Reinforcement Learning with Taxi-Traveller Agent

CS7IS2 Project (2019/2020)

Ankit Taparia, Rocky Bilei, Siddhartha Bhattacharyya, Srijan Gupta, Tanmay Bagla

tapariaa@tcd.ie, bileir@tcd.ie, bhattasi@tcd.ie, guptasr@tcd.ie, baglat@tcd.ie
Trinity College Dublin

Abstract. Reinforcement learning mechanisms have been found to exist in the learning processes of humans as well as animals and has inspired researchers to create computer programs or *agents* that can *learn* optimal actions by interacting with their environment. Reinforcement learning is one of the most active fields in Artifical Intelligence research. We present an evaluation of 4 state-of-the-art reinforcement learning algorithms, namely, Q-learning, SARSA, Expected SARSA and Deep-Q-Network, with Random Search as the baseline. We tested these algorithms in the Taxi environment provided by the OpenAI Gym, across 5000 iterations. Our results demonstrate that the Expected SARSA algorithm outperforms the other algorithms in the Taxi environment and is also the much more consistent across varied sets of hyperparameters. We also observe that all algorithms significantly outperform the baseline, Random Search, with a minimum improvement of 1150% in average penalties.

Keywords: reinforcement learning, Q-learning, SARSA, Deep Q-learning

1 Introduction

Reinforcement Learning (RL) is a semi supervised machine learning technique in which an agent interacts with its environment (Figure 1) and takes appropriate actions to achieve its goal by maximising the rewards. The more an agent explores and interacts with the environment, the more it learns from its own previous experiences and actions undertaken thereby developing an expertise in performing its job. A RL problem is built upon certain essential concepts described below:

1. Agent: The program trained using a policy
2. Environment: The physical space in which an agent operates
3. Action: A move an agent makes to transit to next state
4. Rewards: Assessment of an action which can be either positive or negative

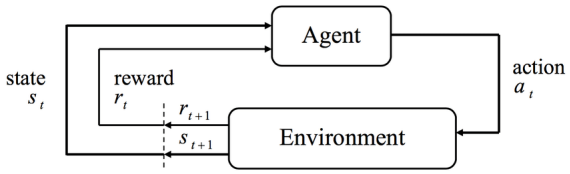

Fig. 1. The agent-environment interaction in reinforcement learning

In this paper various reinforcement learning algorithms are employed to solve a simulated real world problem of cab pickups and drops. The performance of agents trained using these different algorithms and the rewards obtained are evaluated and compared for analysis. The rest of the paper is organised as follows: Section 2 reviews the related work in the domain. We defined the problem and the algorithms used in Section 3. Section 4 discusses the experimental results and evaluation. Finally, Section 5 gives the concluding remarks.

2 Related Work

Osmankovic et al. [1] introduce the concept of Q-Learning as the quintessential algorithm to solve a-typical 'maze' problems using reinforcement learning. The paper highlights an action-reward methodology where the agent's actions are determined based on a knowledge-matrix that is iteratively updated. The methodology suffers the drawback of only utilizing the knowledge matrix to pick future actions, consequently, not allowing the agent to explore the state space.

Kantasewi, Marukatat et al. [2] approaches the same version of this problem with a more robust approach. They involve multiple knowledge matrices from which the agent learns iteratively about its environment with an aim to increase its overall reward score. The methodology proposed involves a dynamic ϵ value, which is initially initialized as a high value, and keeps reducing based on the agent's learned actions to provide a balance between 'exploration' and 'exploitation'. Our project includes an extension of this methodology.

In [3], Shih-Wei Lin compared three reinforcement learning (RL) algorithms to solve a maze problem by exploring the moving mode of mobile robot path planning. These three RL algorithms are Random Algorithm, Strategy Gradient Algorithm and SARSA (State Action Reward State Action). The best advantage of RL is that in the learning process, there is no right response except learning by reward messages.

