Practical 4: Reinforcement Learning

Meriton Ibrahimi
meritonibrahimi@college.harvard.edu

May 7, 2019

1 Introduction

In this practical, we set out with a mission to develop an agent which can navigate through the game *Swingy Monkey*, a game similar to the once very popular *Flappy Bird*. In the game, the agent controls a monkey that is trying to swing on vines and avoid tree trunks. The agent can either make the monkey jump to a new vine, or have the monkey swing down the vine they currently hold. Successfully passing tree trunks without hitting them keeps the game going and increases the game score, while falling off the bottom of the screen, jumping off the top, or hitting the tree trunks all end the game. Our goal is to use concepts from the field of machine learning known as Reinforcement Learning (RL) to train an agent which can learn the game. A good agent will be able to achieve a high score repeatedly.

2 Technical Approach

In order to begin our RL approach, we need to define our environment. Within the game, a number of parameters are important to defining the environment. We define the set of possible actions, state space, and rewards below.

2.1 Actions

At any time point in game, the agent can have the monkey perform one of two possible actions: do nothing and continue swinging down on the current vine, or jump to a new vine.

2.2 State Space

We define the state space S in which the monkey resides with four parameters, effectively making each state space four dimensional. The state parameters are:

1. distance from the monkey to the next tree trunk
2. distance from the top of the monkey to the top of the tree trunk
3. distance from the bottom of the monkey to the bottom of the tree trunk
4. gravity level of the game

The last parameter defined – gravity $\in (1, 4)$ – is an important parameter affecting how the monkey jumps, as we would expect. At time step two, we calculate gravity by taking the difference in the monkey's velocity between time steps one and two – acceleration is defined as a change in velocity over time. At time step one, when we do not know the gravity, we enforce that the monkey do nothing. Playing the game ourselves, we see that monkey cannot always recover if the first action is to jump, but we can recover if we do nothing.

2.2.1 Discretize States

The effective sizes of each of the state space parameters above are on the order of 600, 400, 400, and 2, respectively. This tells us that the number of states is roughly on the order of 200 million. As this is an incredible number of states with which to work, we move to discretize the states and reduce the dimensionality of each parameter. We chose to divide each distance parameter by 20 and discretize into bins, effectively reducing the size of the state space to roughly 200 thousand.

2.3 Rewards

To train our RL agent – to teach the agent how to play the game – we need rewards at each state space and action to inform decision making. Passing a tree trunk, hitting a tree trunk, failing off the bottom of the screen, or jumping off the top of the screen are given rewards +1, -5, -10, and -10, respectively; otherwise, the reward is 0.

2.4 RL Model Building

RL algorithms are divided into two categories, model-based and model-free. In model-based approaches, we look to directly estimate the transition and rewards functions. A transition function is a probabilistic mapping from a state-action pair to the next state, and a reward function is a mapping from a state-action pair to a reward scalar. In a model-based algorithm, the agent hopes to be able to predict the dynamics of the environment with estimates of the transition and reward functions. These functions, however, can be difficult to approximate – especially in a high-dimensional problem as we have here – and hence, the optimal policy might never be found. Model-free methods, on the other hand, estimate the optimal policy without directly using or looking to understand the dynamics of the environment. Essentially, model-free methods look to draw upon a collection of real-world examples to provide a memory on how to optimally act. Two predominate model-free methods are SARSA and Q-Learning. We chose to use these two methods for our approach.

2.4.1 Overview of Model-free Methods

SARSA and Q-Learning are similar model-free methods. Both methods look to estimate a state-action-value function, $Q(s, a)$, which is the expected return starting from state s having taking action a; Both methods go about this estimation by utilizing a similar learning objective. We utilize the Q-value form of the Bellman equations to construct a loss-function for estimating the Q-value weights:

$$\zeta(\mathbf{w}) = \frac{1}{2}\sum_{s,a}\left(Q(s,a;\mathbf{w}) - \left[r(s,a) + \gamma\sum_{s'} p(s'|s,a)\max_{a'} Q(s',a';\mathbf{w})\right]\right)^2 \tag{1}$$

where $r(s, a)$ is the reward seen at state s having taken action a and γ is the discoutning factor for the future. Taking the gradient of the sampled loss $\zeta^{(s,a)}(\mathbf{w})$ (assuming $s' \neq s$) and sampling the next state to approximate the expectation $s' \sim p(s'|s, a)$. we get:

$$\frac{\partial \zeta^{(s,a)}(\mathbf{w})}{\partial w_{s,a}} \approx Q(s, a; \mathbf{w}) - \Bigg[r(s, a) + \gamma \max_{a'} Q(s', a'; \mathbf{w}) \Bigg]. \tag{2}$$

The gradient allows us to iteratively update the Q-value weights for SARSA and Q-Learning. For SARSA, the update becomes:

$$w_{s,a} \leftarrow Q(s, a; \mathbf{w}) - \eta \Bigg(Q(s, a; \mathbf{w}) - \Bigg[r(s, a) + \gamma Q(s', \pi(s'); \mathbf{w}) \Bigg] \Bigg) \tag{3}$$

where η is the learning rate and $\pi(s')$ is the action taken in state s' according to Q-valued derived policy π. For Q-Learning, the update becomes:

$$w_{s,a} \leftarrow Q(s, a; \mathbf{w}) - \eta \Bigg(Q(s, a; \mathbf{w}) - \Bigg[r(s, a) + \gamma \max_{a'} Q(s', a'); \mathbf{w}) \Bigg] \Bigg). \tag{4}$$

From the latter two equations, we see that SARSA and Q-Learning are very similar but differ in how a' is chosen in the update step. Q-Learning always updates with the maximum Q-value over possible actions, while SARSA chooses a' from the Q-value derived policy. For our RL training, we fit both SANSA and Q-Learning agents.

