally decreases. For a random policy, the average return g was determined experimentally to be about -16.03, indicating that any network architecture with an average return $g > -16.03$ has learned a policy better than random, and any network architecture with an average return $g > -10$ has learned a policy that must reach the positive terminal state at least part of the time.

For the architectures shown, the average increase in return of Double Sarsa (DS) over Sarsa (S), $\overline{g_{DS} - g_S}$, is 1.05 ± 6.29, for Expected Sarsa (ES) the average increase in return over Sarsa $\overline{g_{ES} - g_S}$ is 1.42 ± 3.53, and for Double Expected Sarsa (DES), $\overline{g_{DES} - g_S}$ is 0.59 ± 7.23 (the uncertainty is the standard deviation of the differences). Clearly, Double Sarsa, Expected Sarsa, and Double Expected Sarsa are improvements over Sarsa when neural networks are used. Presumably, this is because all three provide increased stability to the action-value estimates, in different ways.

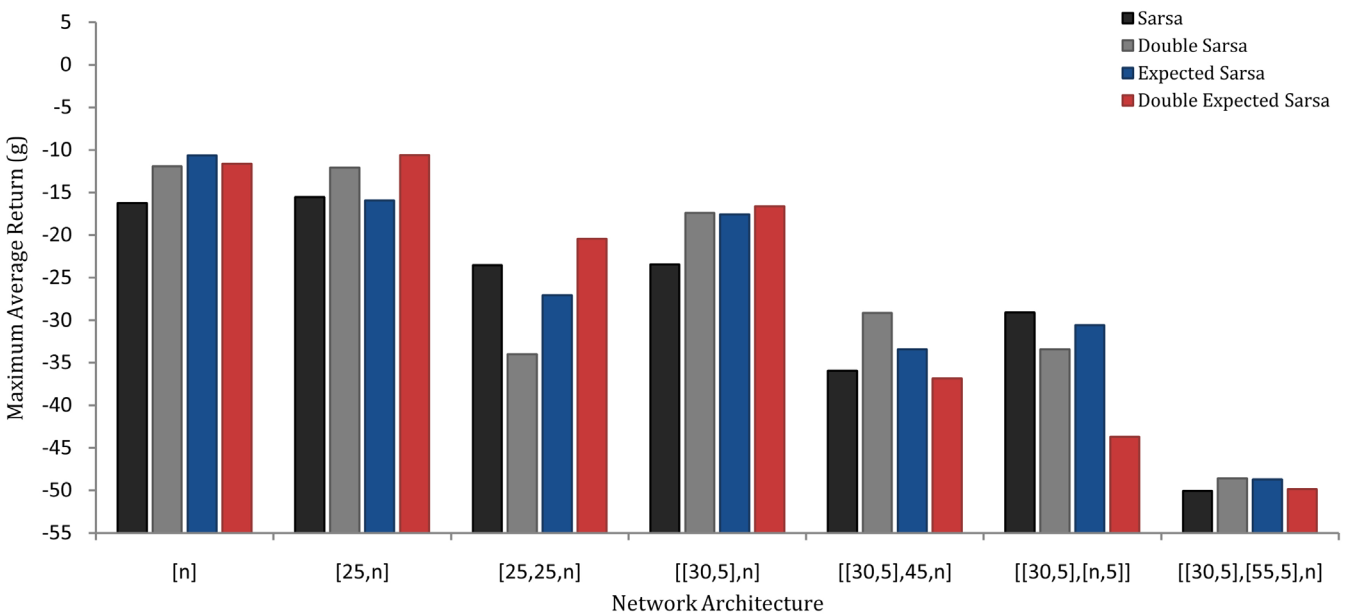

Figure 12. Comparison of average returns collected by neural network implementations of the four algorithms over 10,000 episodes, averaged over 4 runs and maximized over the size of the last hidden layer ($max_n g_n$ for each algorithm and network parameter). The input was a vector of length 64, concatenating the one-hot encodings of the positions of the agent, the "wall", the "pit", and the "goal", each vectors of length 16. The network architecture represents the size of the hidden layers as a list, in consecutive order from left to right, and $n$represents a parameter that was typically in the range of 5 to 10. A sub-list such as $[n_c, n_w]$ indicates this layer is a convolutional layer with n_c filters of n_w inputs. The output layer had 4 units, one for each action-value. Each hidden layer used a rectified linear activation function, and the output layer used a linear activation function.