Mnih et al. [4] demonstrated deep Q-Network (DQN), a novel reinforcement learning agent that uses a Convolutional Neural Network (CNN) as a non-linear function approximator for the action-value (also known as Q) function in Q-learning. The DQN agent was tested across 49 tasks from the Atari 2600 platform and was shown to outperform the best existing reinforcement learning method in 43 tasks while also delivering comparable performance to a professional human games tester across all 49 tasks [5].

3 Problem Definition and Algorithms

We address the Taxi problem using the OpenAI Gym toolkit [6]. There are six primitive actions and 500 state spaces in this environment. These include four navigation actions that move the taxi one square either North, South, East or West and pickup and dropoff. There are 4 possible pickup and drop positions (R,G,B,Y) as shown in Figure 2. The current pickup and drop locations are displayed in Blue and Purple respectively. There is a reward of -1 for each action and an additional reward of +20 for successfully delivering the passenger. There is a reward of -10 if the taxi attempts to execute the drop or pickup actions illegally. The taxi problem requires an algorithm that supports temporal abstraction, state abstraction and subtask sharing which makes Reinforcement learning an ideal candidate for solving the task [7].

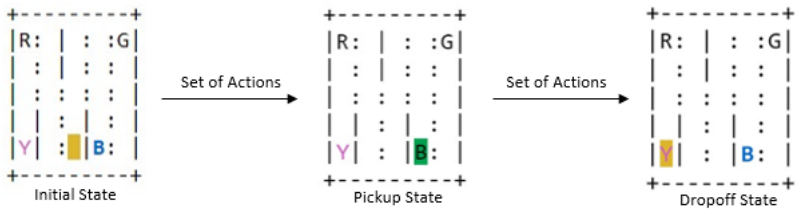

Fig. 2. Frames depicting environment states

To create a baseline we have attempted to solve the problem using 'Random Search', followed by various state-of-the-art algorithms with a collection of hyperparameter values to measure individual algorithm performance over a range of parameters.

3.1 Random Search

Random Search [8] also known as Random Walk is a local search technique that determines random paths to achieve the end goal of a problem comprising multiple actions and states. In Random Search Algorithm, we start at a given initial state, choose a random action from the available actions, transit to the next state; and then keep doing the same until the goal is reached. Which action the agent should choose can also be determined using probability distribution after running the algorithm multiple times to solve a specific problem and recording the observations. Its pseudo-code is given below.

The algorithm is easy to understand and implement as it does not require any domain specific knowledge and hyperparameters. Owing to its nature, it works well for problems with a small number of possible states/ search space. Since the next action is selected at random and independent of the previous state, there

Algorithm 1 Random Search

Require: Initial State S, Terminal State T and all available actions A
1: Initialize $current\_state \leftarrow S$
2: **repeat**
3: Choose a random action $a \in$A
4: Observe and record reward R
5: Transit to $next_state \leftarrow$S'
6: Update $current_state \leftarrow$next_state
7: **until** $current\_state \neq T$

is no need to save the states in the memory. This makes it memory efficient. For complex problems, which requires high computation to determine the next best possible action can leverage the use of Random Search Algorithm to overcome the computation limitations of the agent. The algorithm is asymptotically complete, however does not guarantee an optimal solution.

3.2 Q-Learning

The Q-Learning algorithm is an iterative offline learning algorithm where the environment's agent learns from the rewards and penalties it receives from the environment as it proceeds to achieve its target state. The algorithm maintains a global knowledge matrix Q-table containing these state-action value pairs can be denoted as Q(state,action). The Q-table is initialized to all zeros before training begins, and is iteratively updated using the following formula:

$$\text{Q(state, action)} \leftarrow Q(state, action) + (reward + maxQ(nextstate, actionspace))$$

Where, α is the learning rate parameter and $0 < \alpha \leq 1$. The parameter γ defines the discount level and $0 \leq \gamma \leq 1$. A high value for this parameter indicates that the agent should consider the long term benefits of an action instead of the short term benefits and vice versa. The importance of this parameter is that it enables the agent to select the shortest path to achieve its goal while minimising penalties. The algorithm essentially employs a 'look-ahead' approach as the Q-table is updated with the sum of the weighted value of the current state i.e. Q(state, action), with the sum of the reward received for taking the current step and the discounted maximum reward we might receive by taking the next step i.e. reward + max Q(next state,action space). Therefore we are always considering the step that has the maximum learned value in the knowledge matrix.

