Navigation Project

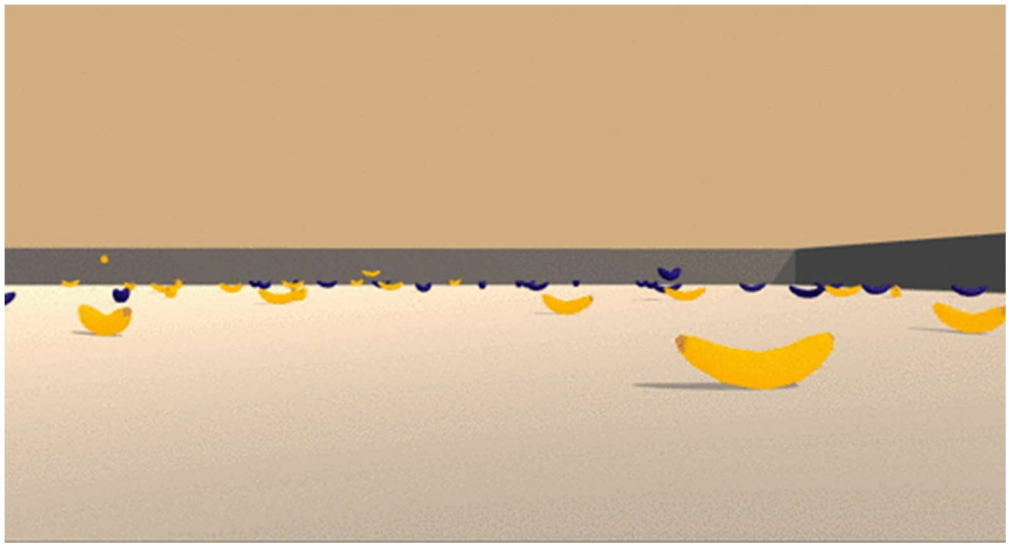
Udacity Deep Reinforcement Learning

Dhar Rawal (dharrawal@gmail.com)

1/1/2021

Introduction

The goal of this project is to train a deep RL agent to navigate a Unity virtual environment consisting of a large, flat, square world and collect yellow bananas while avoiding blue bananas



There is a reward of +1 for collecting a yellow banana and a reward of -1 for for collecting a blue banana. The state space has 37 dimensions and contains the agent's velocity as well as a ray-based perception of objects around the agent's forward direction. The agent has to learn to select from one of 4 actions (0-3)

The task is episodic and lasts a few hundred time steps. Success will be judged by whether the agent can score 13+ per episode averaged over 100 consecutive episodes.

Learning Algorithm

The learning algorithm is a deep-Q network that uses an experience replay buffer to train a deep neural network to follow delayed (fixed) Q-Targets generated using a copy of the network from the prior training pass. The agent is not trained at every step; it is trained only when enough steps have been taken to fill up the experience replay buffer completely. Refer to the paper titled Human-level control through deep reinforcement learning for details on this algorithm.

Q-Network Architecture

Pytorch is used to implement the below deep neural network in a class called *QNetwork*:

1. nn.Linear (37 => 64)

2. nn.Linear (64 => 64)
3. nn.Linear (64 => 4)

All 3 layers have their weights initialized using xavier_uniform_ distribution. ***This step is critical to getting good results***. The Adam optimizer is used for its momentum and automatic learning rate adjustment capabilities.

The RL Agent

The RL agent is implemented in a class called *Agent*. It maintains a couple of instances of the *QNetwork* class – one that is actively used for selecting actions, and the other to maintain a copy of weights from the immediately prior batch training call.

```
self.qnetwork_local = QNetwork(state_size, action_size, seed, dense1_size, dense2_size).to(device)

self.qnetwork_target = QNetwork(state_size, action_size, seed, dense1_size, dense2_size).to(device)
```

ReplayBuffer

The agent also maintains the replay buffer which is as follows:

```
self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
```

Here we use a BUFFER_SIZE of 100,000 to give us enough space to store lots of episodes worth of data and a BATCH_SIZE of 64, so the data from prior 64 steps would be used for training the agent.

This replay buffer is updated at every step and when BATCH_SIZE worth of data is gathered, training begins. Training is then done every UPDATE_EVERY steps (set at 4)

```
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 ReplayBuffer class provides a sample function that returns a batch of experience data sampled according to a uniform random distribution as follows:

```
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)
```

The learn function

This is the guts of the Agent and how it learns! It basically uses Q-learning to determine a TD loss that is then used to train the *QNetwork*. The guts of this algorithm calculates a delayed ("fixed") target using a prior snapshot of the QNetwork.

