myNavigation

May 20, 2020

1 Navigation

You are welcome to use this coding environment to train your agent for the project. Follow the instructions below to get started!

1.0.1 0. Reference

This code closely follows the DQN implementation in https://github.com/udacity/deep-reinforcement-learning/tree/master/dqn

1.0.2 1. Start the Environment

Run the next code cell to install a few packages. This line will take a few minutes to run!

```
In [1]: !pip -q install ./python
```

```
tensorflow 1.7.1 has requirement numpy>=1.13.3, but you'll have numpy 1.12.1 which is incompatib
ipython 6.5.0 has requirement prompt-toolkit<2.0.0,>=1.0.15, but you'll have prompt-toolkit 3.0.
```

The environment is already saved in the Workspace and can be accessed at the file path provided below. Please run the next code cell without making any changes.

```
In [2]: from unityagents import UnityEnvironment
        import numpy as np

        # please do not modify the line below
        env = UnityEnvironment(seed=0,file_name="/data/Banana_Linux_NoVis/Banana.x86_64")
```

```
INFO:unityagents:
'Academy' started successfully!
Unity Academy name: Academy
        Number of Brains: 1
        Number of External Brains : 1
        Lesson number : 0
        Reset Parameters :
```

```
Unity brain name: BananaBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 37
        Number of stacked Vector Observation: 1
        Vector Action space type: discrete
        Vector Action space size (per agent): 4
        Vector Action descriptions: , , ,
```

Environments contain ***brains*** which are responsible for deciding the actions of their associated agents. Here we check for the first brain available, and set it as the default brain we will be controlling from Python.

```
In [3]: # get the default brain
        brain_name = env.brain_names[0]
        brain = env.brains[brain_name]
```

1.0.3 2. The Value-Action Function

Create the Value-Action Function with three fully connected hidden layers.

```
In [7]: import torch
        import torch.nn as nn
        import torch.nn.functional as F

        class QNetwork(nn.Module):

            def __init__(self, state_size, action_size, seed, hidden=[512, 512, 256]):
                """Initialize parameters and build model.
                Params
                ======
                    state_size (int): Dimension of each state
                    action_size (int): Dimension of each action
                    seed (int): Random seed
                    fc1_units (int): Number of nodes in first hidden layer
                    fc2_units (int): Number of nodes in second hidden layer
                """
                super(QNetwork, self).__init__()
                self.seed = torch.manual_seed(seed)
                self.dense1 = nn.Linear(state_size, hidden[0])
                self.dense2 = nn.Linear(hidden[0], hidden[1])
                self.dense3 = nn.Linear(hidden[1], hidden[2])
                self.output = nn.Linear(hidden[2], action_size)

                self.drops = nn.Dropout(0.2)

            def forward(self, state):
```

```
        """Build a network that maps state -> action values."""
        state = F.relu(self.drops(self.dense1(state)))
        state = F.relu(self.drops(self.dense2(state)))
        state = F.relu(self.drops(self.dense3(state)))
        return F.softmax(self.output(state))
```

1.0.4 3. The Agent

Create the Agent with a Replay Buffer and Soft Update.

```
In [9]: import numpy as np
        import random
        from collections import namedtuple, deque
        import torch.optim as optim

        BUFFER_SIZE = int(1e5)  # replay buffer size
        BATCH_SIZE = 256         # minibatch size
        GAMMA = 0.99            # discount factor
        TAU = 1e-3              # for soft update of target parameters
        LR = 2e-5               # learning rate
        UPDATE_EVERY = 8        # how often to update the network

        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

        class Agent():
            """Interacts with and learns from the environment."""

            def __init__(self, state_size, action_size, seed):
                """Initialize an Agent object.

                Params
                ======
                    state_size (int): dimension of each state
                    action_size (int): dimension of each action
                    seed (int): random seed
                """
                self.state_size = state_size
                self.action_size = action_size
                self.seed = random.seed(seed)

                # Q-Network
                self.qnetwork_local = QNetwork(state_size, action_size, seed).to(device)
                self.qnetwork_target = QNetwork(state_size, action_size, seed).to(device)
                self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

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

```python
        self.t_step = 0

    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)

    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))

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

        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) 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(
        # 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)

    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        _target = *_local + (1 - )*_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.para
            target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)


class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.

        Params
        ======
            action_size (int): dimension of each action
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
        """
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "rewa
        self.seed = random.seed(seed)

    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 No
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None

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

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

1.0.5 4. Setting up the environment.

- Loading Pytorch
- Reseting the environment
- Creating the Agent

In [10]:
```
# import random
# import torch
# import numpy as np
# from collections import deque
# from myAgent import Agent
import matplotlib.pyplot as plt
%matplotlib inline

# Setting up the environment
env_info = env.reset(train_mode=True)[brain_name] # reset the environment
state = env_info.vector_observations[0]

actionSize = brain.vector_action_space_size
stateSize = len(state)
agent = Agent(state_size = stateSize, action_size = actionSize, seed = 0)
```

1.0.6 5. Training the network.

In [11]:
```
# def dqn(n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):
def dqn(n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.997):
    """Deep Q-Learning.
```

```
        Params
        ======
            n_episodes (int): maximum number of training episodes
            max_t (int): maximum number of timesteps per episode
            eps_start (float): starting value of epsilon, for epsilon-greedy action selecti
            eps_end (float): minimum value of epsilon
            eps_decay (float): multiplicative factor (per episode) for decreasing epsilon
        """
        scores = []                        # list containing scores from each episode
        scores_window = deque(maxlen=100)  # last 100 scores
        eps = eps_start                    # initialize epsilon
        for i_episode in range(1, n_episodes+1):
            env_info = env.reset(train_mode=True)[brain_name] # reset the environment
            state = env_info.vector_observations[0]
            score = 0
            for t in range(max_t):

                action = agent.act(state, eps)

                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]

                agent.step(state, action, reward, next_state, done)
                state = next_state
                score += reward
                if done:
                    break
            scores_window.append(score)       # save most recent score
            scores.append(score)              # save most recent score
            eps = max(eps_end, eps_decay*eps) # decrease epsilon
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_wi
            if i_episode % 100 == 0:
                print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(score
            if np.mean(scores_window)>=15.0:
                print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.forma
                torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
                break
        return scores

    scores = dqn()
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:38: UserWarning: Implicit dimension


Episode 100	Average Score: 0.49
Episode 200	Average Score: 1.95
```

```
Episode 300	Average Score: 3.54
Episode 400	Average Score: 5.63
Episode 500	Average Score: 7.03
Episode 600	Average Score: 6.12
Episode 700	Average Score: 5.71
Episode 800	Average Score: 7.54
Episode 900	Average Score: 8.23
Episode 1000	Average Score: 10.21
Episode 1100	Average Score: 12.52
Episode 1200	Average Score: 12.90
Episode 1300	Average Score: 13.72
Episode 1400	Average Score: 13.99
Episode 1500	Average Score: 14.58
Episode 1521	Average Score: 15.05
Environment solved in 1421 episodes!	Average Score: 15.05
```

In [12]:
```
# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```