Unsupervised Learning and using SARSA to teach a robot to navigate a virtual problem space to find reward locations

***Abstract*—This project looks at some of the key concepts of unsupervised learning, including Principal Component Analysis (PCA), Hebbian Learning and Oja's learning rule and discusses the implementation and results obtained from using SARSA and reinforcement learning, to allow a robot to learn goal-oriented behaviour (homing) in relation finding a reward in a virtual problem.**

I. Unsupervised Learning

This section answers the questions set out in part I of the assignment brief, with each sub section referring to each question in sequence.

A. Using machine learning to classify spoken vowels

One method of machine learning which would probably not be best applied to the dataset is supervised learning. For supervised learning to be successful a training set must be provided which contains fully labelled data. Whilst it is possible to use the small subset of labelled data as a training set in this example, it is unlikely to be successful as the training set must be larger than the test set (the unlabelled data) overall.

Unsupervised learning is one method of machine learning which is suitable for use on this dataset. Unsupervised learning makes use of the raw data, without labels. The small subset of data which contains labels in this set can be ignored if necessary. It is likely to be successful as unsupervised learning detects the intrinsic structure in the data and clusters the data together which is useful which classifying vowels as it is also a good method for detecting anomalies where some data is unclear and not a vowel.

Another method of machine learning, reinforcement learning is also not a good choice for classifying data in this dataset. Reinforcement learning does not typically need labelled data and focuses on finding a balance between exploration and exploitation of existing knowledge [1]. It typically works by interacting with the environment and learning through rewards and punishments. In this example, there is no environment to be explored and with only a small subset of the data being labelled it would be difficult to establish rewards and punishment when the network classifies each unique sample, unless these were implemented manually.

In summary, the most suitable machine learning method that can be applied to this dataset is unsupervised learning as it works to cluster data without any need for any explicit feedback and is capable of detecting anomalies within the dataset. Reinforcement learning could be used on this dataset, however unless rewards and punishments were implemented manually for each sample which would not be ideal, then reinforcement learning would not be best suited for this dataset. Finally, supervised learning could potentially be used as the number of classes are known and some knowledge of the classes already exists, however it is probably not the most suitable approach for this dataset as a large proportion of the data is unlabelled which will likely lead to the network being very inaccurate.

B. Principal Component Analysis

Principal Component Analysis (PCA) is used in machine learning primarily for dimensionality reduction. There are two main techniques used in dimensionality reduction; feature extraction and feature elimination [2]. Feature extraction is technique used in PCA to combine sets of independent variables which combine features from the input variables in a specific ways allowing for the "least important" features to be dropped. PCA should be used when the user either wishes to reduce the number of variables, but is unsure of which are least important or when the user would like to make sure the variables are independent of one another [2].

The PCA algorithm has four key steps. The first step is to normalize the data to have a mean of 0 and a standard deviation of 1. This can be achieved by [5]:

$$x_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j} \tag{1}$$

Where

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)} \tag{2}$$

And

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2 \tag{3}$$

The second step of the algorithm is to compute

$$\sum = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} x^{(i)^T} \in \mathbb{R}^{nxn} \tag{4}$$

The third step of the algorithm is to computer the orthogonal eigenvectors of the k largest eigenvalues, $u_1, \ldots, u_k$ and the final step of the algorithm is to project the data which maximises the variance among all k-dimensional spaces [5].

Another way to think about this procedure is to first calculate the signal means, then subtract the means from the signals. Then calculate the covariance matrix, find the eigenvalues

and eigenvectors, ensuring the length of the eigenvectors is 1. Decide how many eigenvectors should be kept, Then form the "Feature Vector" from the eigenvectors and then finally derive the new data.

PCA essentially looks at the redundancy of information within the variables in a given dataset and returns a smaller set of variables after applying some sort of transformation to them [6]. If two variables have a high correlation, then it is clear that not much additional information is being dervied from the use of both of these variables, so PCA uses translation and rotation of axes to transfer the variance of one variable onto another, where the direction is determined by the use of eigenvectors and eigenvalues [6]. After running PCA, the data will have less variables than the number of input variables where correlation is used to determine the "usefulness" of each variable to determine which can be removed or combined.

