Navigation

September 24, 2019

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 1. Start the Environment

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

```
In [3]: !pip install --upgrade numpy

Collecting numpy
  Downloading https://files.pythonhosted.org/packages/e5/e6/c3fdc53aed9fa19d6ff3abf97dfad768ae3a
    100% || 20.4MB 2.1MB/s eta 0:00:01
Installing collected packages: numpy
  Found existing installation: numpy 1.12.1
    Uninstalling numpy-1.12.1:
      Successfully uninstalled numpy-1.12.1
Successfully installed numpy-1.17.2
```

```
In [ ]: !pip install 'prompt-toolkit<2.0.0,>=1.0.15'
```

```
In [ ]: !pip3 install --upgrade --user ipython
```

```
In [4]: !pip -q install ./python
```

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 [5]: from unityagents import UnityEnvironment
        import numpy as np

        # please do not modify the line below
        env = UnityEnvironment(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 [6]:
```
# get the default brain
brain_name = env.brain_names[0]
brain = env.brains[brain_name]
```

1.0.2 2. Examine the State and Action Spaces

Run the code cell below to print some information about the environment.

In [7]:
```
# reset the environment
env_info = env.reset(train_mode=True)[brain_name]

# number of agents in the environment
print('Number of agents:', len(env_info.agents))

# number of actions
action_size = brain.vector_action_space_size
print('Number of actions:', action_size)

# examine the state space
state = env_info.vector_observations[0]
print('States look like:', state)
state_size = len(state)
print('States have length:', state_size)
```

```
Number of agents: 1
Number of actions: 4
States look like: [1.         0.         0.         0.         0.84408134 0.
```

```
 0.          1.          0.          0.0748472   0.          1.
 0.          0.          0.25755     1.          0.          0.
 0.          0.74177343  0.          1.          0.          0.
 0.25854847  0.          0.          1.          0.          0.09355672
 0.          1.          0.          0.          0.31969345  0.
 0.        ]
States have length: 37
```

1.0.3 3. Take Random Actions in the Environment

In the next code cell, you will learn how to use the Python API to control the agent and receive feedback from the environment.

Note that **in this coding environment, you will not be able to watch the agent while it is training**, and you should set `train_mode=True` to restart the environment.

```
In [8]: env_info = env.reset(train_mode=True)[brain_name] # reset the environment
        state = env_info.vector_observations[0]            # get the current state
        score = 0                                          # initialize the score
        while True:
            action = np.random.randint(action_size)        # 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 st
            if done:                                       # exit loop if episode finished
                break

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

```
Score: 0.0
```

When finished, you can close the environment.

```
In [9]: env.close()
```

1.0.4 4. It's Your Turn!

Now it's your turn to train your own agent to solve the environment! A few **important notes**: - When training the environment, set `train_mode=True`, so that the line for resetting the environment looks like the following:

```
env_info = env.reset(train_mode=True)[brain_name]
```

- To structure your work, you're welcome to work directly in this Jupyter notebook, or you might like to start over with a new file! You can see the list of files in the workspace by clicking on ***Jupyter*** in the top left corner of the notebook.

- In this coding environment, you will not be able to watch the agent while it is training. However, ***after training the agent***, you can download the saved model weights to watch the agent on your own machine!

1.1 Import libraries

Classic libraries for reinforcement learning problem. The most important ones are PyTorch torch which is responsible for a neural network and unityagents which is responsible for an environment.

```
In [4]: import numpy as np
        import random
        from collections import namedtuple, deque
        import torch
        import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim
        import matplotlib.pyplot as plt
        %matplotlib inline
        from unityagents import UnityEnvironment
        import numpy as np
```

1.2 Feed directory for an environment

We do not need to install Unity for running our environment. Environment and all dependencies are provided and should be downloaded to a directory. We just need to point on the path of those files. Link to the files that have to be downloaded are in README.md file at the root of repositary.

```
In [5]: env = UnityEnvironment(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: , , ,
```

```
In [6]: #Default parameters are provided. We just need to correctly reference those parameters.

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

```
In [7]: # Tuning parameters of our neural network
        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
        device = "cpu"
```

```
In [8]: # neural network which consists of 4 Linear layers responsible for an agent behaviour
        # had to tune neural network for getting good performance
        class QNetwork(nn.Module):
            """Actor (Policy) Model."""

            def __init__(self, state_size, action_size, seed, fc1_units=64, fc2_units=128, fc3_u
                """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.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))
                x = F.relu(self.fc4(x))
                return x
```

1.3 Replay buffer

Replay buffer keeps history of different states, actions, rewards, next states and done parameter. Two most important methods or Replay Buffer class are add which adds to replay buffer data from the agent and sample which gets random sample of data for the agent. The reason why it is a random sample is that our agent have to avoid memorizing sequences rather it should react to different states accordingly.

```
In [9]: 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)

In [10]: 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)
        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.par
            target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)
```

