Reinforcement Learning: Assignment 2

Kristan Papirio s1674939

March 28, 2017

1 Actor-Critic Architecture

Temporal-difference (TD) based reinforcement learning are prediction based methods that are a combination of Monte Carlo and Dynamic Programming problems. As a result, such methods directly learn from experience and update estimates by bootstrapping. The actor-critic architecture within temporal difference learning is a on-policy method in which the policy is not dependent on the value function. It is based on the functionality of two main components: the actor and the critic. The actor is used for action selection, as it is the actual representation of the policy structure. The critic is then used to estimate the value function by criticizing the actions and current policy chosen by the actor. Its form is that of a TD error, which is the difference between the expected state value and the actual state value. If the evaluated error is positive, the policy is modified so that more weight is given to the future selection of that given action. The converse is true for a negative error. The learning that takes place within this architecture is dependent on this scalar signal that is outputted by the critic and sent to the actor to inform future actions. This is the only interaction that takes place between the two, which is how they are able to remain as independent components. There are many different variations of these methods that differ in the techniques used for action selection, include additional parameters to determine the weight given to a corresponding action, etc. However, one of the trade-offs of these methods is that improvement steps can actually make things worse. In addition, approximating gradient policy can introduce bias that may result in the correct solution not being reached, but the utilization of careful function approximations can be used to combat this.

An example of one application of the actor-critic architecture is the Asynchronous Advantage Actor-Critic (A3C) algorithm that was developed by Google's DeepMind for motor control tasks and 3D maze environments. A3C has the ability to operate in both discrete and continuous spaces and can train feed-forward and recurrent agents [2], by exploiting asynchronous parallel workers. The actor-critic architecture allows the algorithm to use exploration to select an action up to a certain number of steps into the future. The agent then receives less than or equal to this number of rewards, and updates the value function and policy for each state-action pair and corresponding reward. As a result, A3C is more simplistic, faster, and performs better on tasks than previously developed learning agents, such as Deep Q-Network (DQN). In addition, the A3C algorithm has been used to play Atari games, TORCS car racing simulator, Mujoco physics engine simulators, and Labyrinth. Even companies like OpenAI have adapted A3C as the starter agent for their environments.

Like the actor-critic architecture, the Sarsa algorithm is also an on-policy TD method. Sarsa is a Markov chain with a reward process that learns the action-value function by taking into account each state-action transition. Updates are made to the function for the given policy following each of these transitions to a non-terminal state. Throughout the updating process, the policy is greedily changed with respect the action-value function. This is similar to how the actor responds to the approximate value function given by the critic to change the current policy being followed to a greedy policy. Because both evaluation and updates occur at each step, Sarsa is known as a generalized policy iteration. However, unlike the actor-critic method, learning is dependent of the action-value function. As the action-value function is updated, the policy is updated in a corresponding manner because of this dependence. Due to the disjoint way that the actor-critic method operates, the critic can make an update to the value function without affecting the actor in any way. With Sarsa, optimal convergence of the function is guaranteed with a probability of 1,

just as the state values under TD(0), which is the simplest TD method. Although the actor-critic methods may do so in a different way, it also has desirable convergence properties as long as it is gradient based [1].

Research has shown that, despite a deviation to Sarsa, Q-Learning, and other methods that learn from estimated values, the actor-critic architecture maintains its relevance and usefulness through two resulting advantages. The first is the benefits of computational cost reduction. Continuous state spaces present a problem by requiring algorithms to iterate through all possible actions. However, this is not a necessity if the policy can be explicitly stored, as with actor-critic. The second is that by learning the optimal values of selecting actions, actor-critic methods have the ability to learn stochastic policies. In addition, it is often also easier to impose domain-specific constraints on a group of policies.

