Project 5: Reinforcement Learning
ESE 650 Learning in Robotics, 2018 Spring

Dian Chen

April 10, 2018

Contents

Introduction

Reinforcement learning is widely used to solve decision-making problems, more specifically, to generate sequential control signals for an agent while the agent is operating in the environment. The learning process is guided by the agent's knowledge of the world, in the form of "rewards" that it receives. Higher rewards for a single step may imply a desirable local decision, while higher cumulative rewards for a whole trajectory may imply a more desirable global behavior sequence. Our goal is to come up with an optimal policy that tells the agent what to do in each state so as to collect as much rewards as possible.

When the dynamics of the world, or mathematically the transition probability and reward function are fully known, the problem becomes a dynamic programming problem and we can easily solve the optimal policy by value iteration or policy iteration. However, in most cases it's impossible to fully know the dynamics, and the only thing that's available to us is the sequence of rewards the agent is seeing. We need to make full use of the rewards as our knowledge of the environment. There are several ways of categorizing the reinforcement learning problem: the environment, i.e., the state and action space can be continuous or discrete, and the algorithms can be roughly divided into value-based or policy-based, and model-based or model-free, etc.

In this project we are given several environments, of discrete type and of continuous type (in state space only or in both state space and action space), to play with. Specifically, we are using a grid world **Maze**, **Acrobot-v1**, **MountainCar-v0** and **MountainCarContinuous-v0** from **gym** by OpenAI. We are supposed to experiment with different algorithms and solve for the optimal policy for each of these problems.

All the plots and animations can be found at this google drive link:

`https://drive.google.com/open?id=1vdsOx9Wk5jkRa649m9q6JN1CmHkgXqnd`

(Notice: I made some slight modifications to the code on printing values and plotting, but major functionalities are left untouched.)

1 Policy Iteration / Value Iteration

1.1 Q-Value Iteration

In this problem, fortunately, all the transition probability and reward function are known, so either policy iteration or value iteration can give us the optimal policy at each state. Moreover, the **Maze** environment is a toy-scale discrete world, which makes the problem computationally feasible.

Value iteration typically converges to the real expected cumulative rewards for each state, and we can use this information together with our known transition model to compute the optimal policy at each state. However, the state values actually give much more fine-grained information than actually needed to make a discrete decision: a much coarser set of state values will still do the job as long as the policy discretization stays the same. So we may consider policy iteration which requires less computation at each iteration but still leads to the true optimal policy.

Q value $Q(s, a)$ is used for evaluating an action at a given state. If $Q(s, a)$ values are known for each state-action pair at a given state, we can conveniently choose the optimal action by picking the one that corresponds to the highest Q value. This circumvents the need for a transition model and an extra evaluation to choose the optimal action if we have only the state values $V(s)$.

I chose to obtain the Q values for this **Maze** problem, that is, to run value iteration directly on Q values according to the following update rule:

$$Q(s, a) = \sum_{s'} T(s, a, s') \left(R(s, a, s') + \gamma \max_{a'} Q(s', a') \right)$$

The convergence criterion is chosen as the Frobenius norm of two successive Q value table being lower than a threshold.

1.2 Results

Please see the submitted package for Q-values. The plots for **Maze** are follows:

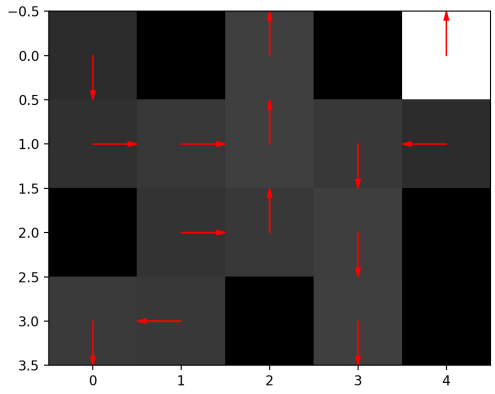

Figure 1: **Maze** Situation 1

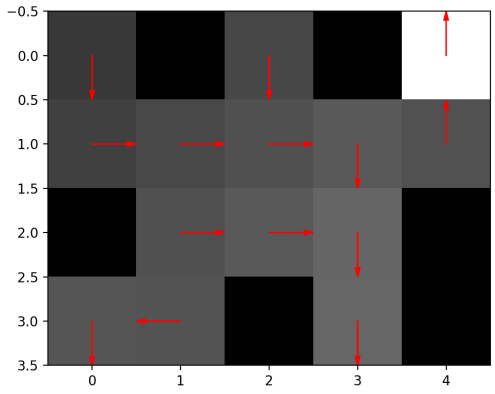

Figure 2: **Maze** Situation 2

2 Q-Learning

2.1 Implementation and Tuning

In most real world applications we no longer have the access to the underlying dynamics that is causing the transitions of our agent, so we have to make use of the reward the agent is seeing along the way and adjust our estimation and decisions online. If our agent made one move a under

