Project 1 – Navigation

May 7, 2020

1 Introduction and objectives

For this project, an agent is trained to navigate and collect bananas in a large, square world. The primary objective is to get an average score of +13 over 100 consecutive episodes. The Deep Q-network and Double Q-network are utilized for the task. The project is completed in the *Udacity Workspaces*.

2 Environment

An agent is trained to navigate and collect bananas in a large, square world. A reward of +1 is provided for collecting a yellow banana, and a reward of −1 is provided for collecting a blue banana. Thus, the goal of the agent is to collect as many yellow bananas as possible while avoiding blue bananas.

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to :

0 – move forward.

1 – move backward.

2 – turn left.

3 – turn right.

3 Deep Q-Learning

The Deep Q-learning algorithm is performed to train the agent as follows:

Algorithm 1: Deep Q-Learning

Initialize an an empty list $scores = [\,]$
for $t \leftarrow 1$ **to** *number of episode+1* **do**
  $score \leftarrow 0$
  $\epsilon = \epsilon_{\text{start}}$
  for $t \leftarrow 0$ **to** *number of step-1* **do**
    Execute action A_t based on epsilon-greedy action selection
    Observe reward R_{t+1} and state S_{t+1}
    Evaluate target $\tilde{Q}_\theta = R_{t+1} + \max_a Q_\theta(S_{t+1}, a)$
    Compute gradient descent by minimizing $\left(\tilde{Q}_\theta - Q_\theta(S_{t+1}, a)\right)^2$
    $score \leftarrow score + R_{t+1}$ and append to the list $scores$
  $\epsilon = \max(\epsilon_{\text{end}}, \epsilon_{\text{decay}}\epsilon)$
return *scores*

The local and target Q-Networks is constructed using 3 hidden fully connected layer neural network. The loss is minimized by using the RMSprop optimizer. A learning rate decay is included to stabilize the minimization:

Code Listing 1: Q-network

```
self.qnetwork_local = QNetwork(state_size, action_size, seed,fc1_units, fc2_units,
                                            fc3_units).to(device)
self.qnetwork_target = QNetwork(state_size, action_size, seed,fc1_units, fc2_units,
                                            fc3_units).to(device)
self.optimizer = optim.RMSprop(self.qnetwork_local.parameters(), lr=LR)
self.lr_scheduler = optim.lr_scheduler.ExponentialLR(self.optimizer, lr_decay)
```

The model architecture for the present neural networks which consist of 3 hidden fully connected layer neural network with Relu activation function:

Code Listing 2: Model architecture

```
def __init__(self, state_size, action_size, seed,fc1_units=128, fc2_units=256,
                                            fc3_units=32):
  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)

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 = F.relu(self.fc3(x))
  return self.fc4(x)
```

The action is evaluated based on epsilon-greedy action selection:

Code Listing 3: 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))
```

A replay buffer is used to retain random previous experiences. This buffer is regularly fed to the network as learning material:

Code Listing 4: Replay buffer

```
e = self.experience(state, action, reward, next_state, done)
self.memory.append(e)
```

4 Double Deep Q-Learning

The risk of using a single Q-network is that it may over-estimate the Q-values, leading to sub-optimal policies. Two networks are used jointly, i.e., a Deep Q-network is used to select the best action, while the other one is used to estimate the target value:

Code Listing 5: Double Q-Learning

```
# best actions based on local network
Q_local_next_states_values = self.qnetwork_local(next_states).detach()
Q_local_best_actions = Q_local_next_states_values.max(1)[1]

# best actions using target network
Q_target_next_states_values = self.qnetwork_target(next_states).detach()
Q_targets_next = Q_target_next_states_values.gather(1, Q_local_best_actions.unsqueeze(
                                                    1))
```

5 Training

After some hyperparameter searches, the following parameters are used to train the agent:

128, 32, and 32 nodes of first, second, and third fully connected layer, respectively;

learning rate $= 5e-4$, learning rate decay $= 0.9999$;

replay buffer size $= 100$, minibatch size $= 32$, discount factor $= 0.99$;

network update$= 4$, soft update of target parameters $= 1e-2$;

maximum step of 500, $\epsilon_{\text{start}} = 0.2$, $\epsilon_{\text{end}} = 0.01$, $\epsilon_{\text{decay}} = 0.99$.

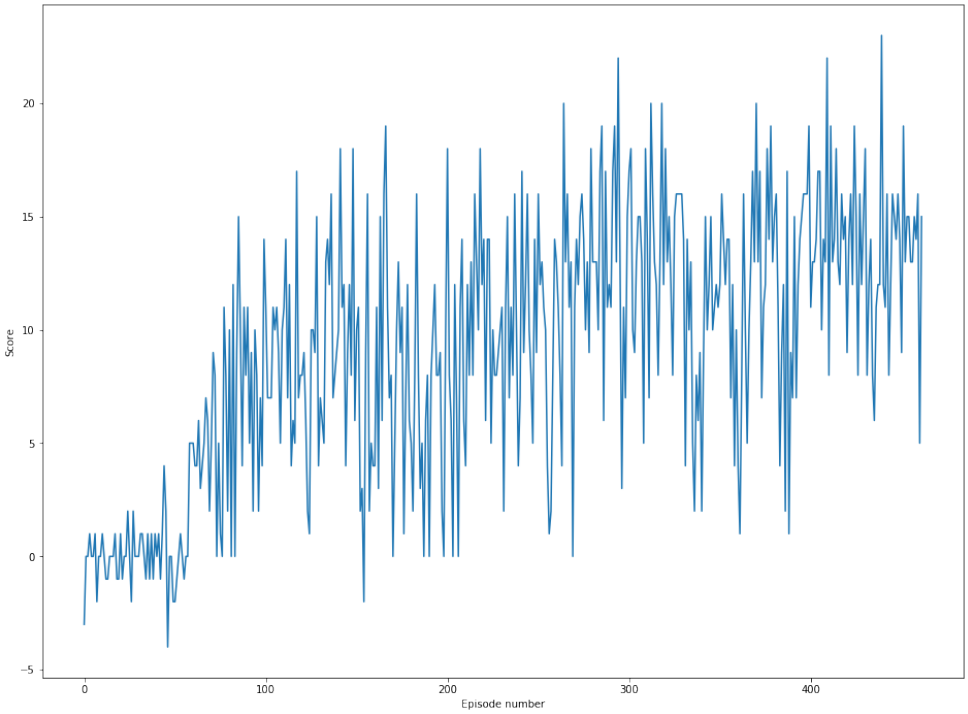

Figure 1: Average of score

Code Listing 6: Deep Q-network training