Although it is apparent that doubling Sarsa and Expected Sarsa generally improves the performance the algorithms when neural networks are used to approximate the action-value function, the advantage of deep learning over shallow learning is contradicted by our experiments. Presumably, this is because there are comparatively very few states in our simple grid world environment; it is likely that, as the size of the grid increases, the benefit of neural network approximation might increase. However, even in this case, the advantage of deep learning over shallow learning might not fully become apparent without increasing the complexity of the environment; deep neural networks might not be beneficial until the environment reaches a certain level of complexity and non-linearity.

Even so, the experiments summarized in **Figure 12** show the effect of using function approximation on an on-policy algorithm, which in this case decreased the average return significantly, which is something that was not observed with off-policy algorithms. Likely, this is a product of the increased feedback present in on-policy algorithms; the choice of action a affects the update of θ, which changes the action-values and policy, and consequently affects the choice of the next action a'. In off-policy algorithms, θ does not affect the policy, meaning that there is a greater degree of stability when training the neural network approximator.

4. Conclusion

Current on-policy reinforcement algorithms are less effective when rewards are stochastic, requiring a reduction in the learning rate in order to maintain a stable policy. Two new on-policy reinforcement learning algorithms, Double Sarsa and Double Expected Sarsa, were proposed in this paper to address this issue. Similar to what was found with Double Q-learning, Double Sarsa and Double Expected Sarsa were found to be more robust to random rewards. For a constant learning rate α, these algorithms are more stable to large variations in rewards, allowing them to still achieve significant returns when the standard deviation σ is significantly larger than the magnitude of the rewards received in the terminal states. We found that the estimated action-values of Double Sarsa and Double Expected Sarsa were much more stable than those of both Sarsa and Expected Sarsa, which resulted in a better policy. However, unlike Double Q-learning, we showed that the double estimators of the proposed algorithms could overestimate the single estimators of the original algorithms. In addition, we found that, for the same average return, a more aggressive learning rate could be used with the doubled versions, at only a minor computational cost. Finally, we demonstrated that this technique could be extended with neural networks and deep reinforcement learning, showing the same improvement from doubling as the tabular forms do. Future work should focus on exploring the robustness of the neural network versions of Double Sarsa and Double Expected Sarsa in more complex environments.

Acknowledgements

We would like to thank the Summer Research Institute at Houghton College for pro-

viding financial support for this study.

References

[1] Kaelbling, L.P., Littman, M.L. and Moore, A.W. (1996) Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research,* **4**, 237-285.

[2] Williams, R.J. (1992) Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning,* **8**, 229-256. http://dx.doi.org/10.1007/BF00992696

[3] Sutton, R.S. (1988) Learning to Predict by the Methods of Temporal Differences. *Machine Learning,* **3**, 9-44. http://dx.doi.org/10.1007/BF00115009

[4] Sutton, R.S. and Barto, A.G. (1998) Reinforcement Learning: An Introduction. Vol. 1, MIT press, Cambridge.

[5] Konda, V.R. and Tsitsiklis, J.N. (1999) Actor-Critic Algorithms. *NIPS Proceedings*, **13**, 1008-1014.

[6] Watkins, C.J.C.H. (1989) Learning from Delayed Rewards. Ph.D. Thesis, University of Cambridge, Cambridge.

[7] Hasselt, H.V. (2010) Double Q-Learning. In: Lafferty, J.D., Williams, C.K.I., Shawe-Taylor, J., Zemel, R.S. and Culotta, A., Eds., *Advances in Neural Information Processing Systems*, Curran Associates, Inc., New York, 2613-2621.