```
        # forward pass to get outputs

        Qs = self.qnetwork_local(states)


        #print("Qs: ", Qs.data.size())

        Qsa = Qs.gather(1, actions)

        #print("Qsa: ", Qsa.data.size())


        Qns = self.qnetwork_target(next_states)

        Qns = Qns.detach()    # since we are not going to use above forward pass for training, remove it from history

        Qmax_ns = Qns.max(1)[0]  # get max Qns along axis 1. [0] because we only want max value, not argmax

        Qmax_ns = Qmax_ns.unsqueeze(1)   # we get [batch_size], unsqueeze to [batch_size, 1]


        TD_Target = rewards + gamma*Qmax_ns*(1-dones)   # multiply by (1-dones): Qmax_ns = 0 for terminal state


        # calculate the loss using predicted and target action values
```

```
    loss = F.mse_loss(Qsa, TD_Target)
```

As the last step, it calls the *soft_update* function to update the prior snapshot of the QNetwork using the latest iteration of the QNetwork

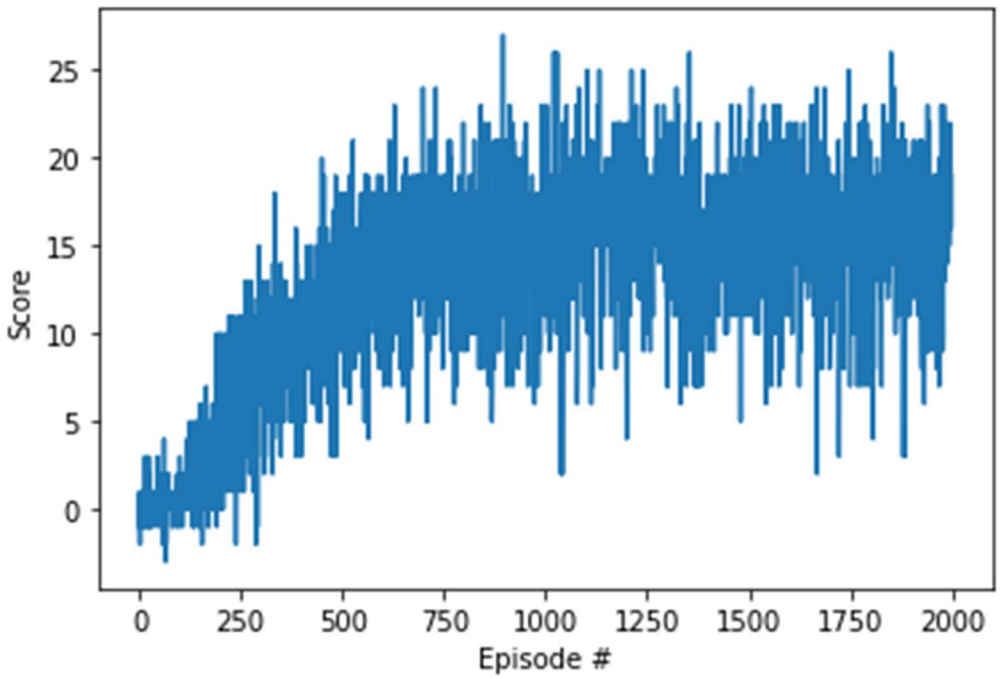
```
def soft_update(self, local_model, target_model, tau):
    """Soft update model parameters.
    ϑ_target = τ*ϑ_local + (1 - τ)*ϑ_target

    Params
    ======
        local_model (PyTorch model): weights will be copied from
        target_model (PyTorch model): weights will be copied to
        tau (float): interpolation parameter
    """
    for target_param, local_param in zip(target_model.parameters(),
local_model.parameters()):
        target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)
```

A value of 0.001 is used for TAU. This value assigns very little importance to the latest iteration of the QNetwork and maximum importance to the prior snapshot of the QNetwork, thus ensuring that the weights update very gradually and the learning improvement is stable.

Plot of Rewards (Performance)

The agent performs relatively well, learning to score 13+ per episode, averaged over 100 consecutive episodes within 500-700 episodes. Below is a snapshot of learning progress from one such run.



When this trained RL agent was tested, it achieved a score of 22 in the episode!

```
# lets see how well our agent performs
agent = Agent(state_size=37, action_size=4, seed=0, dense1_size=64, dense2_size=64)

map_location='cpu'

# load the weights from file
agent.qnetwork_local.load_state_dict(torch.load('checkpoint.pth',
map_location=map_location))

env_info = env.reset(train_mode=False)[brain_name] # reset the environment
state = env_info.vector_observations[0]            # get the current state
score = 0                                          # initialize the score
while True:
   action = agent.act(state).astype(int)       # select an action
   env_info = env.step(action)[brain_name]       # send the action to the environment
   next_state = env_info.vector_observations[0]   # get the next state
```

```
    reward = env_info.rewards[0]                # get the reward
    done = env_info.local_done[0]               # see if episode has finished
    score += reward                             # update the score
    state = next_state                          # roll over the state to next time step
    if done:                                    # exit loop if episode finished
        break

print("Score: {}".format(score))
```

Ideas for Future Work (Recommendations)

The next logical challenges in the project are to implement:

1. Double DQN
2. Prioritized Experience Replay
3. Dueling DQN and finally
4. Rainbow DQN