INFO 550 Final Project – Deep Q Learning Network Play Pacman

Name: Mingde Li

Abstract

Deep Q-Learning Network (DQN) is a combination of two Convolutional Neural Network (CNN) and two Fully Connected (FC) layer. I used this network concept to play the Pacman AI projects developed at UC Berkeley. The network can win the game after more than 4500 episodes of training, and it achieves ~20% winning rate after 6000 training episodes. This network only contains two CNN layers and 512 neurons in the second FC layer. A more complicated network should reach higher dining rate, but it might take longer training time. For a game with $9 * 18$ map and two ghosts, a neural network might be too complicated considering its training time to achieve ~30% wining rate is around 1 hour using 8700 cuda core.

Theory

DQN is one of the reinforcement learning (RL) methods. RL represents the system where an agent makes actions based on its observations of the environment. A positive action is given by a reward and vice versa. Deep RL was used to play Atari and it was well introduced in a paper from DeepMind Technologies [Mnih]. To represent this mathematically, we call the current observed environment s, the agent's action is a, and the procedure to go from s to a is called a policy, π., This written in logic expression is,

$$a = \pi(s) \text{ or } \pi(a|s)$$

It is too complicated if we apply all the observations to predict future state, and thus we introduce and simulate system with Markov Decision Process (MDP). MDP mentions that future is only dependent on current state, which can be represented as,

$$P(s_{t+1}|s_t) = P(s_{t+1}|s_t, s_{t-1}, \dots, s_0)$$

Were P being probability and s_t is the environment observation at time t.

In order to evaluate the action's type, we introduce value function, $v(s)$,

$$v(s) = E[G_t|s_t = s]$$

$$G_t = R_{t+1} + \lambda R_{t+2} + \cdots = \Sigma_{k=0}^{\infty} \lambda^k R_{t+k+1}$$

Where R represents the reward, λ is the discount factor valued between 0 and 1. This equation represents the expectation of s to future reward. Similarly, we introduce Action Value function, Q^{π}, to evaluate each action.

$$Q^{\pi}(s,a) = E[r_{t+1} + \lambda r_{t+2} + \lambda^2 r_{t+3} + \cdots |s,a]$$

Where r is the reward after taking an action a at state s. DQN is using the concept of finding the best Q^{π}, and we can express it with Bellman equation,

$$Q^*(s,a) = \max Q^{\pi}(s,a) = E_{s'}[r + \lambda \max Q^*(s',a') \,|s,a]$$

And it is not hard to see that when a' leads to maximum Q, Q^*also reaches maximum value.

Q-learning's main idea is to evaluate the current Q based on original Q and current reward.

$$Q^*(s,a) = Q(s,a) + \alpha(r + \gamma \max Q(s',a') - Q(s,a))$$

DQN uses neural networks to evaluate Q. We train it using the target Q value, $r + \gamma \max Q(s', a')$, as label and its loss function is,

$$L(w) = E[(r + \gamma \max Q(s', a', w) - Q(s, a, w))^2]$$

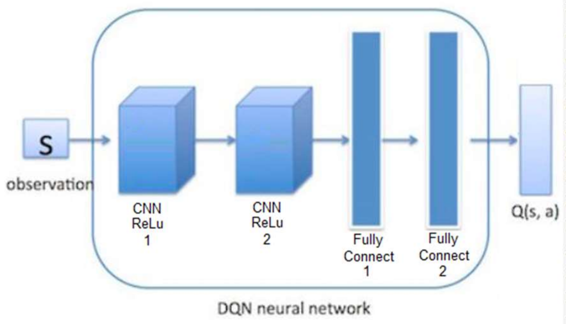
The above knowledge was explained in greater detail by Mnih et al.'s paper [Mnih] and the class textbook [Russel].

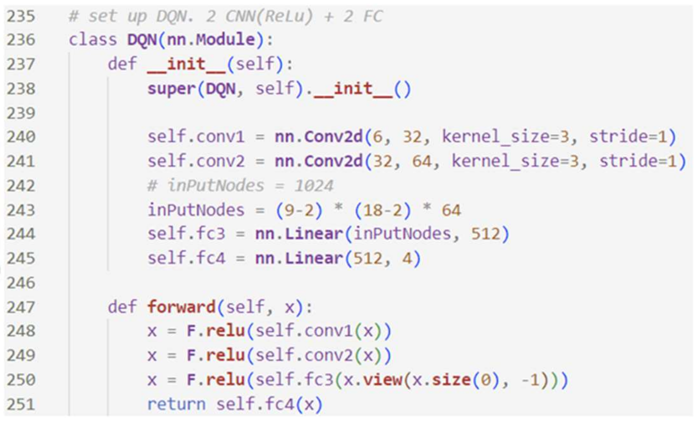
Approach and Results

The Pacman game was copied from Assignment 4 where we program Q-learning Agent. The DQN Agent is added to the "pacmanAgent.py" after line 66. A few irrelevant files, such as "autograder.py" and "submission_autograder.py" were removed from the project. A few external packages, including "pytorch, numpy," were installed for neural network training. Agent will not run if Pytorch is not installed. "mediumClassic" layout was used for training and testing.

Shown below in Figure C1 is a diagram of the DQN that was used in this project. The python code is shown as well. Part of the network setup was inspired by Ouderaa's project [Ouderaa]. The training I set takes 8000 training episodes and it takes the program 1:45:12 to complete it. It only achieves a 43% winning rate. This is an embarrassing result with an average score of 582.69. The average score of all winning episodes is 1015.71.

I also went back to Assignment 4 and used Approximate Q-Learning (AQL) agent to compare the result I got from DQN. I trained the AQL agent with 200 episodes and tested 50 episodes. The results are shown in Table C1.





```python
# set up DQN. 2 CNN(ReLu) + 2 FC
class DQN(nn.Module):
    def __init__(self):
        super(DQN, self).__init__()

        self.conv1 = nn.Conv2d(6, 32, kernel_size=3, stride=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1)
        # inPutNodes = 1024
        inPutNodes = (9-2) * (18-2) * 64
        self.fc3 = nn.Linear(inPutNodes, 512)
        self.fc4 = nn.Linear(512, 4)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.fc3(x.view(x.size(0), -1)))
        return self.fc4(x)
```

Figure C1: Diagram of the DQN used in this project and the code to build it.

Agent	DQN	AQL
Training Episodes	8000	200
Training Time	1:45:12 = 6312 second	24.54 second
Test Episodes	200	50
Win Rate	86/200 = 43%	45/50 = 90%
Average Score	582.96	1192.96
Win Episodes Average Score	1015.71	1329.3

Table C1: Training and testing results using DQN and AQL agents.

It is obvious that DQN is both time consuming and less accurate, with 300 times longer training time and half the win rate compared to AQL agent. However, if I upgrade it to three or more layers of CNN and more layers of fully connected layers, it might give prominent results when using the "originalClassic" layout (size 27*27) with 4 ghosts. Currently, we can conclude that neural networks are not worthy of simple tasks, such as the one in this project with two ghosts in an 18*9 map.

References

Mnih, V. et al. “Playing Atari with Deep Reinforcement Learning” *DeepMind Technologies* 2013

Russel, S. Norvig, P. “Artificial Intelligence A Modern Approach” *Prentice Hall PEARSON*, 978-0-13-604259-4, 2010

Ouderaa, T. “PacmanDQN” github.com/tychovdo/PacmanDQN