Report

August 9, 2019

1 Navigation

In this notebook, you will learn how to use the Unity ML-Agents environment for the first project of the Deep Reinforcement Learning Nanodegree.

1.0.1 1. Start the Environment

We begin by importing some necessary packages. If the code cell below returns an error, please revisit the project instructions to double-check that you have installed Unity ML-Agents and NumPy.

[1]:
```
from unityagents import UnityEnvironment
import numpy as np
```

Next, we will start the environment! ***Before running the code cell below***, change the `file_name` parameter to match the location of the Unity environment that you downloaded.

- **Mac**: `"path/to/Banana.app"`
- **Windows** (x86): `"path/to/Banana_Windows_x86/Banana.exe"`
- **Windows** (x86_64): `"path/to/Banana_Windows_x86_64/Banana.exe"`
- **Linux** (x86): `"path/to/Banana_Linux/Banana.x86"`
- **Linux** (x86_64): `"path/to/Banana_Linux/Banana.x86_64"`
- **Linux** (x86, headless): `"path/to/Banana_Linux_NoVis/Banana.x86"`
- **Linux** (x86_64, headless): `"path/to/Banana_Linux_NoVis/Banana.x86_64"`

For instance, if you are using a Mac, then you downloaded `Banana.app`. If this file is in the same folder as the notebook, then the line below should appear as follows:

```
env = UnityEnvironment(file_name="Banana.app")
```

[2]:
```
env = UnityEnvironment(file_name="./Banana_Windows_x86_64/Banana.exe")
```

```
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.

[3]:
```
# 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

The simulation contains a single agent that navigates a large environment. At each time step, it has four actions at its disposal: - 0 - walk forward - 1 - walk backward - 2 - turn left - 3 - turn right

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around agent's forward direction. A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana.

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

[4]:
```
# 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.

Once this cell is executed, you will watch the agent's performance, if it selects an action (uniformly) at random with each time step. A window should pop up that allows you to observe the agent, as it moves through the environment.

Of course, as part of the project, you'll have to change the code so that the agent is able to use its experience to gradually choose better actions when interacting with the environment!

[5]:
```
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 = 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 step
    if done:                                       # exit loop if episode␣
 ↪finished
        break

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

```
Score: 0.0
```

When finished, you can close the environment.

[6]:
```
env.close()
```

1.0.4 4. It's Your Turn!

Now it's your turn to train your own agent to solve the environment! 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]
```

1.0.5 4.1 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.

[1]:
```
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
```

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

1.0.6 4.2 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.

[3]:
```
env = UnityEnvironment(file_name="./Banana_Windows_x86_64/Banana.exe")
```

```
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: , , ,
```

1.0.7 4.3 Default parameters of environment

Default parameters are provided. We just need to correctly reference those parameters.

[4]:
```
# get the default brain
brain_name = env.brain_names[0]
brain = env.brains[brain_name]
```

1.0.8 4.4 Tuning parameters of our neural network

The main components for a deep learning engineer. At this place we tune hyperparameters of our DQN reinforcement learning algorithms. I want to mention that `device` set to `cpu` because it is very expensive to move data from CPU to GPU and my GPU is not supported as we are using PyTorch 0.4.0 which is compiled for CUDA 8.0. My video card is supported starting from CUDA 9.0.

[5]:
```
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"
```

1.0.9 4.5 First heart of deep learning implementation

A neural network which consists of 4 Linear layers responsible for an agent behaviour. I tuned a littl a neural network to receive a better performance. Also I want to mention that slight change in a neural network might lead to degradation of results and finding correct parameters is a really hard task in reinforcement learning.

[6]:
```
class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=64,␣
 ↪fc2_units=128, fc3_units=64):
        """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
```

1.0.10 4.6 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.

[7]:
```
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", "reward", "next_state", "done"])
        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 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)

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

1.0.11 4.7 Agent - second heart of DQN implementation

The most important components: 1. `__init__` - initialized two identical neural networks. 2. `step` - adds data to replay buffer. And if there is enough samples then it calls `learn` method. 3. `act` - get `actiion_values` using state from `self.qnetwork_local` and checks eps which is responsible for exploration and exploitation ratio. if random value is greater then eps then selects maximum value from `action_values`(exploitation) otherwise make a randow draw from action space(exploration). 4. `learn` - `Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)` get maximum from output of a neural network. `Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))` - the main algorithms of DQN which calculates Q targets of DQN. Gamma is a discount factor to make rewards that are in future less relative than that of current rewards. Then we calculate `Q_expected = self.qnetwork_local(states).gather(1, actions)` which gives us output from `self.qnetwork_local`. `gather` is a multiindex selection method. Then we make optimization which makes `Q_expected` and `Q_targets` closer to each other. Method `learn` then calls `soft_update` method of a class `Agent`. 5. `soft_update` - it copies parameters from `q_network_local` and `q_network_target` according to tau coefficient to ‘`q_network_target`.

