CP8319/CPS824 Reinforcement Learning

Winter 2021

Assignment 3

Rohaan Ahmed
PhD Student - Computer Science
rohaan.ahmed@ryerson.ca

April 4, 2021

Instructor: Dr. Nariman Farsad

1. Test Environment

1.a. Max Reward for Test Environment

The Maximum Reward achievable in the Test Environment over 5 time-steps is 4.1. We will show this is true using a Greedy approach to start, and then incrementally going up in steps to maximize the long-term reward.

Maximum Immediate Reward:
The maximum reward possible after a single state-transition is 2.0, from **2** $\rightarrow$ **1**. We want to maximize the number of times we perform this transition, as it gives us the maximum immediate reward (i.e., be Greedy).

Returning to Maximum Rewars State Transition:
From **1**, we want to perform the least amount of steps that allow us to return to **2**, thus we would perform the transition **1** $\rightarrow$ **2**, and obtain a reward of 0.0. Note that this is *not* a Greedy step, because we are trying to maximize the long-term reward, rather than maximize the immediate reward from **1**.

Subsequent time-steps:
We will continue transitioning from **2** $\rightarrow$ **1** $\rightarrow$ **2** to obtain the maximum possible reward for an even number of time-steps. In this way, at the end of ever even time-step, we will end up in **1**.

Final (Odd) time-step:
For the final time-step (5th), we perform the transition that gives us the maximum immediate reward **1**, since we will not be able to perform the transition **2** $\rightarrow$ **1** any more in the future. Therefore, for the final time-step, we perform the transition **1** $\rightarrow$ **0**, obtaining a reward of 0.1.

Thus, the 5 time-step path that maximizes the total reward is given by:
0 $\rightarrow$ **2**: R = 2.0
2 $\rightarrow$ **1**: R = 0.0
1 $\rightarrow$ **2**: R = 2.0
2 $\rightarrow$ **1**: R = 0.0
1 $\rightarrow$ **0**: R = 0.1
Total Reward = 4.1

2. Q-Learning and Value Function Approximation

2.a.

Representing the Q-function as $\hat{q}(s;\mathbf{w}) \in \mathbb{R}^{|\mathcal{A}|}$ allows us to obtain the optimal action at every forward-pass by looking at the state-action values for all actions from a given state [1].
Alternatively, representing the Q-function with respect to all state-action pairs would increase the number of passes, thus requiring more computation. This alternative case may, however, lead to more accurate update at each step, theoretically.

2.b.

Result of running `q2_schedule.py`

```
Test1:  ok
Test2:  ok
Test3:  ok
```

2.c.

Let:

$$\Lambda = \left(r + \gamma \max_{a'\in\mathcal{A}} \hat{q}(s',a';\mathbf{w}) - \hat{q}(s',a';\mathbf{w})\right)^2$$
$$\implies L(\mathbf{w}) = \mathbb{E}_{s,a,r,s' \ \mathcal{D}}[\Lambda]$$

In order for the update to be considered Stochastic Gradient Descent, it must be of the form:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}}(\Lambda) \tag{0.1}$$

We can check if the update rule can be considered Stochastic Gradient Descent by simply taking the Gradient of Λ with respect to $\mathbf{w}$. Without actually performing the differentiation, using the Power Rule we know that the gradient will contain the following term:

$$\nabla_{\mathbf{w}} \left(r + \gamma \underbrace{\max_{a'\in\mathcal{A}} \hat{q}(s',a';\mathbf{w})}_{\lambda_1} - \hat{q}(s',a';\mathbf{w}) \right)$$

Since we know that the term λ_1 in the above equation depends on $\mathbf{w}$, it will not be zero. Therefore, we know that the update for $\mathbf{w}$ is not of the form shown in Equation 0.1.

Therefore, this update cannot be considered Stochastic Gradient Descent

[1] I am not quite sure what this question is really asking. What is one benefit of representing the Q-function in this way compared with what? I've answered the question as I've interpreted it.

2.d.