2.4.2 Policy: Exploitation versus Exploration

If a RL algorithm is exploitative, given a state s the agent always chooses the action a which maximizes the Q-value function. An exploitative agent, however, may not always arrive at the global, best policy. Further, exploration helps improve Q-value estimation by allowing the agent to learn its surrounding environment. An alternative to a purely exploitative approach will escape the Q-value maximization with a certain, fixed probability and chose a uniformly random action over the action space. This is know as an ϵ-greedy algorithm/agent:

$$\begin{aligned} & z \sim Uniform(0, 1) \\ & \pi(s) = \begin{cases} \text{random} a \in A & z < \epsilon \\ \arg\max_{a'} \quad Q(s', a'); \mathbf{w}) & \text{else} \end{cases} \end{aligned} \tag{5}$$

A modification to ϵ-greedy SARSA and Q-Learning is to have the value of ϵ decrease over time with the number of epochs. Intuitively, after having explored and after having gained significant experience, we would like the agent to settle on what it has learned.

3 Results

We implemented ϵ-greedy SARSA and Q-Learning agents for several combinations of the discounting factor γ, learning rate η, greedy parameter ϵ, and whether to decay ϵ over time. Each agent was trained over 500 epochs. The results for the SARSA and Q-Learning agents are shown in the below table. For each agent, we list the best score achieved. Further, we list the average score

SARSA Agent	Decreasing ϵ	Max Score	Average Score (all training epochs)	Average Score (after burn-in)
$\gamma = 0.6, \eta = 0.1, = 0$	False	482	16.06	24.74
$\gamma = 0.6, \eta = 0.2, = 0$	False	262	12.456	21.815
$\gamma = 0.8, \eta = 0.2, = 0$	False	755	26.002	54.94
$\gamma = 0.8, \eta = 0.2, = 0.1$	True	114	5.94	10.12
$\gamma = 0.9, \eta = 0.1, = 0$	False	549	20.988	35.11
$\gamma = 0.9, \eta = 0.1, = 0.1$	True	33	1.44	1.59
$\gamma = 0.9, \eta = 0.1, = 0.05$	True	198	8.612	16.63
$\gamma = 0.9, \eta = 0.2, = 0$	False	1152	41.222	93.485
$\gamma = 0.9, \eta = 0.2, = 0.1$	True	33	2.738	2.505
$\gamma = 0.9, \eta = 0.2, = 0.05$	False	13	0.618	0.3

Q-Learning Agent	Decreasing ϵ	Max Score	Average Score (all training epochs)	Average Score (after burn-in)
$\gamma = 0.7, \eta = 0.2, = 0.1$	False	11	0.642	0.31
$\gamma = 0.9, \eta = 0.1, = 0$	False	766	26.86	60.97
$\gamma = 0.9, \eta = 0.2, = 0$	False	440	21.688	43.78
$\gamma = 0.9, \eta = 0.2, = 0.1$	True	101	6.746	11.905
$\gamma = 0.9, \eta = 0.2, = 0.1$	False	8	0.6	0.37
$\gamma = 0.9, \eta = 0.2, = 0.05$	True	102	6.082	6.965
$\gamma = 0.9, \eta = 0.2, = 0.05$	False	7	0.628	0.355

over the 500 epochs in addition to the average score over the last 200 epochs, considering the first 300 epochs to be a burn-in period. For the SARSA agents, we see that the clear winner is the exploitative agent where $\gamma = 0.9, \eta = 0.2$, and $\epsilon = 0$, achieving a maximum game score of 1152. For the Q-Learning agents, we see that the clear winner is the exploitative agent where $\gamma = 0.9, \eta = 0.1$, and $\epsilon = 0$, achieving a maximum game score of 766.

4 Discussion

We see from the results that both the SARSA and Q-Learning agents learn *Swingy Monkey* fairly well. In training, we saw that the SARSA agents tended to converge to higher game scores faster than the Q-Learning agents, as we would expect from the literature. Further, across the two types of agents, we see that the adaptive ϵ-greedy method was a significant improvement over the fixed method. The best agent across the two RL algorithms was the SARSA agent which achieved a greater than 1000 game score multiple times and which achieved relatively good scores both quick into training and after burn-in. It is noted, however, that this agent, along with all the other agents, had considerable variability in performance even after burn-in. We attribute this latter observation to the discretization of the state space. We hypothesize that because we discretize the states by a factor of 20 that the agents are, over the epochs, re-learning and overwriting states in which the non-discrete states overlap in the discrete mapping. Moving into the future, we would look to increase the size of the state space we hold in order the remedy the latter problem. Additionally, we would look into using neural network architecture to remove the need of storing a table of Q-values by deploying a deep Q-Learning model as seen in famous examples such as AlphaGo.