Udacity's Deep Reinforcement Learning Nanodegree

Report on Navigation Project

Short descriptionon Deep Q-Network

This project demonstrates on how an agent can learn by interacting with the environment using Deep Q-Network reinforcement learning. Deep Q-Network, or DQN, is a kind of Q-Learning which uses deep learning rather than Q-Table because Q-Table is limited once the state space gets large or even continous. As an agent interacts with the environment using the policy, DQN policy is being trained to maximize the expected total rewards.

Additional resource on DQN

- Playing Atari with Deep Reinforcement Learning
- Demystifying Deep Reinforcement Learning
- Wikipedia: Q-learnig

Learning Algorithm

This project adopts well-known DQN techniques to stablize the learning such as;

- Experience Reply - to break the sequential dependency in learning
- Target network - not to have a moving target during learning

The agent is learning the policy using reinforcement learning method by follow the below steps;

1. Choose an action based on the current state using the policy
2. Apply the chosen action to the environment to receive;
    - Reward, which tells whether the action is good or bad,
    - Next state, which is the resulting state after taking the action,
    - Done, which tells whether the episode is finished or not
3. Put the information acquired at Step 2 into the replay buffer, and execute learning algorithm every given steps (in this case, 4 steps)
    - For learning, it randomly picks samples for mini-batch from the replay buffer,
    - Calculate the target return using the target network as;
        - target return = reward + $\gamma max\hat{Q}(nextstate, action, \theta_{target})$
    - Calculate the expected return using the local network as;
        - expected return = $maxQ(state, \theta_{local})$
    - Then, compute loss using MSE between the target and expected returns, and run stochastic gradient descent to update the weights of the local network
        - Loss = MSE(target return, expected return)

- In this implementation, the weights of the target network is being updated gradually rather than copying all the weights of the local network into the target network.
  - $\theta_{target} = \tau \times \theta_{local} + (1 - \tau) \times \theta_{target}$
- Please refer to DQN paper for further detail.

Code snippet of this process is given below and the full code is in Navigation.ipynb and dqn_agent.py

```
# Step 1 ~ Step 2
action = agent.act(state, eps)
env_info = env.step(action)[brain_name]
next_state, reward, done = extract_info(env_info)
agent.step(state, action, reward, next_state, done)
state = next_state
score += reward
if done:
    break

# Step 3
states, actions, rewards, next_states, dones = experiences

Q_targets_next = self.qnetwork_target(next_states).detach().max(1)
[0].unsqueeze(1)
Q_targets = rewards + (gamma * Q_targets_next * (1-dones))
Q_expected = self.qnetwork_local(states).gather(1, actions)

loss = F.mse_loss(Q_expected, Q_targets)
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()

# ------------------- update target network ------------------- #
self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)
```

Network Architecture for Deep Q-Network (DQN)

Neural network for DQN is a multilayer perceptron with 3 hidden layers which has 64 hidden units each. ReLU activation function is applied to each hidden layer. Dropout is defined and applied to the output of activation outputs, but the dropout probability is set as 0 which disables dropout for now.

```
class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=32,
fc2_units=32, fc3_units=32):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, fc3_units)
        self.fc4 = nn.Linear(fc3_units, action_size)
        self.dropout = nn.Dropout(0)

    def forward(self, state):
        x = self.dropout(F.relu(self.fc1(state)))
        x = self.dropout(F.relu(self.fc2(x)))
        x = self.dropout(F.relu(self.fc3(x)))
        return self.fc4(x)
```

Experiments

In order to find the good performing hyper-parameters, 7 experiments have been done by chaning the number of hidden layers, the number of hidden units, batch size, and update step manually. Among 7 experiments, 3 hidden layers (64-64-64 units each) is chosen as a final hyper-parameter because it shows high average score as well as high minimum score.

