Maze Problem by Deep Q Network

Final Project

Tian Qi, Nan Lin

Overview

In this project, we trained an agent to solve the maze puzzle in a square world. A reward of +1 is provided for hitting our cheese cell and a reward of -1 is provided for falling down the black hole. Each move from one state to the next state will be rewarded by a positive or a negative penalty amount. The goal of the agent is getting to the cheese in the shortest route possible.

The state space has 4*4 squares, along with the perception of objects around the agent's forward direction. Given this information, the agent learns how to best select actions among four discrete actions: moving forward, move backward, turn left, turn right.

What we will cover:

1. Deep Reinforcement Learning
2. Experience Reply
3. Fixed Q-Target
4. Policy
5. Result
6. Next Steps

From RL to Deep RL

1. Reinforcement Learning

In a Reinforcement Learning setting, while Monte-Carlo approaches require we run the agent for the whole episode before making any decisions, this solution is no longer viable with continuous tasks that does not have any terminal state, as well as episode tasks for cases when we do not want to wait for the terminal state before making any decisions in the environment's episode.

For Temporal-Difference(TD) Control Methods, they update estimates based in part on other learned estimates, without waiting for the final outcome. As such, TD methods will update Q-table after every time step.

The Q-table is used to approximate the action-value function q for the policy. The Q-learning is one effective TD method that uses an update rule that attempts to approximate the optimal value function at every time step. The main objective of Q-learning is to develop a policy for navigating the maze successfully. Presumably, after playing many games, the agent should attain a clear deterministic policy for how to act in every situation. In order to improve the

Q-learning process, we use two types of moves:

- Exploitation: There are 90% of the moves of our policy are based on previous experiences and results.
- Exploration: In our project, we choose 10% as 1-epsilon, exploration factor. 90% of times we take a completely random action in order to explore. To note, we also learn and update epsilon through our training to reduce the randomness of the process.

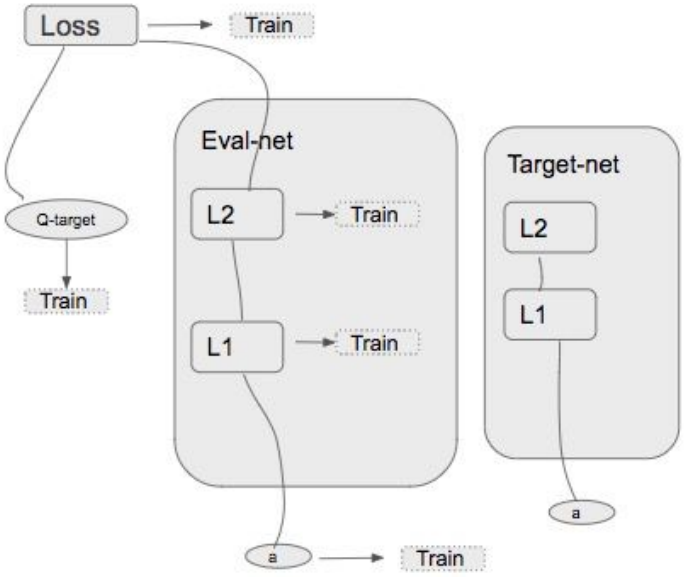
2. Deep Reinforcement Learning

If we are just solving a maze, we can just use the non-DL approach, where use the classic table when the policy is just a fixed mapping of what actions to do in what states. One limitation of tabular Q learning that we need more and more memory to store Q-table. We only applied a toy sample in this project, however, we can imagine if our states increase or the complexity of the problem increase, it would be challenging to store and search Q-table. Another major motivation is to allow the model to learn a policy and generate to unseen states which is what our maze problem about. The use of function approximation created a major breakthrough since this is exactly the purpose of neural networks to create such approximations.

In the standard problem of Navigation, we are given 16 states and 4 discrete actions so a fully-connected neural network is appropriate for the optimal-value function q.

We used a neural network with 2 hidden layers for the q- target and one 2 layers for q-evaluation. Here target net is "delayed version" of evel_net that keeps old parameters of eval_net. By doing this, we can fix the parameters of target_net through training.f

Neural Network:



Address instabilities in Deep Q-learning

In our project, we used two major methods to improve the stability of Q-learning:

1. Experience Replay

When the agent interacts with the environment, the sequence of experience can be highly correlated with each other. The Q-learning algorithm that learns from all of these experiences in sequential order can run into the situation which is stuck into local optimal.

Instead, we can keep track of reply buffer and use experience reply to sample from the buffer randomly. Therefore, we can prevent action values from oscillating or diverging catastrophically. The replay buffer contains a collection of experience. This method of sampling a small batch from replay buffer in order to learn is known as experience replay. Experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences and in general, make better use of the experience.