[8]:
```
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(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)

    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.parameters()):
            target_param.data.copy_(tau*local_param.data + (1.
↪0-tau)*target_param.data)
```

1.0.12 4.8 Instantiate class Agent

We instanstiate class `Agent` with state size equal to 37 and action space equal to 4. For reproducibility we set seed.

[9]:
```
agent = Agent(state_size=37, action_size=4, seed=0)
```

1.0.13 4.9 Iteration

We iterate through our environment. The main components are: `action = agent.act(state, eps).astype(int)` - choose action from DQN. `agent.step(state, action, reward, next_state, done)` - updating DQN.

[10]:
```
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␣
↪selection
        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 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

#             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_window)), end="")
        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.
↪mean(scores_window)))
        if np.mean(scores_window)>=13.0:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.
↪2f}'.format(i_episode-100, np.mean(scores_window)))
            torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
            break
    return scores
```

1.0.14 5. Start training

We call dqn function to start training.

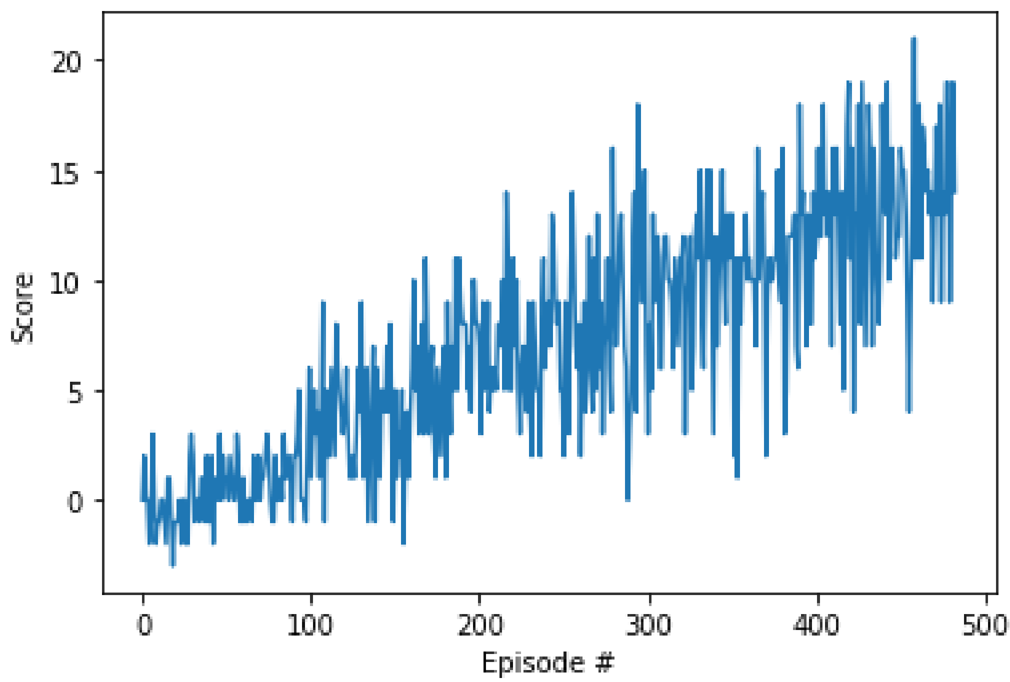
[11]:
```
scores = dqn()
```

```
Episode 100     Average Score: 0.43
Episode 200     Average Score: 4.61
Episode 300     Average Score: 7.61
Episode 400     Average Score: 10.04
Episode 483     Average Score: 13.04
Environment solved in 383 episodes!     Average Score: 13.04
```

1.0.15 6. Plotting loss

We plot a loss to visualize our training.

[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()
```



1.0.16 7. Load gradients

We load gradients from file `checkpoint.pth` that we saved during training.

[17]:
```
agent.qnetwork_local.load_state_dict(torch.load('checkpoint.pth'))
```

1.0.17 8. Final evaluation

The video is pretty fast in training mode that is the reason behind putting this code to have a nice view what is happening in an enivironment.

[22]:
```
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␣
 ↪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
        state = next_state
        score += reward
        if done:
            break
env.close()
```

1.0.18 9. Improvements

We can improve following:

9.1 Hyperparamers `BUFFER_SIZE = int(1e5)` - different buffer size parameters might be more suited for this particular task.

`BATCH_SIZE = 64` - batch size controls how often our neural network updates gradients.

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

`TAU = 1e-3` - let's call it copy ratio. Tuning this hyperparameter might lead to significant improvements.

`LR = 5e-4` - there are magic numbers for a neural network tasks but in reinfocement learning picture is a little bit different and might deviate from parameters of classic supervised deel learning tasks.

`UPDATE_EVERY = 4` - in some cases higher number might be better to improve generalization of results.

`device = "cpu"` - frequent moves from GPU and CPU make this parameter irrelevant for this particular tasks and there not much performance gains using GPU.

9.2 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`.

1.0.19 10. Conclusion

The most important part of reinforcement learning is tuning hyperparameters: slight change in parameters might lead to absolutely different result. And each change in parameter gives new bucket of optimization opportunties for deep learning engineer.

[ ]:

```

```