PROJECT REPORT

UDACITY

DEEP REINFORCEMENT LEARNING NANODEGREE

Project 1: Navigation

AUTHOR:
Thomas Teh

Date: October 8, 2018

1 Learning Algorithms

The deep Q-learning algorithm (DQN) mimics the tabular Q-learning algorithm in order to solve a Markov Decision Process. We know that from dynamic programming and tabular reinforcement learning methods, the optimal action-value function obeys the Bellman Optimality equation below:

$$Q^*(s,a) = \mathbb{E}_{s'\sim\varepsilon}\left[r + \gamma \max_{a'} Q(s',a') \middle| s,a \right].$$

However, in DQN, instead of having a table of action values $Q(s,a)$ for each state, the action-values $Q(s,a)$ are estimated using a function approximator, which is a neural network architecture. The parameters of the Q-network are learned by minimizing the loss function below:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2 \right]$$

where

$$y_i = \mathbb{E}_{s'\sim\varepsilon}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \middle| s,a \right].$$

The optimization of the parameters is usually achieved via gradient descent algorithms (e.g. ADAM, Adagrad, SGD etc) and the gradient of the loss function with respect to the Q-network parameters is given below:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\varepsilon}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i) \right) \nabla_{\theta_i} Q(s,a;\theta_i) \right]$$

It is important to note that in DQN, the parameters are not updated online. Instead, the state-action transitions based on the current action-values are stored in a memory reservoir (experience replay). Samples of the transitions are then sampled in training the agent and updating the respective parameters. The pseudo-code for DQN are given in Algorithm 1.

Initialization:
Initialize replay memory $\mathcal{D}$ to capacity N.
Initialize action-value function $\mathcal{Q}$ with random weights.
for episode = 1:M do
  Initialize sequence $s_1\{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$.
  for t=1:T do
    With probability ϵ select random action a_t
    otherwise select $a_t = \max_a \mathcal{Q}^*(\phi(s_t), a; \theta)$
    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 $(\phi_j, a_i, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max'_a Q(\phi_{j+t}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end
end
Algorithm 1: Deep Q-Learning Networks (DQN) with Experience Replay

1.1 Q-Network: Dueling Q-Network

For the navigation project, the Q-network architecture is shown in Figure 1. The input layer is followed by 3 hidden layers with 128 units of neurons each. Instead of estimating the action-values directly, the last hidden layer is then mapped to the advantage module and the value function module respectively.

We then aggregate the two modules to obtain the estimates of the action-value functions. Mathematically, the two modules are aggregated in the following manner:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \max_{a'} A(s, a'; \theta, \alpha) \right)$$

where $V(s; \theta, \beta)$ is the value function and a scalar value, and $A(s, a; \theta, \alpha)$ is the advantage function and a vector.

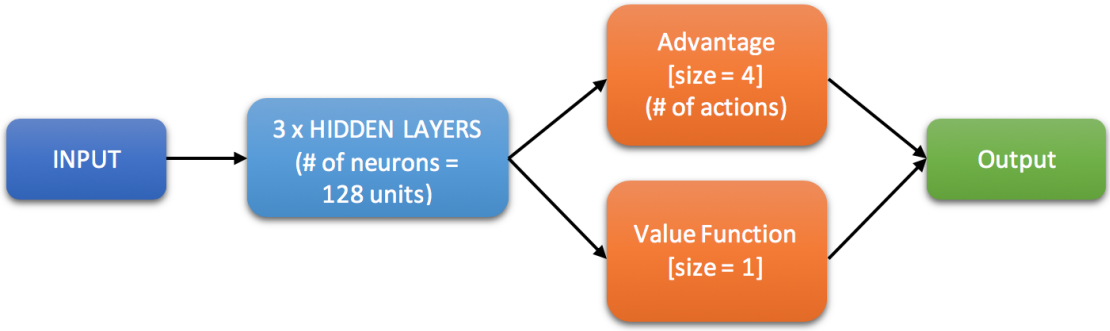

Figure 1: Architecture of the Q-network.

The hyperparameters used in training the Q-network are listed below:

Hyperparameter	Value
Learning rate, α	0.0001
Discount factor, γ	0.9900
Buffer size	500,000
Soft update weights, tau	0.0010
Update frequency	4 steps
$\varepsilon_{initial}$	1.0000
ε_{final}	0.0050
ε- decay	0.9950

The average rewards during the training process is shown in Figure 2. It can be observed that the agent can achieve a mean score of 13 after around 600 episodes. It took around 1800 episodes to achieve a mean score of 17.00.