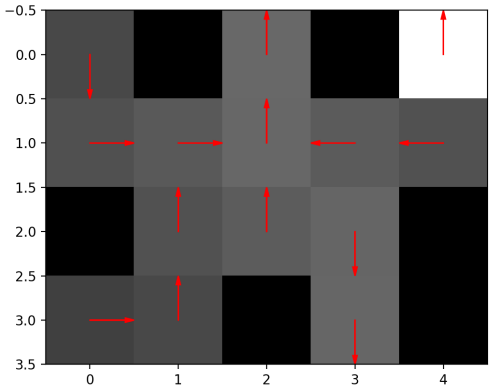

Figure 3: **Maze** Situation 3

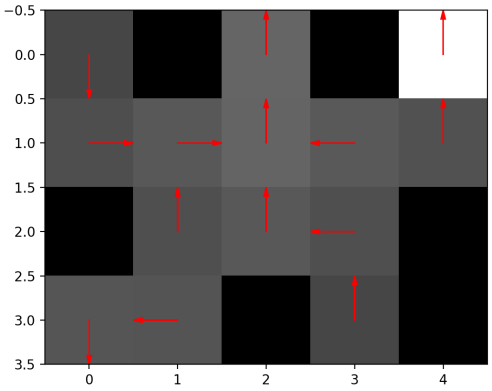

Figure 4: **Maze** Situation 4

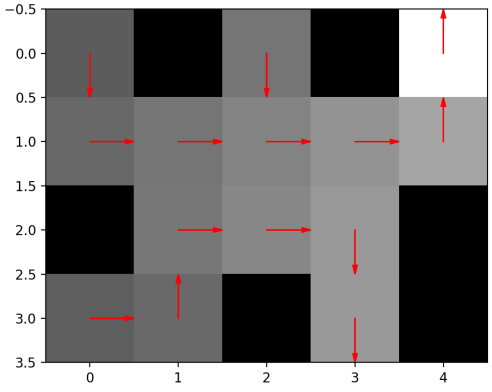

Figure 5: **Maze** Situation 5

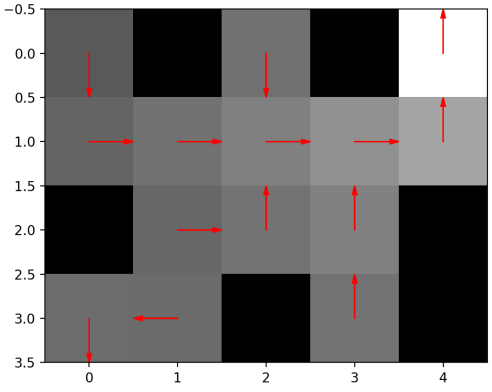

Figure 6: **Maze** Situation 6

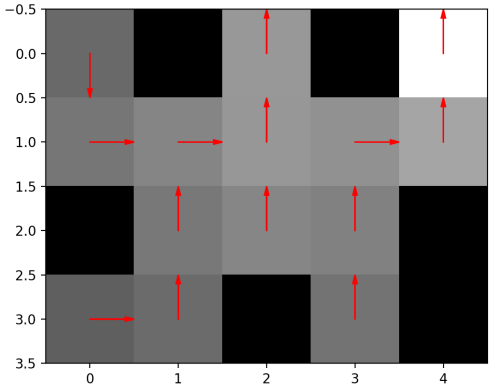

Figure 7: **Maze** Situation 7

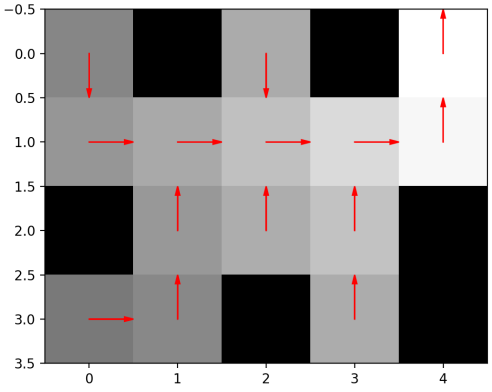

Figure 8: **Maze** Situation 8

current policy π at s, transitioned to s' and received reward R, then the target estimation of $Q(s,a)$ should be $R + \gamma Q(s', \pi(s'))$ (also referred as TD target). However, this single move is only one sample of this particular transition, so we cannot put total trust on the estimation, i.e., change $Q(s,a)$ to be entirely the TD target. A reasonable way of utilizing this piece of information is to pull $Q(s,a)$ a little bit towards the TD target:

$$\begin{aligned} Q(s,a) &= Q(s,a) + \alpha \left(R + \gamma Q(s', \pi(s')) - Q(s,a) \right) \\ &= (1-\alpha) Q(s,a) + \alpha \left(R + \gamma Q(s', \pi(s')) \right) \end{aligned}$$

2.1.1 Learning Rate

The learning rate α is one of the tunable parameters. Too big α leads to too oscillating updates, while too small α makes the learning quite slow. In my experiments I found that α being set to around 0.15 is a good choice.

2.1.2 Action Policy

In this problem I've tried two stochastic mechanism:

1. ϵ **greedy**: with ϵ probability exploit the optimal policy and uniformly explore with $(1-\epsilon)$ probability.
2. **softmax**: convert the $Q(s,a)$ values for each action a at s to probabilities via softmax function, and sample a discrete action according to this distribution.

Here ϵ is another tunable parameter. If we are more concerned with exploiting our optimal policies, good choices for ϵ are typically 0.8 to 0.9; if we just want to explore the world and collect evenly distributed samples, we may set ϵ to be around 0.1 to 0.2.

2.1.3 Optimal Q Values

If we are only concerned to found the optimal Q values, then we should visit each state as much as possible so as to collect sufficient samples to make updates. In my experiments I've found ϵ greedy policy with a low exploitation probability did a much better job in only 5000 iterations, since I have direct control over how spread out the visits can be. Softmax stochasticity samples actions with higher Q values more often, which leads to less diversity. However, in real applications we are more concerned with making optimal decisions, so with low ϵ even if we've converged very close to the true Q values, we aren't able to exploit the Q values since the randomness will dominate the guidance from Q values.

Smaller ϵ leads to faster convergence of Q values, but is less useful for real control; if I use larger ϵ, a lot of entries are unexplored when 5000 iterations ran out since limited exploration is allowed. A good practice is to introduce time-varying ϵ such that it starts from very low at the beginning, allowing sufficient exploration and increases to around 0.9 when the agent keeps progressing, allowing reasonable exploitation.

The RMSE converges to a constant gap, however the optimal policy at each state produced by the Q table is the same as that yielded by the true Q values from Question 1.