```
agentdqn = Agent(state_size=37, action_size=4, seed=0, ddqn=False,fc1_units=128,
                                        fc2_units=32, fc3_units=32,lr_decay=0.9999)
scoresdqn = dqn(agentdqn, 1000, 500, 0.2, 0.01, 0.99)

Episode 100 Average Score: 6.78
Episode 200 Average Score: 8.68
Episode 300 Average Score: 9.03
Episode 400 Average Score: 11.22
Episode 500 Average Score: 12.15
Episode 589 Average Score: 13.04
Environment solved in 489 episodes! Average Score: 13.04
```

For double deep Q-network, the same parameters are applied:

Code Listing 7: Double deep Q-network training

```
agentdqqn = Agent(state_size=37, action_size=4, seed=0, ddqn=True,fc1_units=128,
                                        fc2_units=32, fc3_units=32,lr_decay=0.9999)
scores = dqn(agentdqqn, 1000, 500, 0.2, 0.01, 0.99)

Episode 100 Average Score: 2.53
Episode 200 Average Score: 8.20
Episode 300 Average Score: 10.90
Episode 400 Average Score: 11.94
Episode 462 Average Score: 13.11
Environment solved in 362 episodes! Average Score: 13.11
```

As a result, the deep Q-network **solved in 489 episodes with an average score of +13.04 over 100 consecutive episodes**. The double deep Q-network performed better and **solved in 362 episodes with an average score of +13.11 over 100 consecutive episodes** as shown in Fig 1.

6 Test the agent

Finally, the double Q-network agent is tested against the environment over 10 episodes:

Code Listing 8: Test the double deep Q-network agent

```
scores = []
scores_window = deque(maxlen=100)
  for i_episode in range(1, 10+1):
    state = env.reset(train_mode=False)[brain_name].vector_observations[0]
    score = 0
    for t in range(500):
      action = agentdqqn.act(state)
      env_info = env.step(action)[brain_name]
      next_state = env_info.vector_observations[0]
      reward = env_info.rewards[0]
      done = env_info.local_done[0]
      state = next_state
      score += reward
      if done:
      break
      scores_window.append(score)
      scores.append(score)
      print('\rEpisode {}\tScore: {:.2f}'.format(i_episode, scores[-1]))

Episode 1 Score: 9.00
Episode 2 Score: 12.00
Episode 3 Score: 16.00
Episode 4 Score: 17.00
Episode 5 Score: 15.00
Episode 6 Score: 9.00
Episode 7 Score: 15.00
Episode 8 Score: 12.00
Episode 9 Score: 17.00
Episode 10  Score: 18.00
```

Overall, the performance is as expected (average over 13). Notice that in rare case, the agent perform poorly in certain episodes, such as 1 and 6. We discuss some future works to improve the agent in the following section.

7 Future work

1. **Hyperparameter search**: There are still many combinations of hyperparameter to be explored. This study focused on tuning the number of nodes in neural network, replay buffer size and ϵ. Other parameters remain unexplored are minibatch size, discount factor, soft update of target parameters.

2. **Prioritized replay buffer**: In the exploration of the environment, some experiences might occur rarely while being of high importance. However, these experiences may never be selected as the memory replay batch is sampled uniformly. We may sample the replay buffer according to the training loss value, in order to fill the buffer in priority with experiences where the Q-value evaluation was poor, so that the network will have better training on "unexpected events".

3. **Dueling Deep Q-Network:** Dueling deep Q-network represents two separate estimators, i.e., one for the state value function and one for the state-dependent action advantage function. By decoupling the estimation, we are able to generalize the learning across actions without imposing any change to the underlying algorithm.