C. Problems with the minimal Hebbian learning rule

the Hebbian learning rule is defined in equation (5). An issue with the minimal Hebbian learning rule is that it is unstable as the weight vector can increase infinitely. In contrast, Oja's rule, which is defined in equation (6) converges to a weight vector which has a length of 1, and one that is an eigenvector of the correlation matrix.

$$\Delta w_{ij} = \alpha v_i^{post} v_j^{pre} \tag{5}$$

$$\Delta w_{ij} = \alpha v_i^{post} v_j^{pre} - \alpha w_{ij} \left(v_i^{post}\right)^2 \tag{6}$$

D. Yuille's Update Rule

1) Deriving the update rule:

$$\Delta w_k = \eta \left(y x_k - w_k \left(\sum_i w_i^2 \right) \right) \tag{7}$$

2) comparing Yuille's and Oja's rule: Oja's rule converges to a weight vector which is an eigenvector of the correlation matrix with a length equal to one. The convergence means the weight change is equal to zero and the weight vector lies in the direction of the principle component (maximal eigenvector) for zero mean signals [3].

Yuille's rule was a modification of the Hebbian rule proposed to prevent divergence [4]. It has been shown the Yuille's rule converges in the same or opposite direction of the principal component and whose norm is given by the square root of the largest eigenvalue of [4].

$$\begin{cases} w^1 \; arbitrary \\ e^{k+1} = w^k + \rho \left[y^k x^k - ||w^k||^2 w^k \right] \end{cases} \tag{8}$$

Yuille's rule can be written as shown in Equation (8) [4]. A key difference between the rules is that Yuille's rule tends to converge on lower value, however the direction of convergence of the weight vectors is almost identical for both rules. The overlap in convergence appears to be quite small to begin with but increases rapidly towards one [4].

II. Reinforcement Learning

This section will discuss the implementation of goal-oriented behaviour used by a virtual robot to navigate throughout a room to reach a charging point (reward locations). Each sub-section will answer the questions sequientally as outlined in the brief (i.e. section a refers to question 1 and so on).

A. Learning Curves

Running the solution twice will produce different learning curves due to the randomised nature of the algorithm. The algorithm is initialised with a random start location, random weights and depending on the implementation may move randomly also. This means that it is difficult to reproduce the same results when running the algorithm as the randomised parameters will have a significant affect on the number of steps taken to reach the reward location. Figure 1 and Figure 2 show the resulting plots from running the algorithm with fifty repetitions, a learning rate of 0.5, and $\gamma = 0.9$

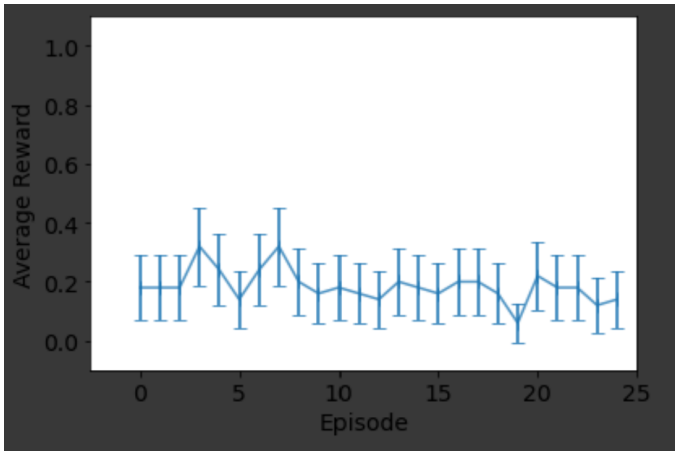

Fig. 1. The average reward plotted against the episode for the initial solution

Figure 1 shows the average reward plotted against the number of episodes. In this example, the solution was run for 15 episodes and the average reward seems to be high enough to suggest the the robot had no issue finding the reward location on most of the trials.

