Pacman by Deep Q-Learning(DQN)

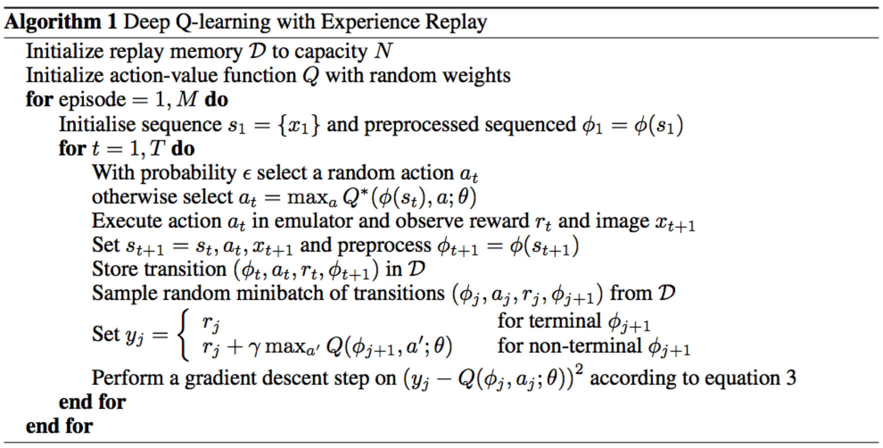
Haiwen Xiao

1 DQN with experience replay

The basic idea of the code is to use the deep q-learning to finish the pacman game. Algorithm implemented here is just following the steps introduced in Deep Mind's paper: *Playing Atari with Deep Reinforcement Learning*. It could be found at *https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf*.

In this project, we store the vector composed by (s_t, a, r, s_{t+1}) into our "experience memory". The memory capacity is 3000. And after the 3000 memory was full, it start to learn by following the q-learning rules. The Q-value is computed by a feed forward neural network with hidden layer. Here we have two network: a target network and a evaluation network. The target network is fixed and updated every 300 iterations by copying the weight of the evaluation network. The evaluation network is updated in each iteration.

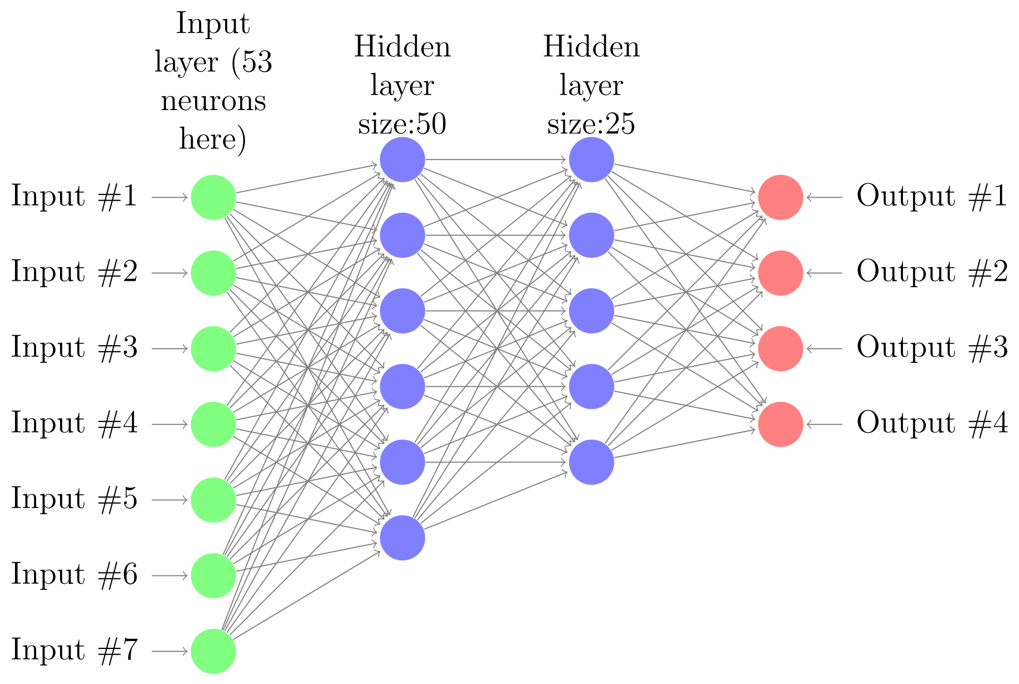
Due to the time limitation for the game finishing, **PyTorch** is used here to implement our neural network due to its fast speed and readability . Adam optimizer is used here for the gradient descent. Batch learning is implemented here to better learn the network, batch size is set to be 512, which means each epoch we extract 512 random examples in the memory to be our training set. The learning rate is set to be 0.01.



```
Algorithm 1 Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode $= 1, M$ do
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      for $t = 1, T$ do
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
          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 $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
          Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
          Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
      end for
  end for
```

Figure 1: DQN with experience replay from Deep Mind's paper

2 Structure of the Neural net work



3 Details of the code

When the game just start, we just pick a random move for the pacman. The input of the network is the features which contains the information on the map: position of the pacman(2 features), position of the ghost(2 features), the state of the pac(-1 is not exist, 1 is exist, 49 features here for smallgrid). Therefore the input dimension of the network for the small grid is 53. Reward for the system is set to be 0.1*score, there exists a special case: when the two pacs both exist in the game, it's not a good idea to eat the one in the middle. Therefore in this situation we set the reward to be the same of the common move and this could make the pacman easier to be win. Another trick in the experience replay is that we need to balance the distribution of the kind in training set. It's easy to know that we win or lose the game by around 20 moves. Then without any processing, the examples of the common move would outnumber the win or lost state. Here we upsmapling the win,lost and eat pac states by 20 and 15 times respectively, so that the training dataset would be balanced.

4 Declaration

The functions for this network is written independently. General style and idea of the code is inspired by the project explained at the official tutorial of PyTorch : ($http://pytorch123.com/SeventhSection/ReinforcementLearning/$).