Report_plus

January 11, 2022

1 Navigation (with PER)

1.1 Brief introduction

In this notebook, I present my attempt to improve the solution to Project 1 in Deep Reinforcement Learning Nanodegree with Prioritised Experience Replay.

- *model.py* defines the Network, it is unchanged from initial attempt.
- *dqn_agent_plus.py* defines the Agent. I have tried to added the feature of prioritised experience replay.

The structure of this notebook is the same as *Navitation.ipynb*. However, it fails to complete the training due to speed being unbearably slow (took 6 hours to run 100 episode). Below I describe my intensions and suspections to the problem for potential further investigation.

1.1.1 Prioritised Experience Replay (PER)

In the file *dqn_agent_plus.py* the following modifications are done

- The relevant PER parameters are added, i.e. alpha, beta and epsilon
- class *ReplayBuffer* has been modified to store the the **priority** (based on the TD error) and **weight** of each set of (state, action, reward, next_action). Each set define an "experience" and is subscripted with i in the equations below.

TD error:

$$\delta_i^{TD} = reward_i + \gamma * argmaxQ(next_s tate) - Q(state, action)$$

priority:

$$p = |\delta_i^{TD}| + \epsilon$$

sampling probability:

$$P_i = \frac{p_i^\alpha}{\sum_i p_i^\alpha}$$

weight:

$$w_i = (\frac{1}{N}\frac{1}{P_i})^\beta$$

- There are new functions *TD_Error()* and *Priority_Weight()* defined for the relevant calculation.
- The experience samplying is updated to draw samples according to the "probability" calculated based on the TD error for each set of (state, action, reward, next_action).

- Each time the local Q-Network is updated, all the priorities and weights are updated in memory.

Problems The above is the intension of the modification. The training speed is greatly reduced after such modifications; and it get slower and slower as more episodes are completed, therefore I am not able to complete the run.

Below are some suspections: - the _replace() method from the namedtuple class is too slow? - looping through all the memory to update priorities and weights are too slow? - updating the local Q-Network with the following code is not efficient?

```
        for param, weight in zip(self.qnetwork_local.parameters(), weights):
            param._grad.data *= weight
```

1.2 Setting it all up

We begin by importing some necessary packages.

[1]:
```
from unityagents import UnityEnvironment
import numpy as np
import torch
from collections import deque
import matplotlib.pyplot as plt
%matplotlib inline
```

Loading the Banana Navigation enviroment by pointing the path to the EXE file in Windows.

[2]:
```
env = UnityEnvironment(file_name=r"./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: , , ,
```

Setting the Environment ***brain***.

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

```
brain = env.brains[brain_name]
```

checking the enviroment

[4]:
```
# reset the environment
env_info = env.reset(train_mode=False)[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
```

Trying with an un-trained agent - it does not take any action at all...

[5]:
```
from dqn_agent import Agent

agent = Agent(state_size=37, action_size=4, seed=0)
env_info = env.reset(train_mode=False)[brain_name]
state = env_info.vector_observations[0]
score = 0
i = 0

while True:
    action = agent.act(state).item()
    env_info = env.step(action)[brain_name]
    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
```

1.3 Training (not able to complete due to speed!)

[5]:
```
from dqn_agent_plus import Agent

agent = Agent(state_size=37, action_size=4, seed=0)

def dqn(brain_name, 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]
        state = env_info.vector_observations[0]
        score = 0
        for t in range(max_t):
            action = agent.act(state, eps).item()
            env_info = env.step(action)[brain_name]
            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]
            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)>=16.0:
            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_plus.pth')
            break
    return scores

scores = dqn(brain_name)
```