There are two ways for the agent to select an action at each iteration. To maintain an effective balance between exploitation and exploration a third parameter is introduced - ϵ where $0 \leq \epsilon \leq 1$. A smaller value of ϵ allows the agent to explore where we allow the agent to sample an action at random from the environment, this helps the agent to learn the environment surroundings, while a larger value allows the agent to be greedy and pick actions from the learned table that has the maximum value for that particular state. To attain a balance

the ϵ-greedy approach is introduced. The pseudo-code including all parameters are as follows:

Algorithm 2 Q-Learning

Require: Initial State S, Terminal State T, all available actions A, Q-table Q(s,a) for all state-action pairs | s ∈ S and a ∈ A , α, γ, ϵ

Initialize $current\_state \leftarrow S$
repeat
  if unif(0,1) $< \epsilon$ **then**
    $action \leftarrow$ sample action from environment
  else
    $action \leftarrow max\_action(Q(current\_state)$
  end if
  $next\_state$, $reward$, $final\_state \leftarrow$ result of action on environment
  $current\_value \leftarrow Q(state, action)$
  $next\_value \leftarrow max(Q(next\_state))$
  $Q(state, action) \leftarrow$ (1 - α) * $current\_value$ + α * ($reward$ + γ * $next\_value$)
  Update $current\_state \leftarrow next\_state$
until $current\_state \neq T$

3.3 SARSA (State Action Reward State Action)

The SARSA algorithm proposed in [9; 10], is an improved Q-learning algorithm. The SARSA method estimates the value of Q(st,at) by applying at in state s(t) according to the following updated formula:

$$\text{Q(S,A)} \leftarrow Q(S, A) + (\alpha)[\text{R} + \gamma * \text{Q(S',A')} - \text{Q(S,A)}]$$

This update is done after every transition from a nonterminal state st. Q(st+1,at+1) is defined as zero if st+1 is terminal. SARSA is an on-policy TD control method. A policy calculates the action taken on each state based on the maximum reward by estimating Q(s,a).If a state S is terminal (goal state or end state) then, Q (S, a) = 0 ∀ a ∈ A where A is the set of all possible actions.
Action is chosen as per Epsilon-greedy policy:

1. Generate a random number r ∈ [0,1].
2. If r $< \epsilon$ choose an action derived from the Q values (which yields the maximum reward).
3. Else choose a random action.

Algorithm 3 SARSA

Require: Initial State S, Terminal State T, all available actions A, Q-table Q(s,a) for all state-action pairs | s ∈ S and a ∈ A , α, γ, ϵ

Initialize $current\_state \leftarrow S$
Choose $current\_action$ from S' using epsilon greedy policy Q
repeat
  $actiona \leftarrow$ sample action from environment
  Take action a, observe reward R , $next\_state \leftarrow S'$
  Choose a' (next action) from S' using epsilon greedy policy Q
  $Q(S,a) \leftarrow Q(S,a) + (\alpha)[\mathrm{R} + \gamma * Q(S',a') - Q(S,a)]$
  Update $current\_state = next\_state$
until $current\_state \neq T$