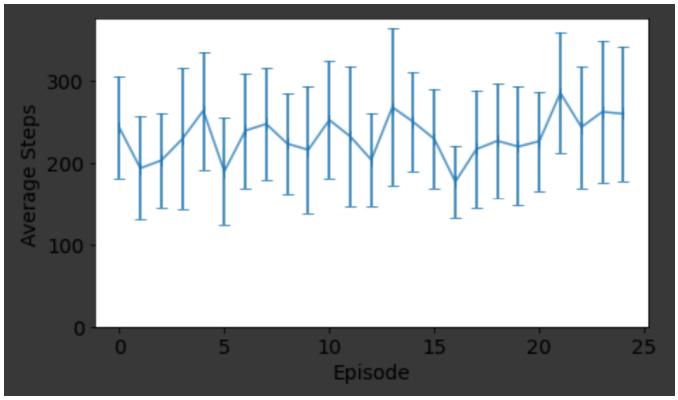

Fig. 2. The average number of steps plotted against the episode for the initial solution

Figure 2 shows the average number of steps taken to find the reward location using the initial solution. It is clear from the large error bars that the results are slightly unreliable and there is quite a large variation in the number of steps taken. This suggests that the robot does not improve as we would expect as the number of trials increases, this may however be because in the initial solution we allow the robot to explore unguided which will influence its ability to improve over time.

B. SARSA(λ)

A snippet of the code used to implement an eligibility trace with SARSA(λ) can be seen in Appendix A. Overall, the robot seemed to improve consistently on all trails, with the robot able to find the reward location on all trials. Using SARSA(λ) also increases the overall runtime quite significantly, meaning that a cost benefit analysis should be considered before using it to see whether or not the overall performance improvement outweighs the cost of the increased time the algorithm takes to run.

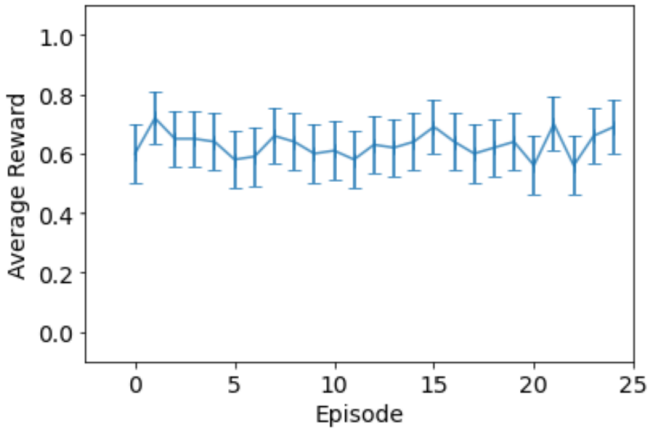

Fig. 3. The learning curve after running the algorithm using SARSA(λ)

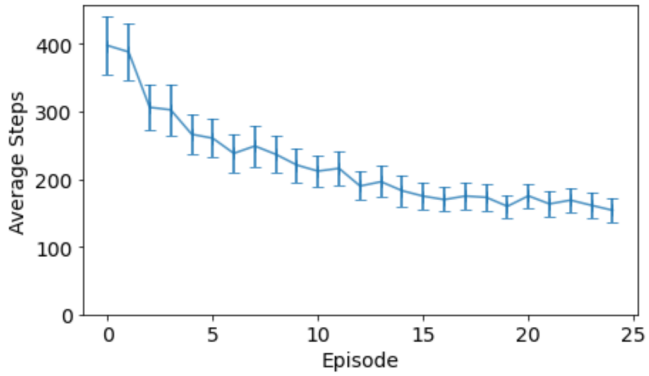

Fig. 4. The average numver of steps taken plotted over the number of episodes using SARSA(λ)

It is clear when comparing Figure 3 and Figure 4 to Figure 1 and Figure 2 that there is a notable difference in performance when using SARSA(λ), where the average reward has increased quite significantly meaning it is preferable to use SARSA(λ) where possible. To allow for the results to be compared accurately, the results shown in Figure 2 have been created using the same parameters as those used in Figure 1.

C. Aiding Exploration

The code used to aid the robot with exploration can be seen in Appendix B. The exploration behaviour is the epsilon greedy algorithm provided in lab 8. If greedy is true (i.e. the random number is greater than ϵ) then the 'best' action will be chosen, otherwise a random action will be chosen. To ensure optimal performance of this method, the value of the ϵ parameter must be tuned. A poorly chosen ϵ value will mean that the robot will explore using randomly chosen actions more often than not which will not improve overall performance.