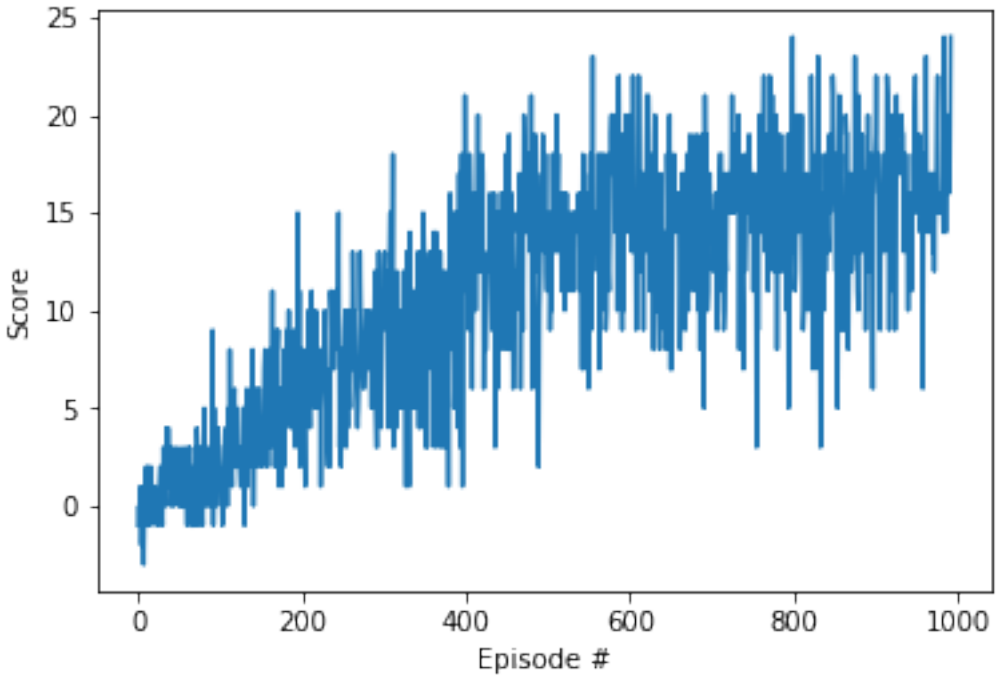
```
Episode 100     Average Score: 0.36
Episode 101     Average Score: 0.35
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-5-b3c6e5ea62da> in <module>
     44     return scores
     45
---> 46 scores = dqn(brain_name)

<ipython-input-5-b3c6e5ea62da> in dqn(brain_name, n_episodes, max_t, eps_start,
↪eps_end, eps_decay)
     27             reward = env_info.rewards[0]                   # get the␣
↪reward
     28             done = env_info.local_done[0]
---> 29             agent.step(state, action, reward, next_state, done)
     30             state = next_state
     31             score += reward

~\Documents\GitHub\DRLND_projects\p1_navigation\dqn_agent_plus.py in step(self,
↪state, action, reward, next_state, done)
     63                 experiences = self.memory.sample()
     64                 self.learn(experiences, GAMMA)
---> 65                 self.memory.update_priority(self.qnetwork_local)
     66
     67     def act(self, state, eps=0.):
```

```
~\Documents\GitHub\DRLND_projects\p1_navigation\dqn_agent_plus.py in␣
↪update_priority(self, Q)
    182     def update_priority(self, Q):
    183         for m in self.memory:
--> 184             m = m._replace(priority = self.epsilon + np.abs(TD_Error(Q,␣
↪m.state, m.action, m.reward, m.next_state, GAMMA)))
    185             m = m._replace(weight = Priority_Weight(m.priority, self.␣
↪batch_size, self.beta))
    186

~\Documents\GitHub\DRLND_projects\p1_navigation\dqn_agent_plus.py in TD_Error(Q␣
↪state, action, reward, next_state, gamma)
    188     state = torch.from_numpy(state).float().unsqueeze(0).to(device)
    189     next_state = torch.from_numpy(next_state).float().unsqueeze(0).␣
↪to(device)
--> 190     return reward + gamma * Q(next_state).detach().max(1)[0].␣
↪unsqueeze(1) - Q(state).detach().numpy()[0,action]
    191
    192 def Priority_Weight(p, N, beta):

~\anaconda3\envs\drlnd\lib\site-packages\torch\nn\modules\module.py in␣
↪_call_impl(self, *input, **kwargs)
   1100         if not (self._backward_hooks or self._forward_hooks or self.␣
↪_forward_pre_hooks or _global_backward_hooks
   1101                 or _global_forward_hooks or _global_forward_pre_hooks):
-> 1102             return forward_call(*input, **kwargs)
   1103         # Do not call functions when jit is used
   1104         full_backward_hooks, non_full_backward_hooks = [], []

~\Documents\GitHub\DRLND_projects\p1_navigation\model.py in forward(self, state
     24     def forward(self, state):
     25         """Build a network that maps state -> action values."""
---> 26         x = F.relu(self.fc1(state))
     27         x = F.relu(self.fc2(x))
     28         return self.fc3(x)

~\anaconda3\envs\drlnd\lib\site-packages\torch\nn\modules\module.py in␣
↪_call_impl(self, *input, **kwargs)
   1100         if not (self._backward_hooks or self._forward_hooks or self.␣
↪_forward_pre_hooks or _global_backward_hooks
   1101                 or _global_forward_hooks or _global_forward_pre_hooks):
-> 1102             return forward_call(*input, **kwargs)
   1103         # Do not call functions when jit is used
   1104         full_backward_hooks, non_full_backward_hooks = [], []

~\anaconda3\envs\drlnd\lib\site-packages\torch\nn\modules\linear.py in␣
↪forward(self, input)
    101
```

```
    102     def forward(self, input: Tensor) -> Tensor:
--> 103         return F.linear(input, self.weight, self.bias)
    104
    105     def extra_repr(self) -> str:

~\anaconda3\envs\drlnd\lib\site-packages\torch\nn\functional.py in linear(input
 ↪weight, bias)
   1846     if has_torch_function_variadic(input, weight, bias):
   1847         return handle_torch_function(linear, (input, weight, bias),␣
 ↪input, weight, bias=bias)
-> 1848     return torch._C._nn.linear(input, weight, bias)
   1849
   1850

KeyboardInterrupt:
```

[7]:
```
# 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()
```



1.4 See trained Agent in action

[12]:
```
agent.qnetwork_local.load_state_dict(torch.load('checkpoint_plus.pth'))
env_info = env.reset(train_mode=False)[brain_name]
state = env_info.vector_observations[0]
score = 0

while True:
    action = agent.act(state).item()
    env_info = env.step(action)[brain_name]
    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: 21.0
```

close the environment.

[13]:
```
env.close()
```