Introduction

We used mountain-car problem in order to compare three different function approximation techniques. In the following three sections, we discuss the result of applying Sarsa(\(\lambda\)) with Tile Coding, Actor-Critic with linear function approximation and Sarsa(\(\lambda\)) with neural network on the mountain car task. To make the results comparable, we start the car from the bottom in all experiments. Furthermore, the maximum number of the actions in a trial is set to 5000 for the first two algorithms and to 10000 for the last one.

Sarsa(\(\lambda\)) with Tile Coding

We started our experiments with Sarsa(\(\lambda\)) with ten \(9 \times 9\) tilings. We modeled the task as undiscounted episodic task and set the parameters at \(\lambda = 0.9, \epsilon = 0, \alpha = 0.05 \frac{1}{m}\), where \(m\) is the number of tilings. We also initialized all initial action values to zero and used accumulating eligibility traces. Figure 1 shows the negative of value function learned after 100 and 1000 episodes. Comparing to the Figure 8.10 in the book, we got quiet the same results as we set the parameters at the same values as discussed in the book.

Then, we studied the effect of parameters \(\alpha\) and \(\lambda\), using accumulating trace, on the rate of learning for the mountain-car task. Figure 2 shows the result averaged over first 20 trials and 30 runs. It suggests that setting parameters to \(\alpha = 0.14\) and \(\lambda = 0.3\) will lead to a better learning process at least in the beginning of the task.

Natural Actor-Critic

Natural Actor-Critic algorithms are novel reinforcement learning algorithms in which actor updates the policy parameters using stochastic gradient descent while critic estimates the value parameters using temporal difference learning. For this section, we implemented the Regular-Gradient Actor-Critic algorithm introduced in [1].
Figure 2: The effect of $\alpha$ and $\lambda$ on early performance on the mountain-car task.

\begin{align*}
\pi(s, a) &= \frac{e^{\theta^T \phi_s}}{\sum_{a' \in A} e^{\theta^T \phi_{s, a'}}} \\
\psi(s, a) &= \frac{\nabla_{\theta} \pi(s, a)}{\pi(s, a)}
\end{align*}

Figure 3: Regular Gradient Actor-Critic algorithm introduced in [1].

1: Input:
- Randomized parameterized policy $\pi(\cdot; \theta)$.
- Value function feature vector $f(s)$.

2: Initialization:
- Policy parameters $\theta = \theta_0$.
- Value function weight vector $v = v_0$.
- Step size $\frac{\beta}{\alpha}$.
- Initial state $s_0$.

3: for $t = 0, 1, 2, \ldots$ do

4: Execution:
- Draw action $a_t \sim \pi(a_t | s_t; \theta_t)$.
- Observe next state $s_{t+1} \sim p(s_{t+1} | s_t, a_t)$.
- Observe reward $r_{t+1}$.

5: Average Reward Update:
- $A_{t+1} = (1 - \gamma_t) A_t + \gamma_i r_{t+1}$.

6: TD error:
- $\delta_t = r_{t+1} + \gamma \max_{a'} \pi(a' | s_{t+1}; \theta_t) - v_t^f(s_t)$

7: Critic Update:
- $v_{t+1} = v_t + \alpha_t \delta_t f(s_t)$.

8: Actor Update:
- $\theta_{t+1} = \theta_t + \beta_t \delta_t \psi(s_t, a_t)$.

9: endfor

10: return Policy and value-function parameters $\theta, \nu$

Figures 3 shows the algorithm used for our implementation. Note that we removed the average reward form all the equations, because we are using the algorithm for the episodic task. Furthermore, we set the step size parameters for actor ($\beta$) and critic ($\alpha$) at constant values.

In order to fairly compare the results with the previous section, we plot the number of time steps required to finish the task for different range of values of $\tau = \frac{\beta}{\alpha}$. The results which are shown in Figure 4 suggest that $\alpha = 0.05$ and $\beta = 0.03$ can be considered as a good potential set of parameters. Note that this is not necessarily the best set of values for $\alpha$ and $\beta$, because we only considered the first 30 trials and 20 runs to draw the plot.