It is difficult to establish the best value for ϵ overall when the other parameters are varied continuously throughout the testing of this project. However it seemed that a large learning rate (≈ 0.9) along with a small value of ϵ worked particularly well in most test cases. The optimal value for ϵ found during the testing at this stage was ≈ 0.1. Figure 5 shows the results of running the solution using SARSA with a learning rate of 0.5, $\gamma = 0.9$ and $\epsilon = 0.1$ so that the results can be compared to that shown in Figure 1.

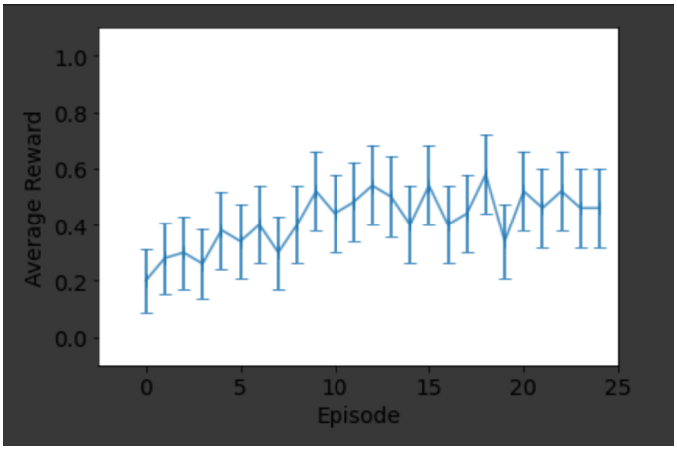

Fig. 5. The learning curve after running the algorithm with epsilon-greedy

D. Finding the optimal parameter values

After thorough testing in an attempt to find optimal values for the parameters of the solution, the most optimal found where when the learning rate was $\approx 0.5, \gamma \approx 0.5, \epsilon \approx 0.1$. Overall, the results show that over time the results tend to perform similarly well, and in some case there are no discernible differences between some of the values tested.

Overall it can be said that for the most part many of the variables perform similarly well over time The results from varying the values of these variables can be seen in Figure 6.

