The “Banana Collector” Agent

Introduction

The agent uses a deep Q-network for reinforcement learning. This means that the agent is trained model-free; it solves the learning task directly using observations from within Unity’s environment without explicitly estimating the reward and transition dynamics from state to state. The algorithm also is off-policy by using an epsilon-greed policy that follows the greedy policy with probability (1-epsilon) and selects a random action with probability epsilon. Hence the agent explores the environment and only later updates the target policy after a batch of experiences have been collected.

Some additional techniques have been employed to improve the stability of the algorithm:

Firstly, it uses a technique called experience replay by storing a set of experiences in a replay memory and random sampling from it is applied to create independent batches for Q-learning updates. This is best practice because owing to the strong correlations between consecutive samples, learning would be inefficient without this technique. Randomizing the experiences breaks their correlations and therefore reduces the variance of the updates.

Secondly, the algorithm uses two independent neural networks as the function approximator for the Q-values. Initially, two Q networks: Q_target and Q_local are created and initialized with random weights. Every k updates Q_local is cloned to obtain Q_target. The latter is then kept constant and is used as a reference for updating the weights of Q_local during the next k steps. Generating Q_target using an older set of parameters adds a delay between the time an update to Q_local is made and the time the update affects the targets, making oscillations much more unlikely.

The above-mentioned deep reinforcement techniques have first been successfully demonstrated in “Human-level control through deep reinforcement learning”.

Implementation

(1) The Q-Network
A class “QNetwork” is defined which creates a neural network with three fully connected layers. The first and second layer each have 64 nodes and the output layer has 4 nodes according to the number of possible actions. The first and second layer are each followed by a RELU activation unit.

```
        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)
```

The function “forward” propagates a state through the network and delivers the Q-values for each possible action.

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

(2) The Replay Buffer

A class "ReplayBuffer" is defined which in this case is initialized with a buffer size of 10^5 experiences. This replay memory holds tuples of (state, action, reward, next_state, done).

```
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_state", "done"])
```

One can add an experience with the "add" function. A random minibatch of experiences can be obtained by the "sample" function. In this case a minibatch size of 64 is used. The replay memory is of type first-in, first-out when the buffer size is exceeded.

```
    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).float().to(device)

        return (states, actions, rewards, next_states, dones)
```

(3) The Agent

A class "Agent" is defined which contains the agent's state and the function "step", "act" and "learn". The agent is initialized with two Q-networks: qnetwork_local and qnetwork_target. Initially these networks have random weights. Also, the replay buffer is instantiated from within the agent's initialization.

```
        # Q-Network
        self.qnetwork_local = QNetwork(state_size, action_size, seed).to(d
evice)
        self.qnetwork_target = QNetwork(state_size, action_size, seed).to(
device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=L
R)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, s
eed)
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0
```

The “step” function adds an experience (i.e. an (state, action, reward, next_state, done) tuple) to the replay memory. Every “UPDATE_EVERY” step (in this case 4) a minibatch of experiences is sampled and the function “learn” is called to update the Q-networks.

```
    def step(self, state, action, reward, next_state, done):
        # Save experience in replay memory
        self.memory.add(state, action, reward, next_state, done)

        # Learn every UPDATE_EVERY time steps.
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:
            # If enough samples are available in memory, get random subset
and learn
            if len(self.memory) > BATCH_SIZE:
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)
```

The “act” function takes a state and propagates it through the qnetwork_local network. It uses the epsilon-greedy policy with probability “1-eps” and a random action with probability “eps”. In this case “eps” is starting at 1 and after each episode decays with factor 0.995. The decay stops if a value of 0.01 is reached.

```
    def act(self, state, eps=0.):
        """Returns actions for given state as per current policy.

        Params
        ======
            state (array_like): current state
            eps (float): epsilon, for epsilon-greedy action selection
        """
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()
```

```
        # Epsilon-greedy action selection
        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))
```

The "learn" function is updating both the qnetwork_local and qnetwork_target weights. For this purpose, a loss function is defined which consists of the mean squared difference between two values: Q_expected minus Q_targets. Averaging is done over a minibatch experience sample (64 in this case). Q_expected contains the results of propagating the states of the minibatch through the qnetwork_local. Q_targets consists of the values obtained by adding the immediate reward of the minibatch experience to the discounted future reward of the next_state action (calculated by propagating next_state through the qnetwork_target network).

In case the optimal policy is reached, both Q_expected and Q_targets are identical according to the Bellmann equations and the above difference would be zero. A minimization job is started by using PyTorch's Adam optimizer on the weights of qnetwork_local.