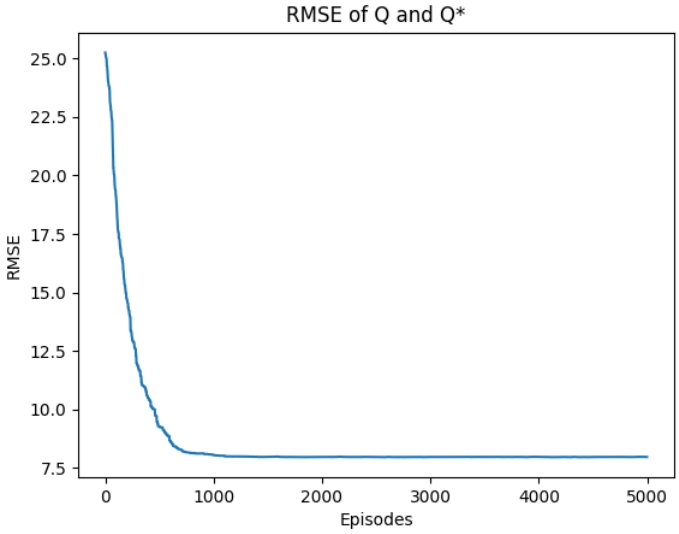

Figure 9: RMSE of Q and Q* with $\epsilon = 0.1$ and $\alpha = 0.2$

3 Continuous State Space Problems

Once we move into continuous space we can no longer use the tabular methods to perform value or policy iterations. Viable solutions for continuous space problems use features, which are typically designed based on states, and approximate functions as the equivalent of a Q value table or policy distribution.

3.1 Deep-Learning-based Discrete REINFORCE

3.1.1 Algorithm Description

In Monte-Carlo Policy Gradient (REINFORCE) method, we directly model our policy distribution using a set of parameters, and update the parameters by stochastic gradient ascent during our learning process. Here, we use the discounted cumulative rewards v as unbiased samples of $Q^{\pi_\theta}(s, a)$, and by using policy gradient theorem (see reference by D. Silver) we can have the following parameter update rule:

$$\theta = \theta + \nabla_\theta \log \pi_\theta(s, a) v$$

In this project, I chose to use neural network as my function approximator for $\pi_\theta(s, a)$. The input layer takes in state features and is followed by two fully connected layers with relu activation, which is then followed by a third fully connected layer and finally a softmax normalizer which outputs probabilities for each discrete action. The final action to take is then sampled according to this probability distribution. To make use of the auto-differentiators, I chose to use **TensorFlow**(1.5.0) to build up the network and take care of the gradients. As in common machine learning problems, we need to have a target "loss" to minimize. By carefully examine the update rule, we can choose the "loss" to be:

$$L(\theta) = -\log \pi_\theta(s, a) v$$

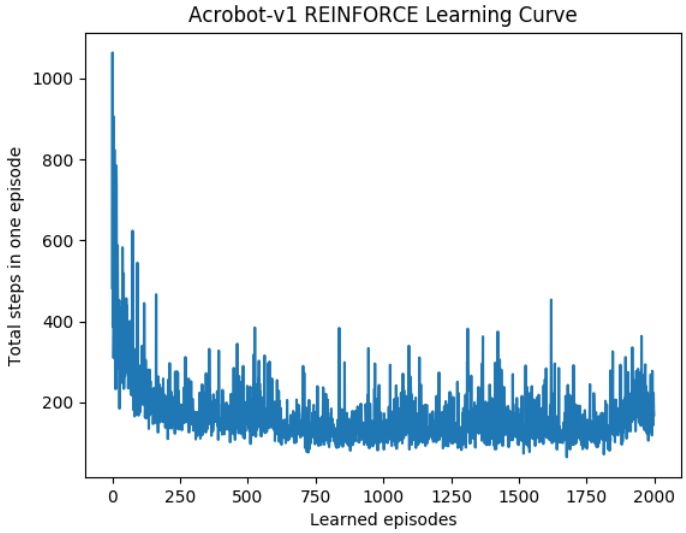

Figure 10: Learning Curve for Acrobot-v1 using REINFORCE

The learning process proceeds as follow: we first sample an complete trajectory of length T, then for each step we can effectively compute the discounted cumulative rewards v_t such that we can compute the above loss. For this dataset containing T samples (each corresponding to one time instance), we can run several epochs with mini-batches in each epoch on it and update our parameters. After making full use of this episode, we reset the agent, start a new episode and repeat the update processes.

3.1.2 Acrobot-v1

For **Acrobot-v1**, we have 6 state features, 3 action choices. The two hidden layers in my neural network have 10 and 20 neurons, respectively. Learning rate is set to $\alpha = 0.001$ for Adam Optimizer. Batch size is 512 and number of epochs is 3. Discount factor $\gamma = 0.9$ but for $\gamma = 0.99$ I was able to achieve faster convergence with better results (fewer steps to complete an episode). After 1000 episodes the steps needed towards completion can effectively converge to around 100, in around 2 miniutes. See Figure 11.

3.1.3 MountainCar-v0

For **MountainCar-v0**, we have only 2 state features, 3 action choices. The other parameters are the same as used in **Acrobot-v1**. Interestingly, I noticed that since **MountainCar-v0** has so few state features, even two hidden layers may be an overkill for it. In my experiments I found that a single hidden layer would also do a satisfactory job, and two layers sometimes has slower convergence. One of the challenges in this mountain car model problem may be posed by its lack of state features. After 1500 episodes the steps needed towards completion can effectively converge to around 180, in around 3 minutes. See Figure **??**

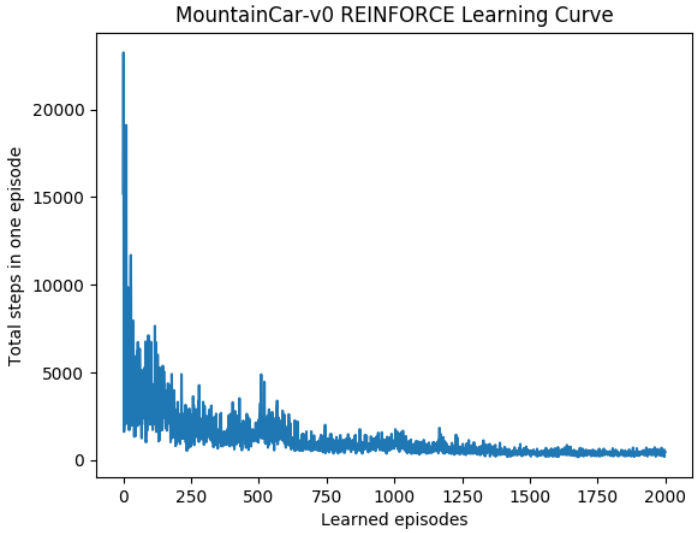

Figure 11: Learning Curve for MountainCar-v0 using REINFORCE

3.1.4 Discussion

As our first attempt to use Deep Learning in Reinforcement Learning, we can first notice that **MountainCar-v0** has so few state features that two hidden layers may be an overkill for it. In my practice I noticed that two hidden layers didn't bring much boost in performance than single hidden layers, sometimes even making it longer to converge. On the other hand, **Acrobot-v1** has more state features, and two hidden layers did serve better than single hidden layer. This lack of state features of **MountainCar-v0** may be one of its challenges, in which case we don't have full representation of the world.

3.2 Deep Q Networks

3.2.1 Algorithm Description

In REINFORCE we directly modeled the policy distribution and use that to directly sample our actions. If we want to switch to value-based learning algorithms, the basic idea is still similar: use function approximators to evaluate Q values for continuous states.