Figure 5 compares the best settings of Sarsa($\lambda$) and Actor-Critic algorithms. The results are obtained as the average value of 20 runs. The results show that Sarsa($\lambda$) outperforms Actor-Critic; both in number of episode that it takes to converge and the value that it converges into. To be more precise, Sarsa($\lambda$) converges after 550 episodes and it takes about 104 time steps to finish one episode after convergence. On the other hand, Actor-Critic converges after about 1400 episodes and it takes between 140-145 time steps for it to finish the trial after convergence.

As we mentioned before, Actor-Critic algorithm selects the actions stochastically. In other words, it is not necessarily selecting the action with the highest action-value; this is similar in essence to $\epsilon$-greedy algorithm. Therefore, the results of Actor-Critic methods are noisier comparing with Sarsa($\lambda$). This noise even exists when the Actor-Critic algorithm converges. Note that we removed the first 75 episodes of the experiments in Figure 5 to better show this behavior.
Figure 4: The effect of $\alpha$ and $\beta$ on early performance on the mountain-car task.

Figure 5: Comparison of convergence time for Sarsa($\lambda$) and Actor-Critic

**Sarsa($\lambda$) with Neural Network**

As Sarsa($\lambda$) worked well on the mountain car problem, we tried to apply a similar algorithm with non-linear function approximation for this task. We used back-propagation neural network as our non-linear function approximation. Note that for the rest of this report, we will refer to this approach as non-linear Sarsa.

The neural network is modeled as a 3 layered network. The input layer is consisted of two nodes, one for position and the other for velocity. Hidden layer is consisted of 10 hidden nodes and it uses a sigmoid function while the output layer is linear and consisted of one node. One bias node is also added to input and hidden layers.

We used the neural network as the state value function. Furthermore, a weight node and an eligibility trace is defined corresponding to each node in the hidden and output layer. Note that the updates for the eligibility traces are done based on the gradient of the value function in each layer. There are also two parameters for the learning rate, one for the hidden layer and the other for the output layer.

Figure 6 shows the result of applying non-linear Sarsa on the mountain car problem. The parameters were set at $\gamma = 1$, $\lambda = 0.9$, and $\epsilon = 0.01$. The learning rates for the hidden and output layers were set at 0.2 and 0.1.

The figure shows a normal learning behavior in the first 300 trials. Then, it constantly oscillates between maximum number of actions per trial, 10000, and some values near 200. We suggest that either our setting for learning rates or the $\epsilon$—greedy policy caused this result. However, we also consider a small probability for a bug in our implementation. In order to test our hypothesis, we set the epsilon to zero after 300 time steps. We also tried to
decrease the learning rates over the time using either of $lr_t = \frac{lr_0}{\sqrt{t}}$ or $lr_t = \frac{1000 \times lr_0}{1000 + t}$. Unfortunately, we found none of them successful on changing the behavior. We also tried to study the different values for learning rates on the initial episodes, but this attempt was also failed due to the fact that all different settings lead to almost similar results.

There are numerous number of related works which discussed using neural networks along the reinforcement learning framework. Most of them tested their results on the mountain-car problem. For example, [3] used neural network along with Sarsa(0) to solve the mountain-car problem. Their network structure was similar to ours, but they used 20 hidden nodes and they also changed the reward to -0.1 per time step. Setting the learning rates to 0.063, 0.062 respectively for hidden and output layers, they empirically showed that the algorithm converges to a near optimal policy after about 67000 trials.

Unlike our strategy, [2] used neural network as an action-value function by considering three different nodes corresponding to three different actions in the output layer. It divided each state feature, velocity and position, into ten regions and assigned one input for each region; therefore, it modeled the input layer with twenty nodes. At each time step, the inputs associated with the current region were set to one. Hence, only two inputs were activated for any given state.

Conclusions

As we learnt from the book, it was obvious that Sarsa($\lambda$) with tile coding will do perfect on the task. Furthermore, we also expect good results from Actor-Critic; however, we did not expect better results comparing to Sarsa($\lambda$) because we are not using the Actor-Critic algorithm on the correct domain. In other words, Actor-Critic is designed to handle continuous state-action spaces and our problem does not have this property. Finally, we believed that non-linear Sarsa should converge to a near optimal policy. However, our results did not support this hypothesis due to the fact that setting the parameters in neural network is a hard task and small mistakes in this task may lead to poor results.

References

