CS7642 RLDM Project #2

1. Problem definition

The state has 8 components: horizontal and vertical position, horizontal and vertical velocity, angel and angular velocity, and left and right leg ground contact (binary).

The action space consists of 1)do noting, 2)fire main engine, 3) fire left engine (push right), and 4) fire right engine (push left).

The objective is to land the vehicle on the target fast, safely, and efficiently.

2. Baseline solution: Q Table

3. Experiments

I implemented the DQN algorithm proposed by DeepMind [1] with Tensorflow. In short, the Deep Q-Learning algorithm selects actions according an ε-greedy policy. Each experience tuple <s, a, r, s'> is stored in a Replay Memory structure. On each algorithm iteration, a random sample of these stored memories (minibatch) is selected and Q-Learning updates are applied to these samples. The detailed algorithm and the advantages of this approach are described in detail in reference

But **we don't have any idea of the real TD target.** We need to estimate it. Using the Bellman equation, we saw that the TD target is just the reward of taking that action at that state plus the discounted highest Q value for the next state.

There are a lot of decision-making involved in the implementation: how many hidden layers, how many nodes in each layer, how to initialized the parameters, what activation function to use, etc.

3.1. Experiment 1: Explore network structure

A DQN is built with the following structure:

1. Neural network with 2 hidden layers, each with 32 nodes
2. Activation function for both hidden layers is Rectified Linear activation (ReLU)
3. output layer has a linear activation function with mse loss function
4. Adam optimzer with learning rate of 0.001
5. ϵ -gready poliy was used during training

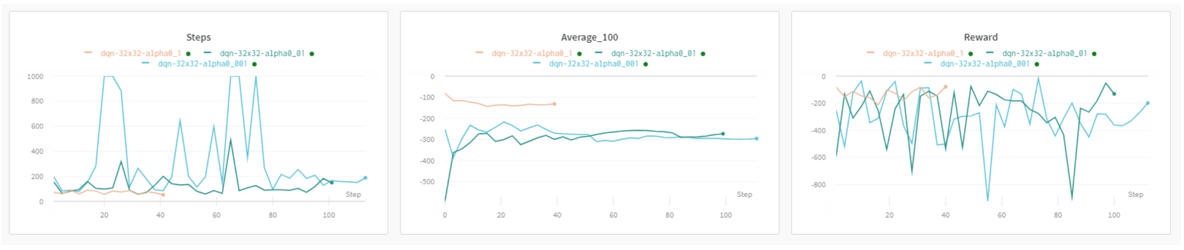
3.2. Experiment: Change epsilon decay rate

The intuition of using time decaying epsilon (exploration rate) is simple: at the beginning of the learning process, we want the agent to explore the environment more since it has little prior knowledge about the environment. A random choice of actions gives the agent to explore the state space more. As the learning goes on, the agent knows more about the environment and should be more confident with the action predictions, thus random exploration should be reduced.

3.3. Experiment: Change learning rate ($\alpha $)

learning_rate(α)

The learning rate determines how aggressive the q values are updated. A smaller learning rate means the q values is updated slower. A extreme case of 0 indicates that q values are never updated. Learning rate is an important parameter that impact how fast the model can be trained. In this experiment, I varied the learning rate from very small value 0.001 to relative large value 0.1: [0.001, 0.01, 0.1]. The remaining parameters are intact. Figure 1 plots the impact of learning rate.



The plot on the left demonstrate the number of steps in a episode. A larger learning rate (0.1) seems to push the learning to occur quickly and number of steps is significantly smaller.

The *learning rate*, set between 0 and 1 and is defined as how much we accept the new value vs the old value. This value then gets added to our previous q-value which essentially moves it in the direction of our latest update. Setting it to 0 means that the Q-values are never updated, hence nothing is learned. Setting a high value such as 0.9 means that learning can occur quickly.