Here, I chose to use neural network to approximate Q values, i.e., Deep Q Networks (DQN). The input layer takes in the state features and is followed by two fully connected layers with relu activation, which is then followed by a third fully connected layer. The output of the third layer is then used as Q values, and I chose to apply ϵ greedy to sample a discrete action based on the output Q values for a given state. Again, the network is built using **TensorFlow** as in REINFORCE.

The main idea of DQN learning goes like follow. The agent has a memory that keeps records of (s, a, r, s') for each step, and we are trying to minimize the MSE between the estimation and target Q values, i.e., the following quantity:

$$L(\theta) = \frac{1}{n} \sum \left(r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a) \right)^2$$

in which n is the number of samples used in one batch (512 in my experiments). Here, unlike in REINFORCE, we do have a meaningful loss. After each step, we add this new piece of record to the memory, and sample n pieces from the memory as a batch to feed into the network and perform one update. When the records reach the memory capacity N, the newly arrived records would overwrite the old ones so that the memory always keeps the most recent N records. The use of a memory and sampling from it at every step is called "**experience replay**", which is commonly used in DQN to achieve better convergence (see reference by D. Silver).

Another trick that helps DQN achieve better performance is "**fixed Q-targets**" (see reference by D. Silver). The main idea is that, in practice we keep two separate networks, which for convenience I'll refer to as the "fresh" one and "fixed" one. The fresh network is updated at every step and is responsible for computing $Q_{\theta_{fresh}}(s, a)$ in the above equation, while the fixed network is only update to be the same as the fresh network every certain number of steps (300 steps in my experiments), and is responsible for computing $Q_{\theta_{fixed}}(s', a')$. To make the functionality of the two networks more clear, the above loss in practice should be computed as:

$$L(\theta) = \frac{1}{n} \sum \left(r + \gamma \max_{a'} Q_{\theta_{fixed}}(s', a') - Q_{\theta_{fresh}}(s, a) \right)^2$$

The process of choosing action and single step learning repeats.

3.2.2 Acrobot-v1

For **Acrobot-v1**, we have 6 state features, 3 action choices. The two hidden layers in my neural network have 10 and 20 neurons, respectively. Learning rate is set to $\alpha = 0.001$ for RMSProp Optimizer. Memory size is 3000. Discount factor $\gamma = 0.9$ but with $\gamma = 0.99$ I was able to achieve faster convergence with better results (fewer steps to complete an episode). However in practice, I found that it's extremely hard for DQN to converge for **Acrobot-v1** using the original rewards. I've modified the rewards as follow:

$$r = -\left(\cos\theta_1 + \cos(\theta_1 + \theta_2)\right)$$

which is an intuitive, valid choice since the goal is to make the end-effector reach the line above. This led to extremely fast convergence: after 20 episodes the steps needed towards completion can effectively converge to around 80, in around 3 seconds. See Figure 12

This problem will be discussed later in detail.

3.2.3 MountainCar-v0

For **MountainCar-v0**, we have 2 state features, 3 action choices. The two hidden layers in my neural network have 10 and 20 neurons, respectively. Learning rate is set to $\alpha = 0.001$ for RMSProp Optimizer. Memory size is 4000. Discount factor $\gamma = 0.9$ but with $\gamma = 0.99$ I was able to achieve faster convergence with better results (fewer steps to complete an episode). Here I've encountered the same problem: it's extremely hard for DQN to converge using the original rewards. I've modified the rewards as follow:

$$r = |position - (-0.5)|$$

Here the -0.5 is the mean of Gaussian distribution from with the initial position is sampled. This again is an intuitive, valid choice since the goal is to make car reach the flag, i.e., farther away from