2 RL with Function Approximation

2.1 Construction of Feature and Parameter Vectors

Typically, value function estimates are represented as a table that consists of one entry for each state-action pair. However, these tabular cases are restricted to tasks with a small number of state-action pairs because of the time and information required for construction, which can be problematic for spaces with continuous variables, visual images, or other complex sensations. This requires the use of generalization to generate an approximation based on previously experienced states in a subset of the state space and means that not all states visited will have been previously experienced. Linear function approximation is one example of such an approach, which tries to generalize examples from the desired function in order to construct an approximation of the entire state-action value function. Let the linear approximation of the $Q_t(s, a)$ state-action value function at time t be given by:

$$Q_t(s, a) = \Theta_t^T \Phi_{s,a} = \sum_{i=1}^{n} \Theta_t^i \Phi_{s,a}^i,$$

where Θ_t is the parameter vector and $\Phi_{s,a}$ is the feature vector. Since $Q_t(s, a)$ is dependent on Θ_t and $\Phi_{s,a}$, the vectors should be constructed in such a way that the generalization is as close as possible to the tabular case. If done correctly, linear approximation can be very efficient as far as the data and computational cost required. Therefore, features should be chosen that are appropriate and correspond to the natural features of the task. Not only will this allow for the generalization of those that are most important, but it will add prior domain knowledge to the system. Combinations and conjunctions of feature values also need to be accounted for because the linear nature of the method does not allow for the representation of interactions between features. In practice, this can be done by constructing features that represent each of the state-action pairs in the tabular case, such that each pair has an index on an array and there is a corresponding weight vector of the same length. If a feature, $\Phi_{s,a}$, is binary, it can be given a value of 1 when at state-action pair (s,a) and 0 otherwise, accounting for the gradient of a weight at a given update. Although not every state has been exactly experienced, if a sufficient amount of appropriate features are used, the values of the original table are reproduced by the linear approximation method.

2.2 Implementation

The feature vectors below are based on the sensing capabilities of the agent and are used to account for the following behaviors: collision avoidance, moving faster to pass more cars, and staying in the center of the road whenever possible.

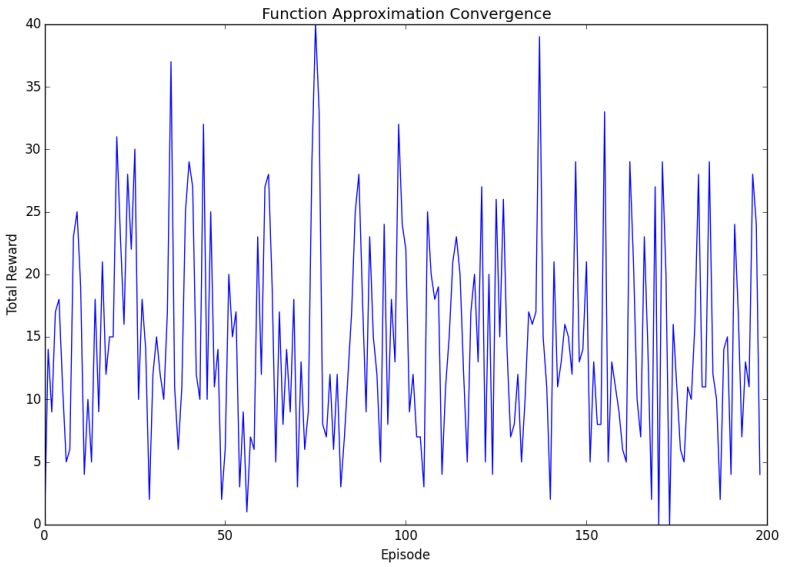
$F_1(s,a)$	1- if agent is accelerating and there is a car in front 0- otherwise
$F_2(s,a)$	1- if agent not accelerating and no car in front 0- otherwise
$F_3(s,a)$	1- if agent is accelerating and speed is greater than or equal to 25 0- otherwise
$F_4(s,a)$	1- if agent is not accelerating and speed is less than 25 0- otherwise
$F_5(s,a)$	1- if agent is accelerating and speed is less than 25 0- otherwise
$F_6(s,a)$	1- if agent is accelerating 0- otherwise
$F_7(s,a)$	1- if agent moves right 0- otherwise
$F_8(s,a)$	1- if agent moves left 0- otherwise
$F_9(s,a)$	1- if agent moves right and car ahead to the left 0- otherwise
$F_{10}(s,a)$	1- if agent moves left and car ahead to the left 0- otherwise

Features 1, 2, 9, and 10 are designed to test the likelihood of a collision, when paired with the requirement to move faster in order to pass cars or choosing and alternative action; features 3, 4, 5, are designed to test the likelihood of passing a car based on current speed and action; and features 6, 7, 8 are designed to test the favorability of each action.