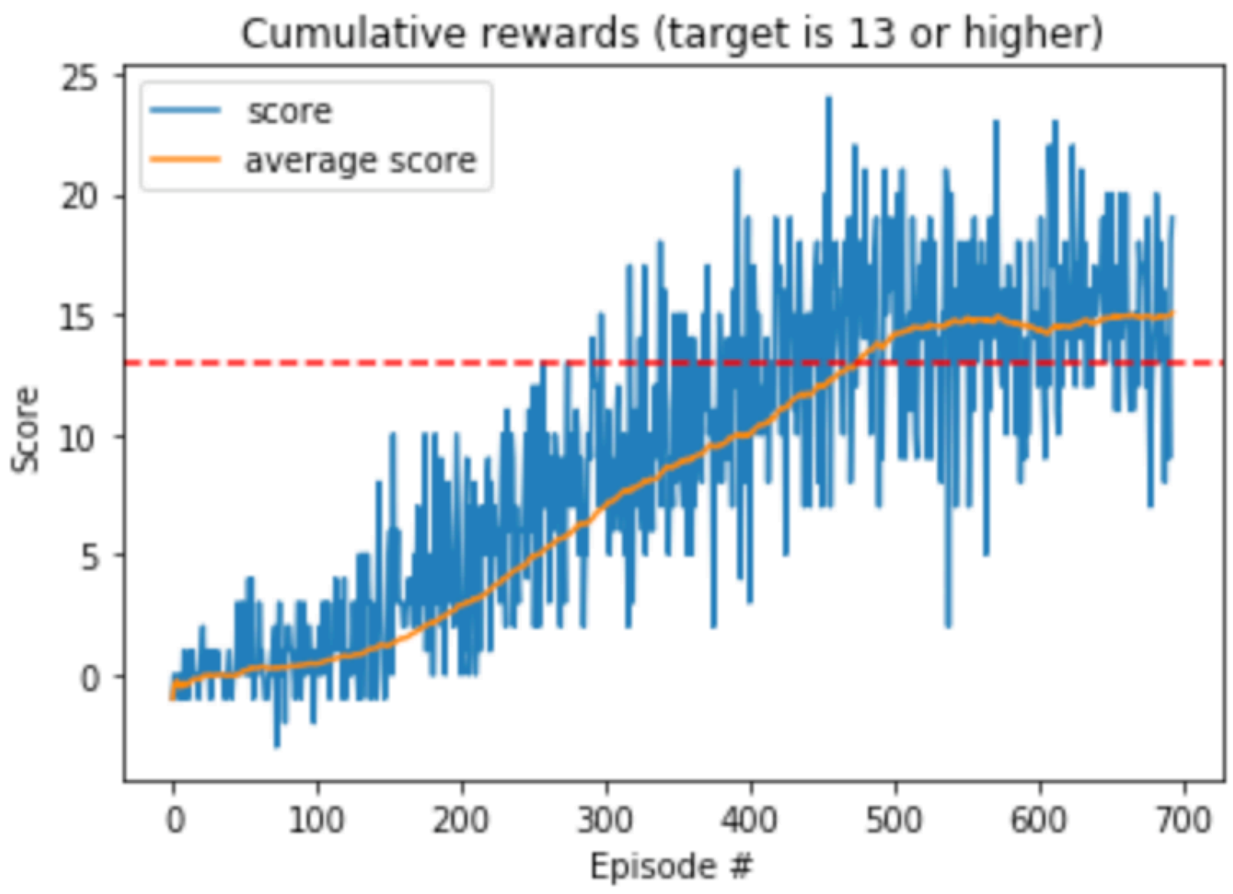
DQN (default batch size: 8, update step: 4)	# of episode	average score	minimum score
2 Hidden layers (32-32 units each)	424	13.02	1.00
3 Hidden layers (32-32-16 units each)	442	13.01	1.00
3 Hidden layers (32-32-32 units each)	449	13.01	1.00
3 Hidden layers (64-32-32 units each)	390	13.00	5.00
3 Hidden layers (64-64-64 units each)	421	13.04	5.00
3 Hidden layers (64-64-64 units each) with batch 128	385	13.01	2.00
3 Hidden layers (64-64-64 units each) with update step 8	451	13.01	2.00

