Udacity Deep Reinforcement Learning Project 1 - Navigation

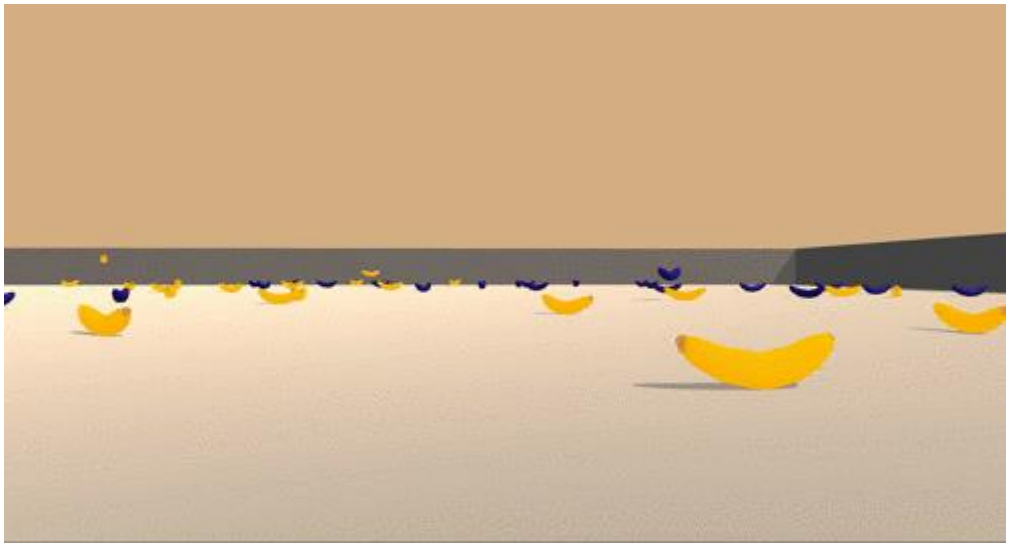
Jonathan Salfity | August, 2018

Navigation

Description

This project featured a single agent learning to navigate and collect bananas in a Unity simulated environment. The agent was trained in a reinforcement learning framework, where the agent learned a policy to collect yellow bananas while avoiding blue bananas. The agent used a Deep Q-Learning approach to learn a policy through a series of episodes. The agent was trained using Deep Q-Learning in a manner very similar to the agent found in 'Human-Level Control Through Deep Reinforcement Learning', Minh et al, 2015.

Unity was used to simulate the environment and provide the reinforcement learning framework with actions, observations, and rewards. Below is a screen capture from the agent's field of view.



Observation Space, Reward Structure, Action Space

The state space has 37 dimensions and contains the agent's velocity, along with the ray-based perception of objects around the agent's forward direction.

Four discrete actions are available: move forward, move backward, turn left, turn right.

A reward of +1 is provided for collecting a yellow banana and a reward of -1 is provided for collecting a blue banana.

Learning Algorithm

Software Framework

A Jupyter notebook hosted on a Udacity server with GPU support was used to write code. PyTorch was used to build and train a deep neural network.

Deep Q-Learning Theory

Q-Learning is a form of temporal difference learning, where the goal is to arrive at a policy which always controls an agent towards the maximum sum of discounted future rewards.

In its simplest form, Q-Learning defines a function, $Q(s,a)$, which represents the utility or value of an agent's state-action pair in the environment.

The Q function is iteratively updated through experiences:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R + \gamma * \max_{a \epsilon A} Q(s',a) - Q(s,a))$$

The reward received by the next state is represented by R. The discount rate is represented by γ. The learning rate is represented by α.