2. Fixed Q-Targets

If we target network to represent the Q-function, which will be used to calculate the loss of every action during training. At every step, the q-function values change so we can use a single network and the value estimates. When we train our agent, we update the weights accordingly to the TD Error. But the same weights apply to both the target and the predicted value.

We move the output closer to the target, but we also move the target. So, we end up chasing the target and we get a highly oscillated training process. It would be great to keep the target fixed as we train the network.

Instead of using one Neural Network, we used two. One as the main Deep Q Network and a second one (called Q-Eval Network) to update exclusively and periodically the weights of the target. This technique is called Fixed Q-Targets. In fact, the weights are fixed for the largest part of the training and they updated only once in a while.

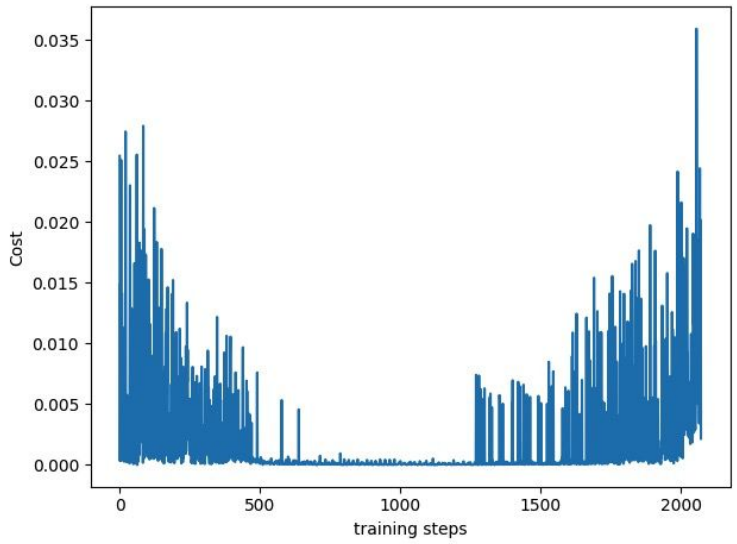
Policy

DQN is an off-policy method. We randomly searched and selected previous experiences for learning. In this project, the learning policy is Greedy in the Limit. With max 0.9, we set 1-epsilon 0.1 as a starting point and enable maximum exploration. Throughout DQN, we applied epsilon decay for each step to reduce the randomness of the process but always make sure it greater than 0.1.

Results

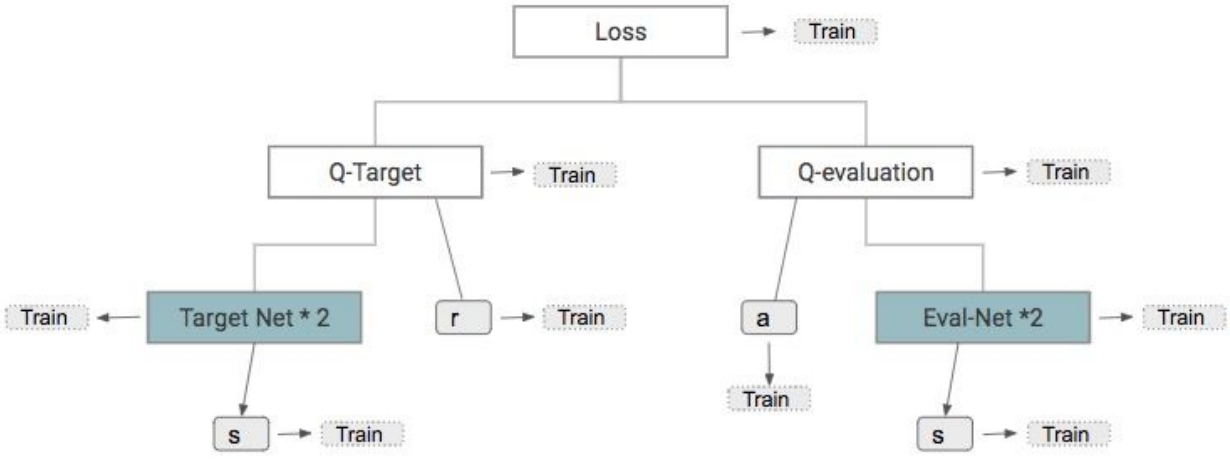
If we record the cost(target - eval)for each step, we found our results as below, the cost

decreases as we trained more steps. They are not linearly smooth since for deep Q network, unlike supervised learning, the input data changes based on situations.



Next Steps

In this project, we used deep Q-learning. Deep Q-learning tends to overestimate action values. In the future, added q_target calculation into Tensorflow graph as below to make it better interpretability.



Code Base

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For t = 1,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D

    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For