2.3 Analysis

2.3.1 Learning Curves

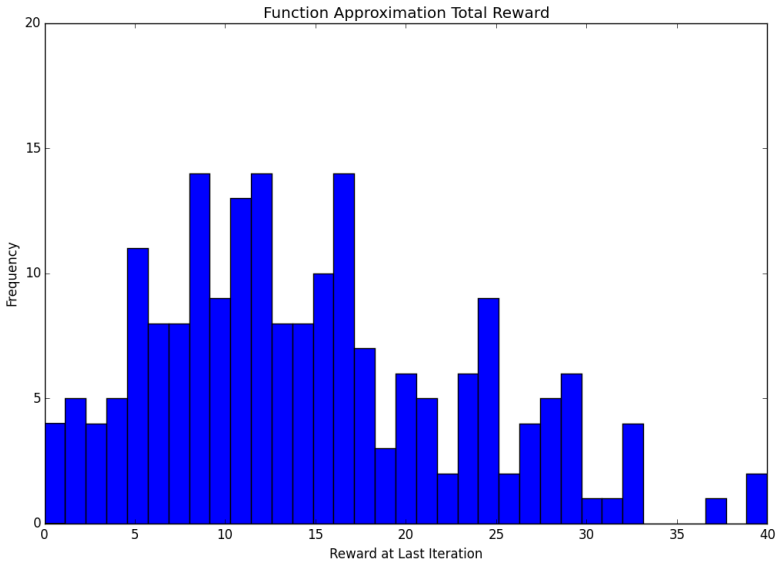
In the event of a repeated number of iterative episodes, learning curves are used to represent the increase of learning as experience progresses. Below are the learning curves for the newly implemented function approximation agent and the previously used Q-Learning agent. The Q-Learning algorithm that was implemented for the sake of this comparison is the sample solution that was distributed following the marking period.



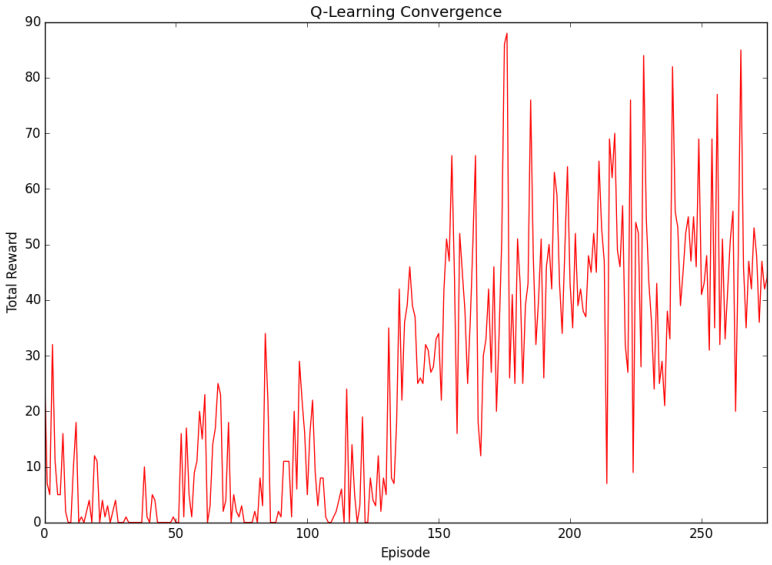
Over the course of 200 episodes consisting of 6500 iterations each, the Function Approximation

algorithm experiences only a slight increase in learning. In general, the total reward produced per episode is relatively consistent, as the algorithm learns quickly which features prove most useful and exploits them for the remainder of the run. The resulting weights of each corresponding feature are explained in the subsequent section.