3.4 Expected-SARSA

Expected SARSA is a variance of Q-Learning. Expected-SARSA is an on-policy algorithm since it updates its Q-values using the Q-value of the next state s' and the current policy's action a''. Expected SARSA selects the output of a state not on the Q-value but instead on the expected Q-value. The benefit of choosing the expected Q-value is that we reduce variance in the update from state to state. Having a lower variance means that we can increase α so that our algorithm learns faster. SARSA requires an $\alpha < 1$ while Expected-SARSA can have $\alpha = 1$. Since Expected-SARSA can take action-selection after the update, this is advantageous for returning actions such as what occurs in Taxi-v3. Importantly, Expected-SARSA has the same convergence guaranteed [11] as SARSA and thus can reach the optimal policy in the limit.
The hyperparameters for Expected-Sarsa are alpha, gamma, epsilon start, epsilon decay and epsilon cut. Choosing the right hyperparameters is important in reaching the highest reward possible. Alpha (α) is the most important hyperparameter. It affects how fast the algorithm converges but comes at a cost of a lower reward.

Algorithm 4 Expected-SARSA

Require: Initial State S, Terminal State T, all available actions A,Q-table Q(s,a) for all state-action pairs | s ∈ S and a ∈ A , α, β, γ

Initialize $current\_state \leftarrow S$
repeat
  Choose $current\_action \leftarrow a$ from state using policy π derived from Q
  Take action a, observe reward R , $next\_state \leftarrow S'$
  $Vs' \leftarrow \sum \pi(S',a) * Q(S',a)$
  $Q(S,a) \leftarrow Q(S,a) + (\alpha) * [\mathrm{R} + Vs' * Q(S',a') - Q(S,a)]$
  Update $current\_state = next\_state$
until $current\_state \neq T$

3.5 Deep Q-Learning

The algorithm used to train a deep Q-network has been described as a variant of the Q-learning algorithm in the original research paper. Q-learning selects optimal actions at each time-step using the action-value (also known as Q) function. The optimal Q-function follows an identity known as a *Bellman Equation* which is shown below.

$$Q^*(s,a) = E_{s' \sim \varepsilon}[r + \gamma \max_{a'} Q^*(s',a')|s,a] \tag{1}$$

Reinforcement learning algorithms estimate the Q-function using an iterative update $Q_{i+1}(s,a) = E_{s' \sim \varepsilon}[r + \gamma \max_{a'} Q_i(s',a')|s,a]$. The Deep Q-Learning algorithm uses a Convolutional Neural Network (CNN) to estimate the Q function, $Q(s,a;\theta) \approx Q^*(s,a)$, which is also known as a Q-network. A Q-network is trained by minimising the loss function,

$$L_i(\theta_i) = E_{s,a \sim \rho(.)}[(y_i - Q(s,a;\theta_i)^2] \tag{2}$$

at each iteration i, where $y(i) = E_{s' \sim \varepsilon}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a]$ is the target for iteration i. The weights of the Q-network are updated using stochastic gradient descent. Reinforcement learning algorithms have been known to be unstable with non-linear function approximators. To tackle this, the Deep-Q Learning algorithm uses Experience Replay [12], i.e., randomly sampling from past observations to reduce correlations in the training data. Pseudo-code below:

Algorithm 5 Deep Q-Learning with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for $episode = 1, M$ **do**
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = argmax_a Q(s_t, a; \theta)$
    Execute action a_t and observe reward r_t and state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
    Set

$$y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$$

    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  end for
end for

4 Experimental Results

The agent is trained with 5000 iterations or episodes using the each of algorithms described above. A single iteration in the experiment consists of certain epochs, which is the number of time-steps the agent took to reach the final state from the initial state, and a penalty count. The penalty count is the number of times the agent incurred a reward = -10. The observations of averaging over 5000 iterations are recorded in a table, and individual algorithm performances are shown in Figure 3. The algorithms are tested with a collection of hyperparameter values as shown in Table 1.

4.1 Random Search

Random search is applied to the problem to create a baseline performance for reinforcement learning algorithms mentioned above. The graph (Figure 3) is plotted which shows the total penalty count incurred by the agent for each iteration. As evident from the graph, the performance of the agent does not improve with iterations unlike RL algorithms. The reason being that the agent does not learn from its previous actions and the environment. Hence, it takes a very large number of steps to reach the terminal state.