[8] Rummery, G.A. and Niranjan, M. (1994) On-Line Q-Learning Using Connectionist Systems. Department of Engineering, University of Cambridge, Cambridge.

[9] Van Seijen, H., Van Hasselt, H., Whiteson, S. and Wiering, M. (2009) A Theoretical and Empirical Analysis of Expected Sarsa. 2009 *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, Nashville, 30 March-2 April 2009, 177-184. http://dx.doi.org/10.1109/ADPRL.2009.4927542

[10] Boyan, J. and Moore, A.W. (1995) Generalization in Reinforcement Learning: Safely Approximating the Value Function. In: Tesauro, G., Touretzky, D.S. and Leen, T.K., Eds., *Advances in Neural Information Processing Systems*, MIT Press, Cambridge, 369-376.

[11] LeCun, Y., Bengio, Y. and Hinton, G. (2015) Deep Learning. *Nature,* **521**, 436-444. http://dx.doi.org/10.1038/nature14539

[12] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. (2013) Playing Atari with Deep Reinforcement Learning. https://arxiv.org/abs/1312.5602

[13] van Hasselt, H., Guez, A. and Silver, D. (2015) Deep Reinforcement Learning with Double Q-Learning. http://arxiv.org/abs/1509.06461

[14] Liang, Y., Machado, M.C., Talvitie, E. and Bowling, M. (2016) State of the Art Control of Atari Games Using Shallow Reinforcement Learning. *Proceedings of the* 2016 *International Conference on Autonomous Agents & Multiagent Systems*, Singapore, 9-13 May 2016, 485-493.

[15] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D. and Riedmiller, M. (2014) Deterministic Policy Gradient Algorithms. *Proceedings of the* 31*st International Conference on Machine Learning* (*ICML*-14), Beijing, 21-26 June 2014, 387-395.

[16] Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D. (2015) Continuous Control with Deep Reinforcement Learning. https://arxiv.org/abs/1509.02971

[17] Silver D., Huang A., Maddison C.J., Guez A., Sifre L., Van Den Driessche G., Schrittwieser

J., Antonoglou I., Panneershelvam V., Lanctot M., *et al.* (2016) Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, **529**, 484-489. http://dx.doi.org/10.1038/nature16961

[18] Tromp, J. (2016) Number of Legal Go Positions. https://tromp.github.io/go/legal.html

[19] Tokic, M. and Palm, G. (2011) Value-Difference Based Exploration: Adaptive Control between Epsilon-Greedy and Softmax. In: Bach, J. and Edelkamp, S., Eds., *KI* 2011: *Advances in Artificial Intelligence*, Springer, Berlin, 335-346.

[20] Brandon (2015) Q-Learning with Neural Networks. http://outlace.com/Reinforcement-Learning-Part-3/

[21] Vermorel, J. and Mohri, M. (2005) Multi-Armed Bandit Algorithms and Empirical Evaluation. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M. and Torgo, L., Eds., *Machine Learning*: *ECML* 2005, Springer, Berlin, 437-448. http://dx.doi.org/10.1007/11564096_42

[22] Chollet, F. (2015) *keras*, GitHub. https://github.com/fchollet/keras

[23] Tieleman, T. and Hinton, G. (2012) Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of its Recent Magnitude. *COURSERA*: *Neural Networks for Machine Learning*, **4**, 26-30.

Scientific Research Publishing

Submit or recommend next manuscript to SCIRP and we will provide best service for you:

Accepting pre-submission inquiries through Email, Facebook, LinkedIn, Twitter, etc.
A wide selection of journals (inclusive of 9 subjects, more than 200 journals)
Providing 24-hour high-quality service
User-friendly online submission system
Fair and swift peer-review system
Efficient typesetting and proofreading procedure
Display of the result of downloads and visits, as well as the number of cited articles
Maximum dissemination of your research work

Submit your manuscript at: http://papersubmission.scirp.org/

Or contact jdaip@scirp.org