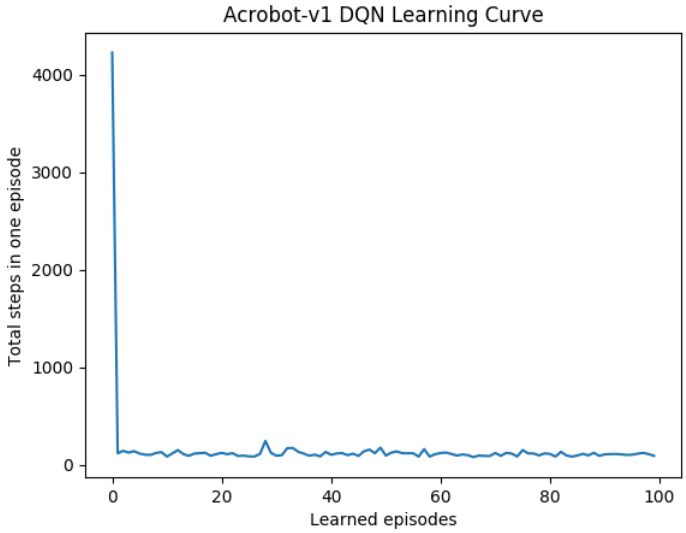

Figure 12: Learning Curve for Acrobot-v1 using DQN

the pit. This also led to extremely fast convergence: after 20 episodes the steps needed towards completion can effectively converge to around 140, in around 5 seconds. See Figure 13

This problem will be discussed later in detail.

3.2.4 Discussion

The use of DQN in this section has revealed a big problem: the original undiscriminizing rewards, uniformly collected at each step may be of little value for this kind of single-step learning. The advantage of the experience replay trick commonly used in DQN is to break up temporal relations between steps and make full use of each piece of record, so it is best suited when the reward varies for different states. However, in these two models the reward is uniformly -1 as long as the agent has not reached the goal. Although it makes perfect sense for us to minimize the length of the trajectory through reinforcement learning, but the single-step learning nature and decoupled memory data in the algorithm make the agent unable to see the effect of reaching the goal. This is why modifying the reward in an intuitive way would tremendously boost the performance, while sticking with uniform -1 rewards leads to poor performance.

3.3 Performance Evaluation

There are many ways to evaluate the performance. I chose to use the steps required for completion as an evaluation metric, which is very straightforward. This is equivalent to using cumulative rewards (undiscounted), since both models output uniform rewards before termination. Learning curves are included in the sub-sections in which the models are discussed.

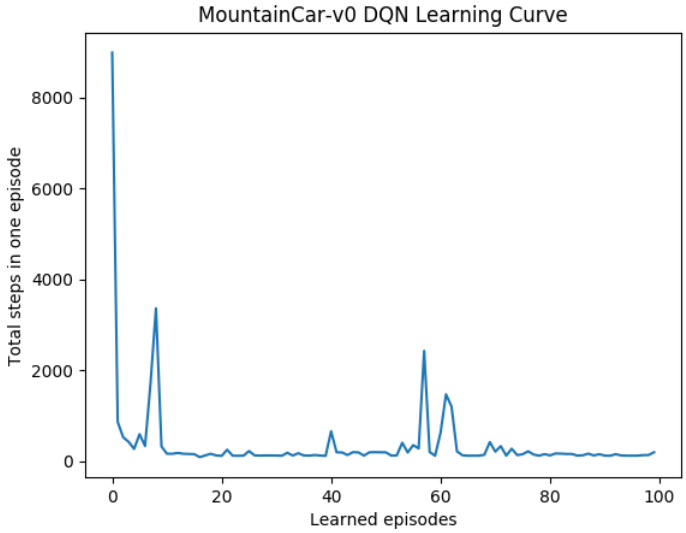

Figure 13: Learning Curve for MountainCar-v0 using DQN

3.4 Value or Policy Iteration?

Here since our worlds become continuous, we can no longer use the tabular value iteration or policy iteration as in problem 2. If we do want to estimate the values of states or state-action pairs, i.e., $V(s)$ or $Q(s, a)$, we may use either function approximation or discretize the world.