4.2 Q-Learning

The Q-Learning is applied is to the problem and values of learning parameters are varied to study how agents behave in different situations. The experimentation produced distinguishable results for $\alpha = 0.1$ and 0.4, $\gamma = 0.6$ and 0.9. It is evident from Figure 3 that the exploiting ($\epsilon = 0.9$) the learned values helped the agent converge faster and steadily as compared to exploring ($\epsilon = 0.1$) and balanced ($\epsilon = 0.5$) approaches. Also the latter combinations of α and γ were found to be more effective in almost all algorithms.

4.3 SARSA

SARSA algorithm applied to the problem involved comparison of the outcomes using different parameter values of α, γ and ϵ. Different values of ϵ used are 0.9 (Exploitation), 0.1 (Exploration) and 0.5 (Less extreme). For parameter values of $\alpha = 0.1$ and $\gamma = 0.6$, Q-learning performed better than SARSA as it took more epocs to converge with high penalty. On the other hand, SARSA outperformed Q-learning for parameter values of $\alpha = 0.4$ and $\gamma = 0.999$ as evident from Figure 3.

4.4 Expected-SARSA

Another algorithm used to solve the problem is Expected-SARSA. As per the algorithms before, the hyperparameters α, γ and ϵ are varied and the convergence

and number of penalties are monitored. As seen in Figure 3 the algorithm has wild fluctuations at the start before stabilizing steadily to converge. This shows that the initialization of the α has a profound impact on the stability of the Expected-SARSA algorithm. We can also observe from Table 1 that the $\alpha = 0.4$ configurations have nearly 50% less penalties than when $\alpha = 0.1$. Finally, when comparing the average epochs and average penalties of Expected-SARSA with SARSA and Q-Learning we can see that Expected-SARSA has the lowest number of penalties incurred.

4.5 Deep Q-Network

The original algorithm was configured for Atari 2600 games and received successive frames (static images) of the game as input and used a CNN for the Q-network. We modify the algorithm to take as input a one-hot vector representing one of 500 possible states of the taxi environment.Since the input is a 1-D vector, linear feed-forward layers are used instead of convolutional layers. The implementation contains 1 hidden layer of 150 neurons as well as a Dropout layer. Its performance is depicted in Figure 3. A few other parameter settings were also tried but yielded much lower performance than the parameters shown in Table 1.