Again, order for the update to be considered Stochastic Gradient Descent, it must be of the form shown in Equation 0.1

Using the same technique as in Section 2.c., we let

$$\Lambda = \left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}(s', a'; \mathbf{w}^-) - \hat{q}(s', a'; \mathbf{w})\right)^2$$
$$\implies L^-(\mathbf{w}) = \mathbb{E}_{s,a,r,s'\ \mathcal{D}}[\Lambda]$$

We can check if the update rule can be considered Stochastic Gradient Descent by simply taking the Gradient of Λ with respect to $\mathbf{w}$. Without actually performing the differentiation, using the Power Rule we know that the gradient will contain the following term:

$$\nabla_{\mathbf{w}} \left(r + \gamma \underbrace{\max_{a' \in \mathcal{A}} \hat{q}(s', a'; \mathbf{w}^-)}_{\lambda_2} - \hat{q}(s', a'; \mathbf{w}) \right)$$

λ_2 in the above equation *does not* depend on $\mathbf{w}$, but on $\mathbf{w}^-$, which is considered a constant. Therefore, we know that the update for $\mathbf{w}$ will be of the form shown in Equation 0.1.

Therefore, this update can be considered Stochastic Gradient Descent

3. Linear Approximation and DQN

3.a. Linear Approximation

Yes, we achieve the maximum reward of 4.1 using Linear Approximation, as shown in Fig 1.

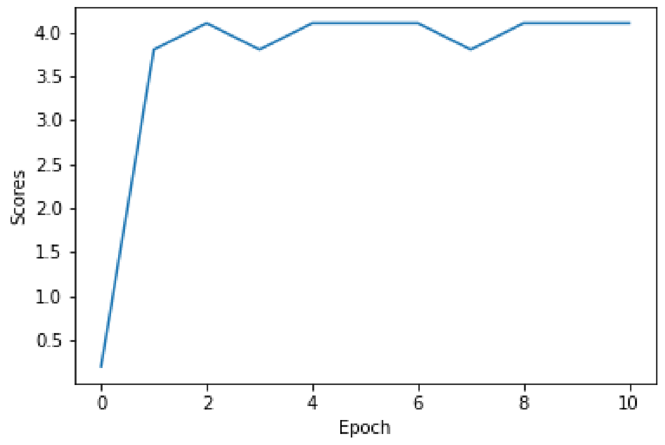

Figure 1:

3.b. Deep Q-Network

We achieve the maximum reward of 4.1 using Deep Q-Network as well, as shown in Fig 2.

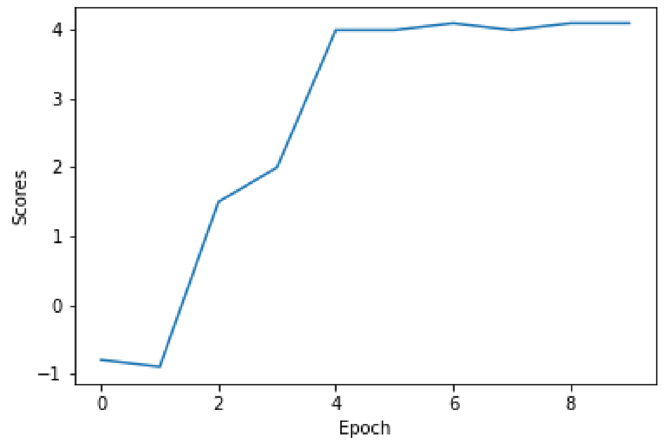

Figure 2:

Comparison: Linear Approximation vs Deep Q-Network

As can be seen from the plots in Figures 1 and 2, the Linear Approximator converges to the optimal solution quicker. At the end of Epoch 1, Linear Approximation is already quite close to the maximum reward, and actually achieves it at the end of Epoch 2. On the other hand, the DQN does not converge to the optimal solution until Epoch 4. It should be noted that the plots generated will vary each time we train the network anew, since the weights are initialized randomly. However, in the average case, the Linear Approximator will converge faster than the DQN, as shown.

This is understandable, because the DQN is significantly more complex than the Linear Approximator, far more than what is required for this toy problem. In the DQN, we forward and backward propagate through several layers of the network, which requires much more computation. For simple problems, such as the Test Environment, this added complexity is unnecessary.

Based on experience in industry, I can say that matching the complexity of the model with the complexity of the problem is quite important. Using overly-complex models can not only increase training time, but lead to reduced performance through "overfitting" or "underfitting". Although "overfitting" is a more clearly understood concept in Supervised Machine Learning, in Reinforcement Learning overfitting may cause an agent to correlate rewards with certain spurious or undesirable features from the observations. "Underfitting" can be caused by insufficient training or reward tuning.

Thanks to the Profesor and Teaching Assistants for an excellent course. Unfortunately, due to external commitments, I could not interact with the class as much as I would have liked to. Regardless, the course was very well taught enjoyable.

- Rohaan