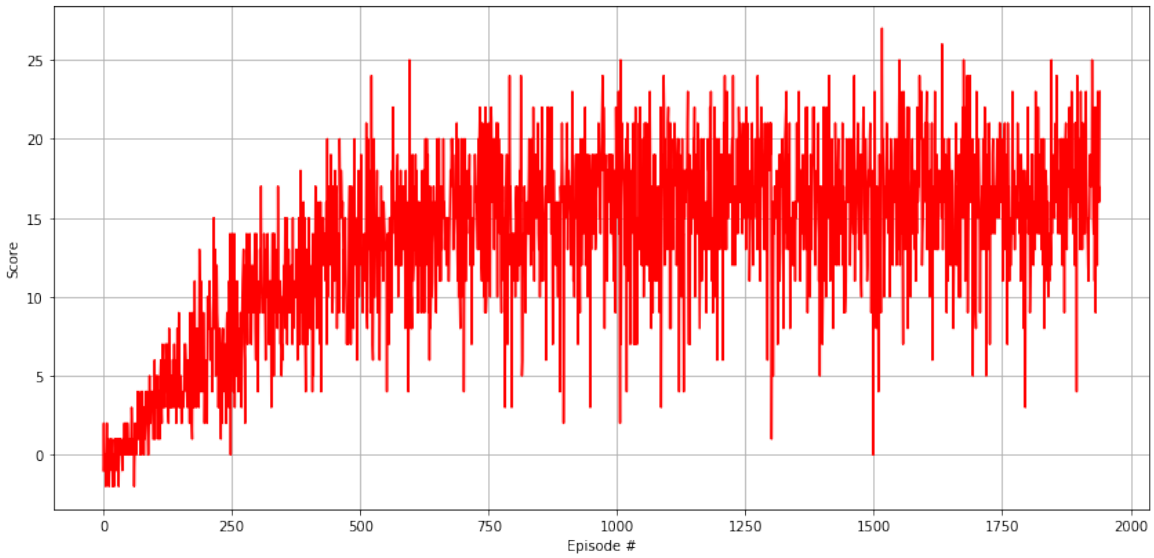

Figure 2: Average reward vs number of episodes

2 Ideas for Future Work

Improvements or redesign of the solution can be done in two aspects: algorithms and problem formulation. In terms of algorithms, we can potentially implement the double-Q learning, prioritized experience replay, or to depart from the estimation of action-values and opt for policy gradient methods instead. As for problem formulation, we can potentially solve the problem by learning directly from the pixels, in which we need to implement computer vision techniques to disentangle the information and patterns from each frame during game play.

2.1 Choice of Algorithms

1. Double-Q Learning: Double Q-learning allows the algorithm to avoid overestimation of the Q-values as well as provide stability in the training process. In order to implement the Double Q-learning, we just need to make some small changes to the original DQN algorithhm. The DQN algorithm is based on the Q-learning algorithm which has the following update equation:

   $$Q(s,a;\theta_i) \leftarrow Q(s,a;\theta_{i-1}) + \alpha\left[r + \gamma \max_{a'} Q(s',a',\theta_{i-1}) - Q(s,a;\theta_i)\right].$$

   In Double-Q Learning, we can change the update equation to the following:

   $$Q(s,a;\theta_i) \leftarrow Q(s,a;\theta_{i-1}) + \alpha\left[r + \gamma Q\left(s', \arg\max_{a'} Q(s',a';\theta_i);\theta_{i-1}\right) - Q(s,a;\theta_i)\right].$$

2. Prioritized Experience Replay: Under the current implementation, the experience replay instances are sampled uniformly. However, it is possible to quickly prioritize the different instances based on the TD error. Prioritize experience replay will shorten the training process by allowing the algorithm to focus on the instances with larger TD errors.

3. Policy Gradient Methods: While the specific policy gradient methods have yet to be introduced, it is possible to us to solve the navigation project using policy gradient methods. Instead of having a function approximator to estimate the action-value functions, the function approximator will directly map to the appropriate function for each given state.

2.2 Learning from Pixels

Lastly, it is possible to learn directly from the pixels of the game play. Several frames of the game play can be stacked and preprocessed in order to retain the temporal information and to reduce the input size respectively. Then preprocessed input can then be passed through a convolutional neural network to extract meaningful feature representations and to estimate the action-value function.

Lastly, it is important to note that the different ideas for future work are not mutually exhaustive. In fact, many of the ideas can be implemented and fine-tuned to achieve state-of-the art performance in the navigation game play.