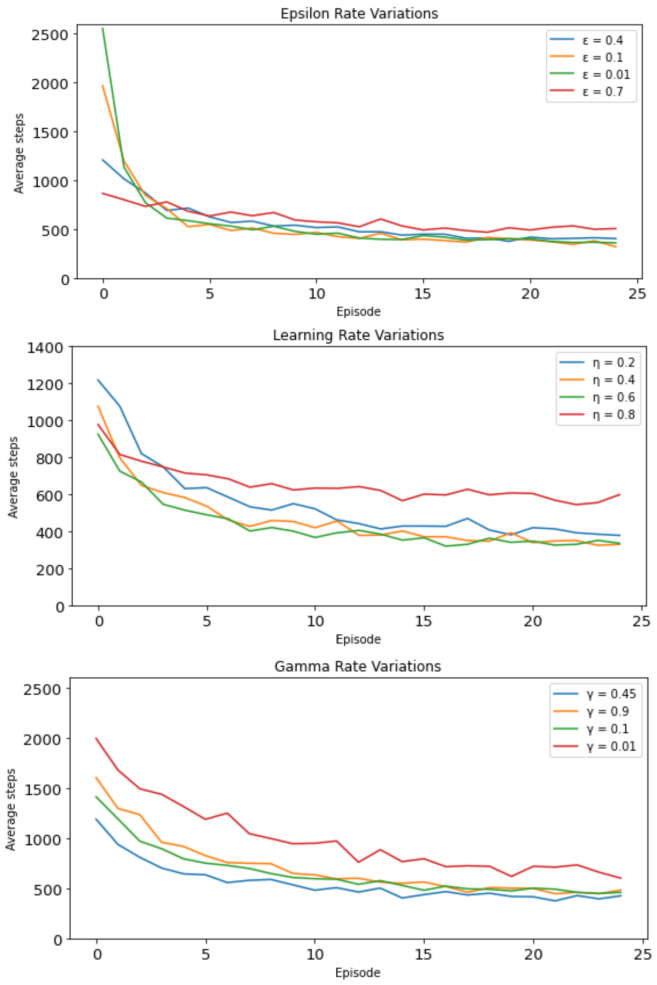

Fig. 6. The results from testing to find the optimal learning rate

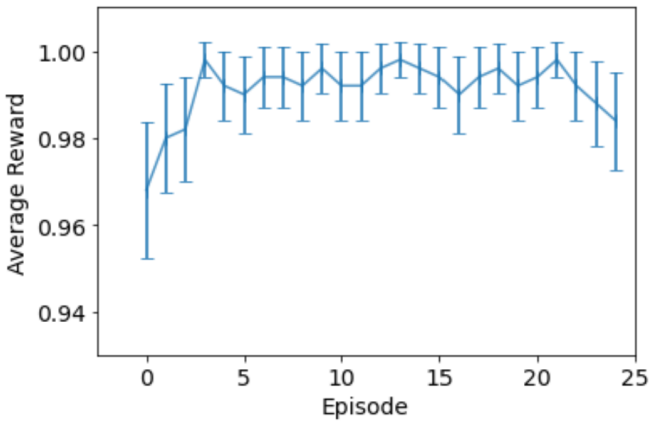

Fig. 7. The average reward plotted against the number of episodes when the robot have 8 possible actions to choose

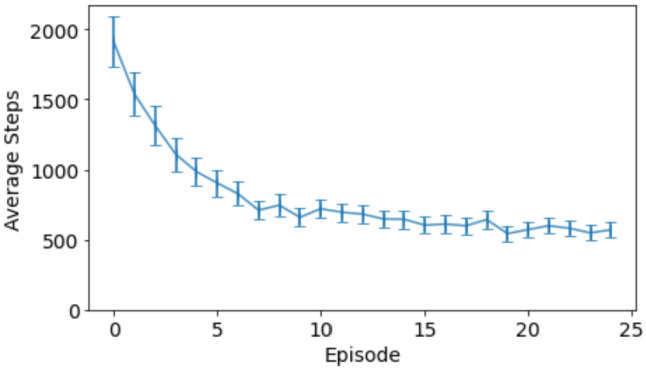

Fig. 8. The average number of steps taken for the robot to reach the goal state with 8 possible actions

E. Increasing the number of actions

Up until this point, the solution has been run with four potential actions; move North, East, South or West. To add the ability to increase the amount of actions to eight (to include diagonal movements) a global variable called 'actions' has been defined which can be changed to alter the number of movements the robot can make. The value of this variable must be an integer $0 \leq actions \leq 8$. Overall, the time taken for the algorithm to execute seemed to have increased notably, whilst the average reward and average number of steps taken to reach the reward location seemed to remain very high across trials. The results obtained when running the solution when allowing the robot to choose from eight possible actions is shown in Figures 7 and 8.

F. Spawning additional reward locations

To test how the robot reacts to having multiple reward locations, four additional locations with the reward at each location increasing by five in each location, i.e. the potential rewards the robot could receive was now 1, 6, 11, or 16. It is expected that the robot will learn to find the reward location with the highest value over time. In the case of this implementation, the average reward remained fairly constant over the number of trials, and for the most part the average was approximately the average of the four reward locations. This test was run using ϵ-greedy behaviour to assist with exploration and SARSA(λ). The results of this test suggest there is an issue with the implementation of one of these algorithms. The results for this implementing are summarised in Figure 9.

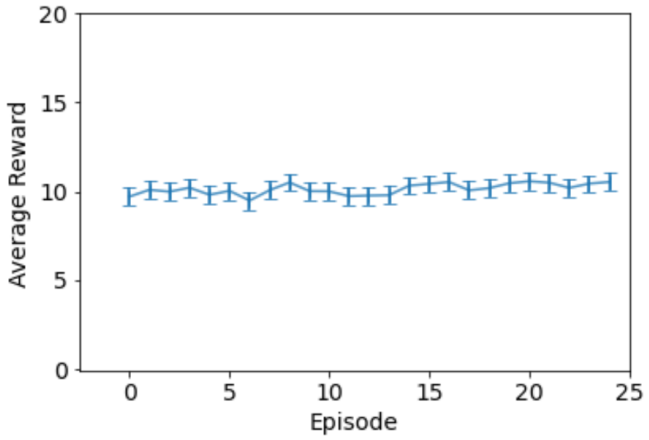

Fig. 9. The results from testing the robot with 5 reward locations

G. Increasing the size of problem space

As the size of the problem space increases, the amount of time it takes to search the problem space increases exponentially. For example, if the problem space consisted on one-thousand squares, and the agent could choose one of four actions (move North, East, South or West) then the number of possible combination of actions to find the reward location is infinitely large for any practical purposes. One approach that could be explored to help reduce the time spent searching would be to look at metaheuristics, such as ant colony optimisation. However this would likely not reduce the search time to something feasible if the problem space was very large.