For validating the effect of the different learning rates on the model performance, I have trained different agent with different learning rates. Learning rates chosen for this experiment are *0.0001, 0.001, 0.01, 0.1*. Best performance is observed for the middle value of the learning rate of 0.001. Orange line in figure 3 corresponds to this value and provides the maximum reward. The agent is not able to learn at the higher learning rate and the reward values are diverging.

3.4. Candidate solutions

Vanilla Q-learning

Linear function approximation

Deep Q-learning

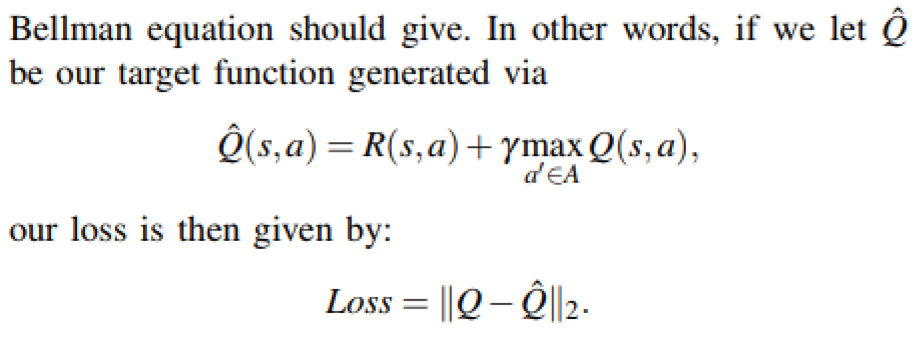
3.5. Issues and Fixes

3.6. Settled Solution

Deep Q learning. The goal here is to learn the Q function, which gives our total maximum expected reward if we start at state s and take action a. The tricky part is to get the target value for training.

Loss function. Given that Q should satisfy the optimal Bellman equaiton:

we can train the network to optimize predicted Q values towards the optimal Bellman equaiton should give.



Bellman equation should give. In other words, if we let $\hat{Q}$ be our target function generated via

$$\hat{Q}(s,a) = R(s,a) + \gamma \max_{a' \in A} Q(s,a),$$

our loss is then given by:

$$Loss = ||Q - \hat{Q}||_2.$$

4. Impact of Hyper-parameters

4.1. Impact of learning rate

4.2. Impact of exploration rate

The same idea has been employed in home work #2 and achieved desirable results.

4.3. Impact of discount factor

4.4. Impact of dropout rate

4.5. Size of replay buffer

5. Reflections

what can be improved

Amazing to see we can accomplish the results

A lot of time has been wasted in figuring out the network structure of the DQN paper by DeepMind. Suddenly realize that DQN did nothing more .

As I read more about this topic, other potential solutions popped up. Most promissing ones include dueling deep Q networks, which in fact is only minor modification to the original DQN, and

假设我们需要更新当前状态St下的某动作A的Q值：Q(S,A),我们可以这样做： 1. 执行A，往前一步，到达St+1; 2. 把St+1输入Q网络，计算St+1下所有动作的Q值； 3. 获得最大的Q值加上奖励R作为更新目标； 4. 计算损失 - Q(S,A)相当于有监督学习中的logits - maxQ(St+1) + R 相当于有监督学习中的lables - 用mse函数，得出两者的loss 5. 用loss更新Q网络。

也就是，我们用Q网络估算出来的两个相邻状态的Q值，他们之间的距离，就是一个r的距离。

DQN更新：

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q\left(S', a\right) - Q(S,A)\right]$$

5.1. 总结

1. 其实DQN就是Qlearning扔掉Qtable，换上深度神经网络。
2. 我们知道，解决连续型问题，如果表格不能表示，就用函数，而最好的函数就是深度神经网络。
3. 和有监督学习不同，深度强化学习中，我们需要自己找更新目标。通常在马尔科夫链体系下，两个相邻状态状态差一个奖励r经常能被利用。

DQN其实没有什么神秘的，不是吗？