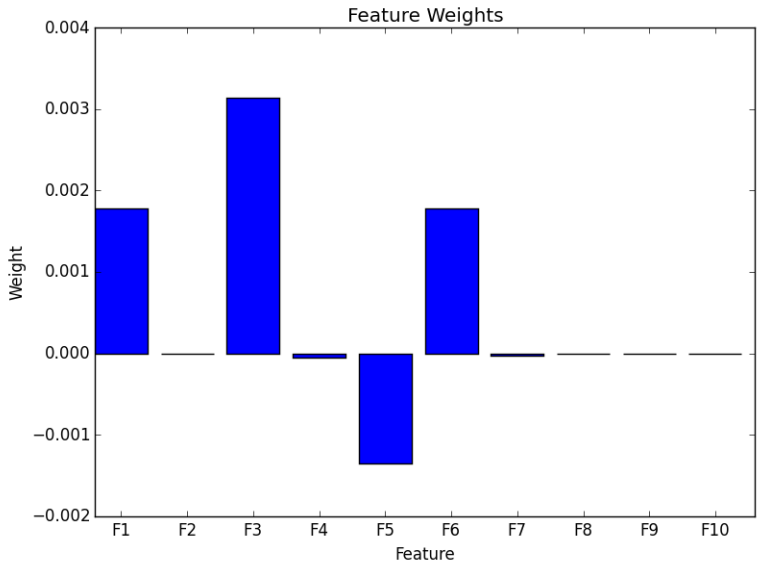
In addition, the function approximation algorithm experiences quite a bit of variation between total reward at the end of each episode, as can be seen below. In general, most of the maximum reward per episode values are seen throughout the duration of the episode, not directly at the end.



In contrast, the Q-learning algorithm experiences the majority of its learning later on. The graph below shows the resulting reward at the last iteration over the course of nearly 300 episodes. It is not until approximately episode 130 that a substantial increase occurs. It is clear that the Q-Learning algorithm benefits much more from and extended learning period than the Function Approximation algorithm. In addition, once this behavior is learned the level of reward remains relatively consistent as the agent progressed in episode.



2.3.2 Feature Inspection



Based on the graph above, the four features with the highest weights were:

$F_1(s, a)$: if agent is accelerating and there is a car in front
$F_3(s, a)$: if agent is accelerating and speed is greater than or equal to 25
$F_5(s, a)$: if agent is accelerating and speed is less than 25
$F_6(s, a)$: if agent is accelerating

It can be noted that all features inform whether or not the agent is accelerating. It can be concluded then that this is the characteristic that the agent learned results in the most reward, and can confirmed by knowledge of the functionality of the game. The only way that an agent receives a reward is if it is able to pass an opponent's car, which requires that the chosen action must be acceleration, as that is the only way in which the agent will be moving faster than an opponent. In addition, feature 3 shows that the agent learned that maximum acceleration is the most beneficial. In contrast, feature 5 shows that the early stages of acceleration do not yield a reward, so the collisions that create such a situation should be avoided.

2.3.3 Convergence Rate Comparison

A convergence rate is the speed at which a given function approaches its limit. In terms of this reinforcement learning task, this occurs when maximum learning has occurred and the learning curve experiences a plateau. The convergence of both the Function Approximation and Q-Learning algorithm can be seen in the graphical representations of the learning curves above. It is clear that the Q-Learning method suffers from slow convergence. Not only does the algorithm require about 125 episodes before it begins to converge and produce a consistent range of reward values, but it happens slowly. This can be seen in the period from about episode 125 to 200. In contrast, the Function Approximation algorithm has a very high rate of convergence, as very few iterations are needed to reach a useful approximation. It does not take long for the agent to learn the value of each feature and the weight given to possible scenarios and actions. As a result, it is able to produce a consistent range of reward values across almost all 200 iterations because convergence occurs in approximately the first 10.

References

[1] Konda, V.R. and Tsitsiklis, J.N., 1999, November. Actor-Critic Algorithms. In NIPS (Vol. 13, pp. 1008-1014).

[2] Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T.P., Harley, T., Silver, D. and Kavukcuoglu, K., 2016, February. Asynchronous methods for deep reinforcement learning. In International Conference on Machine Learning.