After Adam has finished the weights of the qnetwork_target are updated by using the original values times "1-tau" plus the new values (obtained from minimizing the squared mean error between Q_targets and Q_expected) times "tau". In this case "tau" is 0.001.

```
    def learn(self, experiences, gamma):
        """Update value parameters using given batch of experience tuples.

        Params
        ======
            experiences (Tuple[torch.Variable]): tuple of (s, a, r, s', do
ne) tuples
            gamma (float): discount factor
        """
        states, actions, rewards, next_states, dones = experiences

        # Get max predicted Q values (for next states) from target model
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)
[0].unsqueeze(1)
        # Compute Q targets for current states
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # Compute loss
        loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

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

(4) Running the Agent
The agent is running 2000 episodes and each of it executed a maximum of 10000 steps.

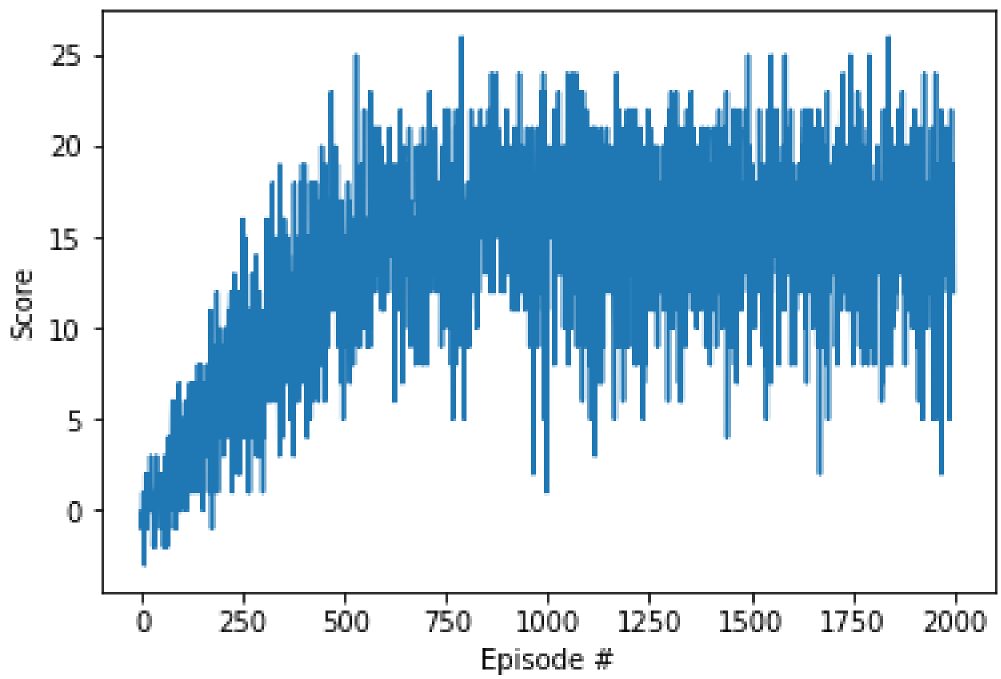
```
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]
        state = env_info.vector_observations[0]
        score = 0
        for t in range(max_t):
            action = agent.act(state, eps)                     # agent propose
s action
            #
            env_info = env.step(action)[brain_name]            # action is exe
cuted in unity environment
            #
            next_state = env_info.vector_observations[0]    # get the next
state
            reward = env_info.rewards[0]                       # get the rewar
d
            done = env_info.local_done[0]
            #
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break
        eps = max(eps_end, eps_decay*eps) # decrease epsilon
```

<u>Results</u>

The following scores were obtained:

```
Episode 100	Average Score: 1.26
Episode 200	Average Score: 4.76
Episode 300	Average Score: 7.79
Episode 400	Average Score: 10.93
Episode 500	Average Score: 13.22
Episode 600	Average Score: 14.47
Episode 700	Average Score: 15.29
Episode 800	Average Score: 14.99
Episode 882	Average Score: 16.00
Environment solved in 782 episodes!	Average Score: 16.00
Episode 900	Average Score: 16.40
Episode 1000	Average Score: 15.98
Episode 1100	Average Score: 16.54
Episode 1200	Average Score: 15.65
Episode 1300	Average Score: 15.53
Episode 1400	Average Score: 15.55
Episode 1500	Average Score: 16.01
Episode 1600	Average Score: 15.58
Episode 1700	Average Score: 15.28
Episode 1800	Average Score: 15.31
Episode 1900	Average Score: 15.66
Episode 2000	Average Score: 15.56
```

After 782 episodes the agent obtained already a score >16 and remained close to this performance thereafter. The following plot shows the full score history per episode.



Further improvements

One obvious improvement is to increase the resolution of the actions, i.e. 16 or 32 possible directions for movement. This will allow the agent to improve the precision with which it can target or avoid bananas. With the current model the agent path is more like the one of a “drunken” agent because targeting with 4 possible actions is difficult and results in a zig-zag movement. This lack of precision may also explain why the score per episode is still fluctuating significantly.

Another straightforward improvement of the above algorithm would be to use “prioritized experience replay” as describe here, for example. This method is a kind of variance reduction method in which those experiences which have a higher deviation between Q_target and Q_expected are given more weight or a higher probability when sampling the experience minibatches (in the above solution all experiences were equally likely to be selected).