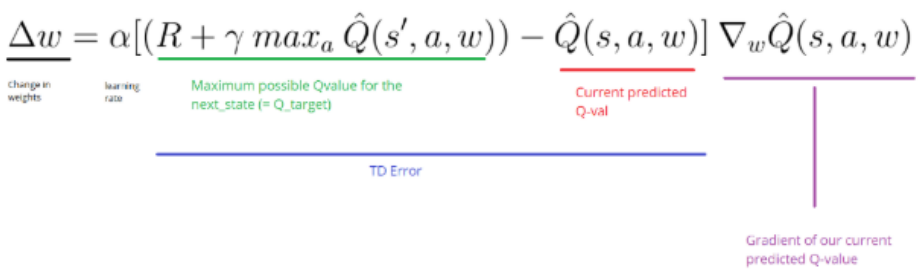
This update rule is then followed by an epsilon greedy action from the Q function:

$$\pi \leftarrow \epsilon greedy(Q)$$

In Deep Q-Learning, a deep neural network was used as a function approximator for the $Q(s,a)$ function. Over the course of experiences, the network associates states with actions that lead to maximum sum of discounted future rewards.

The neural network used here had one input layer, two hidden layers, and one output layer.

The update rule of the weights of the neural network is expressed as:



The TD error, the difference between the maximum possible Q value for the next state and the current predicted Q value, is the main driver affecting the weights.

Prioritized Experience Replay

In this experiment, an experience replay was used to prevent weights from being biased by experience sequence correlation. Keeping track of a replay buffer and using experience replay to sample from the buffer at random, the actions are prevented from oscillating or diverging. The replay buffer contains a collection of experience tuples, (S, A, R, S'). An experience replay samples a small batch of tuples from the replay buffer.

Prioritized experience replay is added because some experiences are more important than others. Priority is assigned to experiences with larger TD error:

$$p = |\delta|$$

Recall TD error is the difference between the maximum possible Q value for the next state and the current predicted Q value:

$$\delta = R' + \gamma * \max_{a \epsilon A} \hat{Q}(S', a, \vec{w}) - \hat{Q}(S', a, \vec{w})$$

Fixed Target Q-Network and Double Q-Learning

The motivation of Double Q-Learning lies in the early stages of training, when Q values are still evolving and not enough information has been gathered from experiences to find the next best action. The accuracy of Q-values depends on which actions has been tried and which neighboring states have been explored. Overestimation of Q-values can occur because the actions are picked over a noisy set of numbers.

Recall the Target:

$$R + \gamma * \max_{a \epsilon A} \hat{Q}(S', a, \vec{w})$$

Expanding the max operator:

$$R + \gamma * \hat{Q}\left\{S', \arg\max_{a \epsilon A} \hat{Q}(S', a, \vec{w}), \vec{w}\right\}$$

This expression shows the Q-value for the state S' and the action that results in the maximum Q-value among all possible actions in that state.

A second Q-value networks is introduced, with weights $\vec{w}^-$:

$$R + \gamma * \hat{Q}\left\{S', \arg\max_{a \epsilon A} \hat{Q}(S', a, \vec{w}), \vec{w}^-\right\}$$

Here, the action is chosen by the network with weights $\vec{w}$ but evaluated by a network with weights $\vec{w}^-$. The second network with weights $\vec{w}^-$ is updated at a slower rate than the first network with weights $\vec{w}$. The two networks work together to agree on the best set of actions.

Deep Neural Network Model Architecture

Agent.py contains the neural network architecture.

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

    def __init__(self, state_size, action_size, seed, fc1_units=64,
fc2_units=64):
        """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, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x
```

PyTorch was used to construct two neural networks used for the local and target Q functions described above. The target and local network had the same structure. The input layer was composed of size of 37 neurons, to match the state size, two hidden layers with 64 neurons each and a final fully connected layer with an output size equivalent to the action size of 4. A Rectified Linear Unit was used in each neuron.

During backpropagation of the network, the loss function was Mean Square Error and the optimizer was Adam.

Training the Network

The following is referring to the `dqn()` function found in `Navigation.py`. This function set up the training architecture.

1. The Q-learning training was first defined with arguments for:
   - n_episodes: how many episodes to run unless environment is solved
   - max_t: number of experiences in one episode
   - eps_start: the epsilon starting value
   - eps_end: the epsilon ending value
   - eps_decay: the epsilon decay rate
   - brain_name: the brain controlling the actions
   - solve: a Boolean describing if the Boolean is solved.