To help deal with the uncertainty of the problem in a large space, it would be useful to try and approximate or model the Q-value function in some way to help the robot understand which operations to avoid, thus increasing the probability of the robot successfully finding the reward location. One such example of this comes from some research conducted by Tuyls et al. [12] where a combination of decision trees and Bayesian networks were used to model the environment and Q-function, where the environment was a soccer field with the goal to allow a robot to play soccer. Whilst this environment is very different to the problem addressed in this assignment, the theory of how to deal with large problem spaces to which there is little to no prior knowledge of can potentially be applied.

Similarly, research by Dulac-Arnold et al [13] suggets that a good approach may be to leverage any prior knowledge actions and embed them in a generalisable continuous space, whilst also suggesting that "approximate nearest neighbour methods could be used to methods allow for logarithmic-time lookup complexity relative to the number of action" [13] which would be useful for reducing the runtime of the algorithm in larger problem spaces.

In summary, there are many pieces of literature which have attempted to address the problem of how to deal with large problem spaces in reinforcement learning, however it is difficult to say which approach would be best in the context of the problem posed in this assignment. Generally speaking, the best approaches appear to include allowing Q-value function to be approximated in some way, and either attempting to reduce the number of state in the problem space or using existing knowledge to help future decision making.

H. Implementing deep Q-learning

Due to time constraints on this project, deep Q-learning was not fully implemented, however we will discuss the impact deep Q-learning would likely have on the solution should it have been implemented correctly.

In deep Q-learning, a neural network is used to estimate the values of the Q-value function. Some of the key steps involved in deep Q-learning involve: the user stores all past experiences in memory, then the next action is determined by the maximum output of the Q-function. With deep Q-learning the loss function could be determined by the mean-squared error for example, meaning it is simply a regression problem where the target is not known [11].

Typically in deep Q-learning, the Bellman Function (shown in equation (9)) is used as the cost function, and when looking to minimise the difference between the left and right hand side of the cost function it typically becomes something like that shown in equation (10). [14].

$$Q(s,a) = r(s,a) + \gamma \max_{a} Q(s',a) \tag{9}$$

$$Cost = \Big[Q(s,a;\theta) - (r(s,a) + \gamma \max_{a} Q(s',a;\theta)\Big]^2 \tag{10}$$

Equation (10) is essentially the mean squared error function (shown in equation (11)) where the future rewards are the target (y') and the current Q value is the prediction (y) [14].

$$MSE = \frac{1}{n}\sum_{1}^{n}(y_i - y_i')^2 \tag{11}$$

In theory, deep Q-learning should significantly enhance performance. Each state follow a Markov property and with a neural network to estimate the Q-values of each state the agent should be able to better navigate through the environment assuming the network is well trained.

I. Summary

Overall, this report and the code associated with it has successfully addressed almost all of the points outlined in the brief in some detail. Due to time constraints, no implementation of deep Q-learning was attempted meaning no meaningful comparison of performance on this project could be completed, however the question has still been answered in a more generic sense, answering based on some initial research of what deep Q-learning is how it may have helped improve the performance of the solution should there have been sufficient time to implement this.

The solution contains extensive documentation, and details on how to interpret and use the available documentation to reproduce the results seen within this report will be discussed in due course. It is also clear that some aspects of the solution

that have been attempted do not work as expected as some of the results are not in-line with what we would expect to see. For example, the robot fails to learn over time to reach the location with the highest reward when there are multiple reward locations. Due to time constraints on this project, it was not possible to fix some of these problems as it is difficult to say without detailed analysis what is causing them, however they have been identified and adequately discussed in the relevant sections to ensure there is sufficient warnings for the reader that the results are not entirely representive of what we would expect to see.

Unfortunately, some of the figures may be slightly difficult to read. Every effort was made to ensure they are as clear as possible, however the solution was built and run in a cloud environment (via Google Colab) '*pyplot.savefig()*' function could not be used.