With the selected one, I extended the target score from 13 to 15 in order to train the DQN longer. As a result, it gives higher mininum score as 7.

```
Episode 100 Average Score: 0.45 in 0.79 sec
Episode 200 Average Score: 2.90 in 0.81 sec
Episode 300 Average Score: 6.97 in 0.82 sec
Episode 400 Average Score: 10.02 in 0.82 sec
Episode 500 Average Score: 14.02 in 0.85 sec
Episode 600 Average Score: 14.36 in 0.83 sec
Episode 694 Average Score: 15.05 in 0.82 sec
Environment solved in 594 episodes! Average Score: 15.05    Min Score: 7.00
```

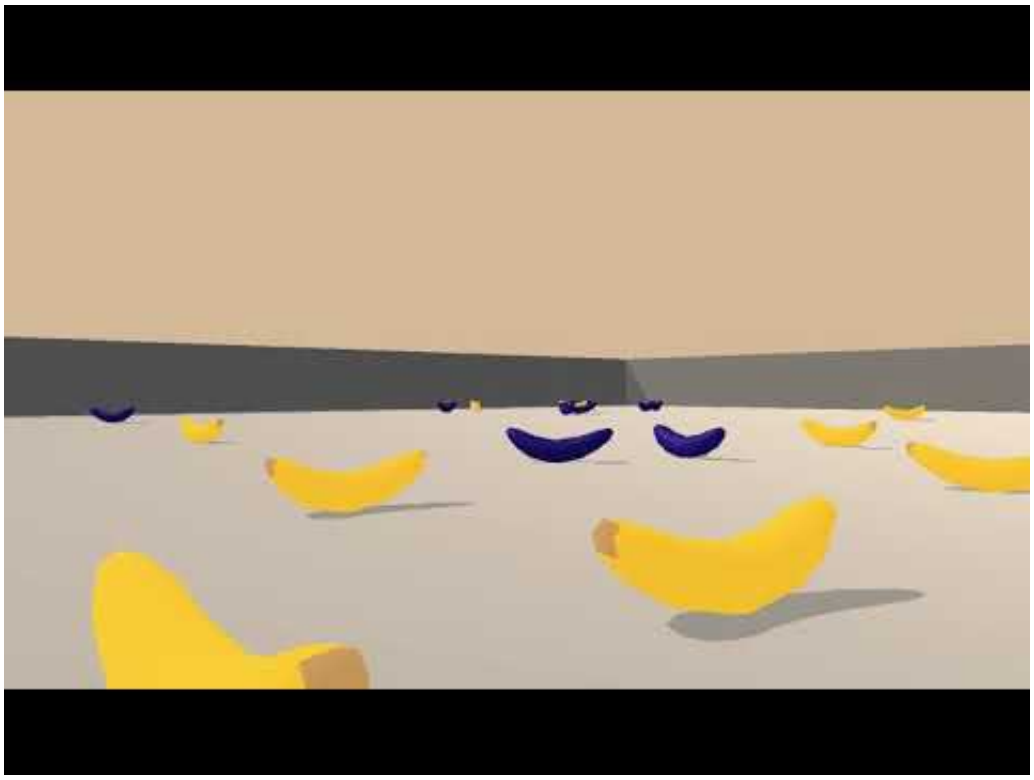
Plot of Rewards

The best performing agent has been trained with 694 episodes which gives the average rewards over previous 100 episodes as 15.05. The below plot shows the cumulative rewards and the average scores.



As seen in the above plot, the average score starts to be saturated after about 500 episodes. Why? I need to investigate further :)

Watch how the trained agent performs



Ideas for Future Work

DQN is believed to be stable as it keeps learning, but it sometimes shows a symptom of catastrophic forgetting, which means that the agent's performance drops significantly after a a period of learing. One of the reason of this failure is that DQN's instability of approximating the Q-function over large-space using Bellman updates.

One of approach to address is this issues to use imatation learning with reinforcement learning, where an expert demonstrates actions, and the policy is pre-trained using this information. In this way, the result can achieve better performance than reinforcement learning.

Reference : Reinforcement and Imitation Learning for Diverse Visuomotor Skills

More curious on more challenging RL task?

Asynchronous Advantage Actor-Critic Agent for Starcraft II