4 Continuous State and Continuous Action Problems

In this fourth problem the action space also becomes continuous, but the idea behind learning is essentially the same for REINFORCE: modelling policy distributions directly.

4.1 Deep-Learning-based Continuous REINFORCE

4.1.1 Algorithm Description

To deal with the continuous action space, we only need to slightly modify the output layer of the network used in REINFORCE for discrete action problem. For 1-D continuous action space in this problem, I chose to model the policy distribution as a Gaussian distribution, and the output is used as a set of weights which is applied on the state features to obtain the mean:

$$\mu = \phi(s)^T s$$

This μ together with a predefined standard deviation σ is used to compute the $\log \pi_\theta(s, a)$ term in the "loss" (see above), as well as sampling our continuous action. All the other learning procedures are exactly the same as in discrete REINFORCE.

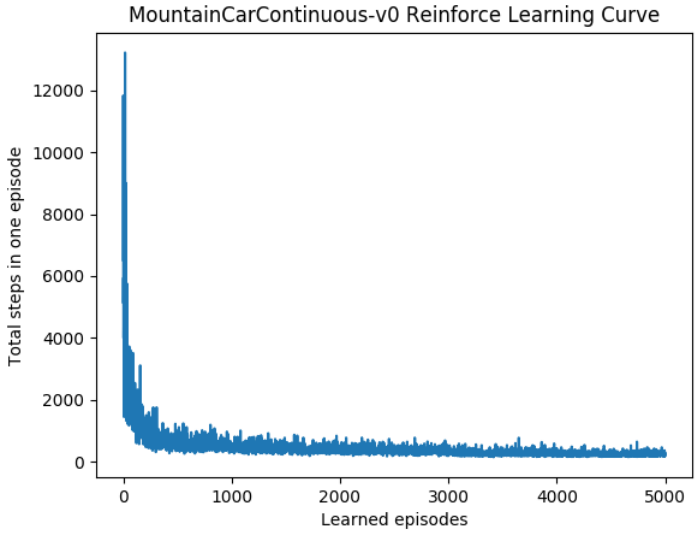

Figure 14: Learning Curve for MountainCarContinuous-v0 using REINFORCE

4.1.2 MountainCarContinuous-v0

For **MountainCarContinuous-v0**, we have 3 state features, which corresponds to 3 coefficients in the output. The two hidden layers in my neural network have 10 and 20 neurons, respectively. Learning rate is set to $\alpha = 0.002$ for Adam Optimizer. Batch size is 512 and number of epochs is 1. Discount factor $\gamma = 0.9$ but with $\gamma = 0.99$ I was able to achieve faster convergence with better results (fewer steps to complete an episode). After 3000 episodes the steps needed towards completion can effectively converge to around 200, in 3 minutes. See Figure 14

4.1.3 Discussion

Notice that here we are using a predefined σ and only learning the linear weights $\phi(s)$ for computing μ. This makes the learning task easier, however by doing so we are essentially assuming equal amount of exploitation-exploration. If I make the network to also learn σ, the performance may become better, but also may become more finicky. In practice, I noticed that when using larger σ (e.g. $\sigma = 1$), it's more possible for the agent to find a way out in the earlier episodes, but the convergence is slower and "steps required for completion" metric is noisier because we are still allowing a fair amount of random exploration. On the other hand, if I use smaller σ (e.g. $\sigma = 0.3$), the convergence may be faster and the steps required for completion are fewer, since we increased the exploitation. However life gets harder for the agent in the early episodes and some episodes may last for ridiculously long time. This is a trade-off, and I found that $\sigma = 0.5$ is a good choice.

Reference

1. ESE 650 2018 Spring Lecture Notes
2. `http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html`, Reinforcement Learning Course by David Silver

3. https://www.youtube.com/watch?time_continue=29&v=7huURSBATmg, Artificial Intelligence Course by Anca Dragan