III. Running the solution

The solution has been provided as a Jupyter Notebook. The solution was developed in the Google Colab environment which allows for notebooks to be executed on Google's cloud servers [10], however the solution can be run in any environment which supports Python notebooks (.ipynb) files.

The notebook itself contains code cells as well as in-line comments to explain provide a detailed understanding of the design decisions, how the code works and the purpose of the various variables, including which ones to amend to run the solution with a different set of parameters, or alter the grid size etc. Please refer to the text blocks located above each code cell to understand the purpose of each code cell and any important details regarding the variables used which you may choose to amend, and then the in-line comments for further explanation on the design of the code. Due to the nature of the code, and that much of the documentation is included in text cells within the notebook itself, the full code is not included in an appendix in this document.

References

[1] Kaelbling, Leslie P.m Littman, Michael L., Moore, Andrew W. *Reinforcement Learning: A Survey*., 1996., Journal of Artificial Intelligence Research., doi:10.1613

[2] Brems. M, *Top highlight A One-Stop Shop for Principal Component Analysis*, April 2017, accessed: April 2020, available at: https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c

[3] Ellis, M, Adaptive Intelligence Lecture 3: Hebbian Learning Oja's Rule PCA, n.d.,

[4] Hassoun, Dr., *"MIT BOOK" 3.3 Unsupervised Learning*, n.d, unknown publisher, accessed: 25 March 2020, available at: https://neuron.eng.wayne.edu/tarek/MITbook/chap3/3_3.html

[5] Amidi, A., Amidi, S., *CS229 - Machine Learning: Unsupervised Learning cheatsheet*, n.d., accessed: 26 April 2020, available at: https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-unsupervised-learning

[6] kumar.r, S., *Principal Component Analysis: In-depth understanding through image visualization*, September 2019, accessed: 26 April 2020, available at: https://towardsdatascience.com/principal-component-analysis-in-depth-understanding-through-image-visualization-892922f77d9f

[7] White, C., *State Space Complexity Management In Reinforcement Learning*, March 2005, The Faculty of the School of Engineering and Applied Sciences University of Virginia, available at: https://pdfs.semanticscholar.org/7844/4ce69bee66e73147a52b7f2701ac229a2954.pdf

[8] Buchard, A., *What are techniques to reduce the search space in reinforcement learning?*, July 2017, available at: https://www.quora.com/What-are-techniques-to-reduce-the-search-space-in-reinforcement-learning

[9] Mnih, V., et al. *Human Human-level control through deep reinforcement learning.* Nature 518, 529-533 (2015). available at: http://dx.doi.org/10.1038/nature14236

[10] Google Colab., Google Research, n.d., https://colab.research.google.com/notebooks/intro.ipynb

[11] Choudhary, A., *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python*, April 2019, available at: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

[12] Tuyls, K., Maes, S., Manderick, B., *Reinforcement Learning in Large State Spaces*, Computational Modeling Lab, Department of Computer Science, Vrije Universiteit Brussel, n.d., available at: https://link.springer.com/content/pdf/10.1007%2F978-3-540-45135-8_27.pdf

[13] Dulac-Arnold, G., Evans, R., Sunehag, P., Coppin, B., *Reinforcement Learning in Large Discrete Action Spaces*, December 2015, available at: https://www.researchgate.net/publication/288059770_Reinforcement_Learning_in_Large_Discrete_Action_Spaces

[14] Zychlinski, S., *Qrash Course: Reinforcement Learning 101 Deep Q Networks in 10 Minutes*, Jan 2019, accessed: 24 May 202, available at: https://towardsdatascience.com/qrash-course-deep-q-networks-from-the-ground-up-1bbda41d3677

Appendix

A. Extract of code used to implement SARSA(λ)

```
# For action taken, increase e[action] by
    1
eligibility_trace[action] += 1
# Update e for all actions
eligibility_trace = eligibility_trace * (
    lambdas * gamma)
# Update Q-value function based on
    eligibility_trace
Q += ( (learning_rate * ((gamma * Q[action
    ]) - Q_old)) * eligibility_trace )
```

B. Code used for aiding exploration

```
greedy = (np.random.rand() > epsilon)
if greedy:
    # Pick the best action
    action = np.argmax(Q)
else:
    # Pick a random action
    action = np.random.randint(N_actions)
```