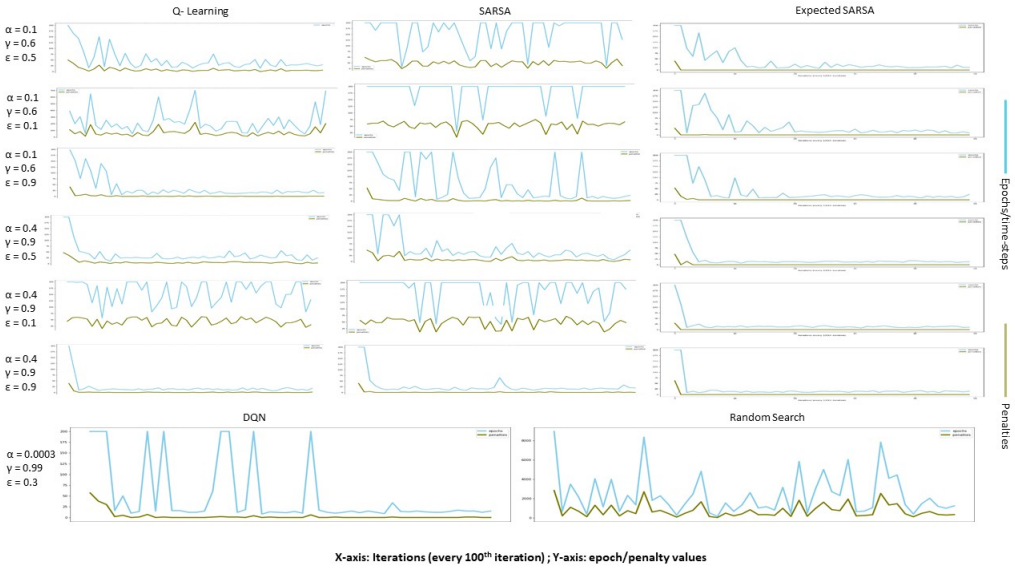

Fig. 3. Performance Graphs: Iterations vs Epochs

5 Conclusions

Our comparative analysis led us to believe that the Expected-SARSA algorithm worked best giving the least average time steps for the agent to reach its final destination, followed by Q-Learning, DQN and SARSA. Comparing these state-of-the-art algorithms with our baseline Random Search results clearly shows immense improvement. Moreover, it can also be observed better results are achieved using the exploitation approach as compared to the exploration. Thus, it is prudent to give more weightage to the iteratively learned values as compared to letting the agent explore the environment.

Algorithm	Epsilon	Alpha	Gamma	Average Epochs	Average Penalties
Q-learning Balanced	0.5	0.1	0.6	54.4146	8.5604
		0.4	0.999	35.0344	5.4728
Q-learning Exploit	0.9	0.1	0.6	38.7876	1.4156
		0.4	0.999	222.867	63.0116
Q-learning Explore	0.1	0.1	0.6	247.5218	70.4168
		0.4	0.999	222.867	63.0116
SARSA Balanced	0.5	0.1	0.6	49.084	7.8396
		0.4	0.999	152.8246	22.9844
SARSA Exploit	0.9	0.1	0.6	80.2608	2.988
		0.4	0.999	22.346	1.056
SARSA Explore	0.1	0.1	0.6	191.6582	55.1742
		0.4	0.999	172.191	49.893
Expected SARSA Balanced	0.5	0.1	0.6	36.1002	0.4164
		0.4	0.999	17.68	0.3212
Expected SARSA Exploit	0.9	0.1	0.6	36.1648	0.4256
		0.4	0.999	17.6968	0.3274
Expected SARSA Explore	0.1	0.1	0.6	36.1956	0.4142
		0.4	0.999	17.7486	0.2948
DQN (Deep Q Network)	0.3	0.3	0.99	47	2
Random Search	-	-	-	2486.424	805.074

Table 1. Results observed for algorithms for different values of hyperparameters

Bibliography

[1] D. Osmanković and S. Konjicija, "Implementation of q—learning algorithm for solving maze problem," in *2011 Proceedings of the 34th International Convention MIPRO*, pp. 1619–1622, IEEE, 2011.

[2] N. Kantasewi, S. Marukatat, S. Thainimit, and O. Manabu, "Multi q-table q-learning," in *2019 10th International Conference of Information and Communication Technology for Embedded Systems (IC-ICTES)*, pp. 1–7, IEEE, 2019.

[3] S.-W. Lin, Y.-L. Huang, and W.-K. Hsieh, "Solving maze problem with reinforcement learning by a mobile robot," in *2019 IEEE International Conference on Computation, Communication and Engineering (ICCCE)*, pp. 215–217, IEEE, 2019.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Open ai gym: A toolkit for developing and comparing reinforcement learning algorithms. https://gym.openai.com."

[7] T. G. Dietterich, "Hierarchical reinforcement learning with the maxq value function decomposition," *Journal of artificial intelligence research*, vol. 13, pp. 227–303, 2000.

[8] F. Spitzer, *Principles of random walk*, vol. 34. Springer Science & Business Media, 2013.

[9] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *Advances in neural information processing systems*, pp. 1038–1044, 1996.

[10] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[11] H. Van Seijen, H. Van Hasselt, S. Whiteson, and M. Wiering, "A theoretical and empirical analysis of expected sarsa," in *2009 ieee symposium on adaptive dynamic programming and reinforcement learning*, pp. 177–184, IEEE, 2009.

[12] L.-J. Lin, "Reinforcement learning for robots using neural networks," tech. rep., Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.