```
def dqn(n_episodes=1000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995,
brain_name=brain_name, solve=False):
```

2. Next the variables for the score of each episode, a container for the score over the last 100 episodes, epsilons initial value are set and the environment is reset to get the initial state information:

```
    scores = []                        # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    eps = eps_start                    # initialize epsilon
    env_info = env.reset(train_mode=True)[brain_name] #reset environment
    state = env_info.vector_observations[0] #get current state
```

3. For each episode:
   - Reset the environment and get the state information:

```
     env_info = env.reset(train_mode=True)[brain_name] #reset environment
     state = env_info.vector_observations[0] #get current state
```

4. Until the episode ends:
   - Get the agent to take an action chosen at random with probability epsilon, or by the greedy policy learned by the online network:

```
    action = agent.act(state, eps)
```

   1. Pass the action to the environment and get the next state:

```
    env_info = env.step(action)[brain_name]
    next_state = env_info.vector_observations[0]
```

   2. Receive a reward from the action

```
reward = env_info.rewards[0]
```

3. Check if the episode is done:

```
done = env_info.local_done[0]
```

4. Step the agent forward 1 step and store the SARS' tuple for learning:

```
agent.step(state, action, reward, next_state, done)
```

5. Set the current state to be the next state:

```
state = next_state
```

6. Update the score received for the most recent reward

```
score += reward
```

7. Check if the episode has ended:

```
if done:
    break
```

8. Update the total scores for the episode and the rolling 100 episode window:

```
scores_window.append(score)       # save most recent score
scores.append(score)              # save most recent score
```

9. Decrease epsilon:

```
eps = max(eps_end, eps_decay*eps) # decrease epsilon
```

10. All episodes are done, return the scores for each episode:

```
return scores
```

HyperParameters

The hyper-parameters were held in the `dqn_agent.py` file. The consisted of:

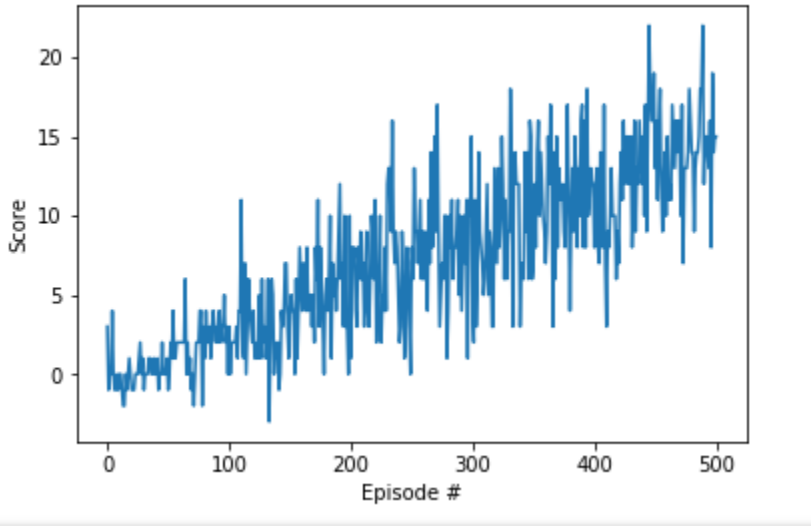
```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network
```

These hyper-parameters were chosen because they were identical to an earlier Udacity project which introduced the student to Q-Learning.

Results

Using the network and parameters described above, the algorithm was able to reach an average score of above 13 in 401 episodes.

```
Episode 100    Average Score: 0.99
Episode 200    Average Score: 4.02
Episode 300    Average Score: 7.47
Episode 400    Average Score: 10.23
Episode 500    Average Score: 12.98
Episode 501    Average Score: 13.05
Environment solved in 401 episodes!  Average Score: 13.05
```



Ideas for Future Work

There is plenty of room to improve the network. The hyper-parameters chosen along with their decay rates, the optimizer, the error function, as well as the network architecture are all tuning knobs that can be changed in order to decrease the number of episodes needed to solve the environment.

If time permits, I'd like to explore Dueling Double Deep Q-Learning, where one network holds the state value and another network holds the advance values.