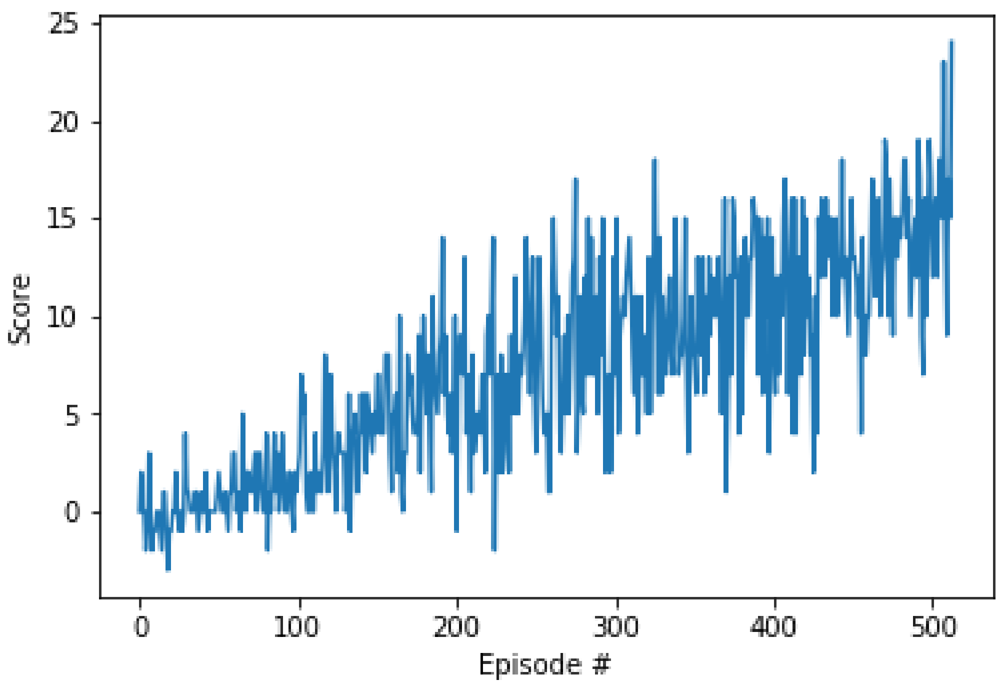
```
In [11]: agent = Agent(state_size=37, action_size=4, seed=0)
```

1.3.1 Iteration and training the agent

```
In [12]: def dqn(n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):
             """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]            # get the current state
                 score = 0
                 for t in range(max_t):
                     action = agent.act(state, eps).astype(int)
                     env_info = env.step(action)[brain_name]        # send the action to the envir
                     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 finishe

 #                 next_state, reward, done, _ = env.step(action)
                     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)>=13.0:
                     print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.forma
                     torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
                     break
             return scores

In [13]: scores = dqn()

Episode 100        Average Score: 0.63
```

```
Episode 200	Average Score: 4.46
Episode 300	Average Score: 7.36
Episode 400	Average Score: 9.95
Episode 500	Average Score: 12.41
Episode 513	Average Score: 13.10
Environment solved in 413 episodes!	Average Score: 13.10
```

```
In [14]: # 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()
```




```
In [15]: agent.qnetwork_local.load_state_dict(torch.load('checkpoint.pth'))

In [16]: scores = []                        # list containing scores from each episode
         scores_window = deque(maxlen=100)  # last 100 scores
         eps = 0.01                         # initialize epsilon

         for i_episode in range(1):
             env_info = env.reset(train_mode=False)[brain_name] # reset the environment
             state = env_info.vector_observations[0]            # get the current state
```

```
        score = 0
        for t in range(200):
            action = agent.act(state, eps).astype(int)
            env_info = env.step(action)[brain_name]        # send the action to the environme
            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
            state = next_state
            score += reward
            if done:
                break
    env.close()
```

1.3.2 Improvements

We can improve following for better results of our algorithm : #### Tuning the Hyperparamers

BATCH_SIZE = 64 - batch size can help us get better learning of our neural net.

GAMMA = 0.99 - we can make future events less relevant and focus on immediate results which might be good when bananas are grouped.

LR = 5e-4 - we can try some different learning rate to get the low loss

UPDATE_EVERY = 4 - we can try different rate improve generalization of results.

Trying different Algorithms upgrade

Double DQN - fight with overestimation of action values.

Prioritized Experience Replay - make impornant experience more relevant.

Dueling DQN - imorove performance by dividing a neural network in state values and advantage values.

1.3.3 Conclusion

The most important part of reinforcement learning and also deep learning is tuning hyperparameters, getting the right set of magic parameters gives us the desired results.