3.dueling dqn

December 5, 2019

[1]:
```
import math, random

import gym
import numpy as np

import torch
import torch.nn as nn
import torch.optim as optim
import torch.autograd as autograd
import torch.nn.functional as F
```

[2]:
```
from IPython.display import clear_output
import matplotlib.pyplot as plt
%matplotlib inline
```

Use Cuda

[3]:
```
USE_CUDA = torch.cuda.is_available()
Variable = lambda *args, **kwargs: autograd.Variable(*args, **kwargs).cuda() if␣
 ↪USE_CUDA else autograd.Variable(*args, **kwargs)
```

Replay Buffer

[4]:
```
from collections import deque

class ReplayBuffer(object):
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        state      = np.expand_dims(state, 0)
        next_state = np.expand_dims(next_state, 0)

        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        state, action, reward, next_state, done = zip(*random.sample(self.
 ↪buffer, batch_size))
        return np.concatenate(state), action, reward, np.
 ↪concatenate(next_state), done
```

```
    def __len__(self):
        return len(self.buffer)
```

Cart Pole Environment

[5]:

```
env_id = "CartPole-v1"
env = gym.make(env_id)
```

Epsilon greedy exploration

[6]:

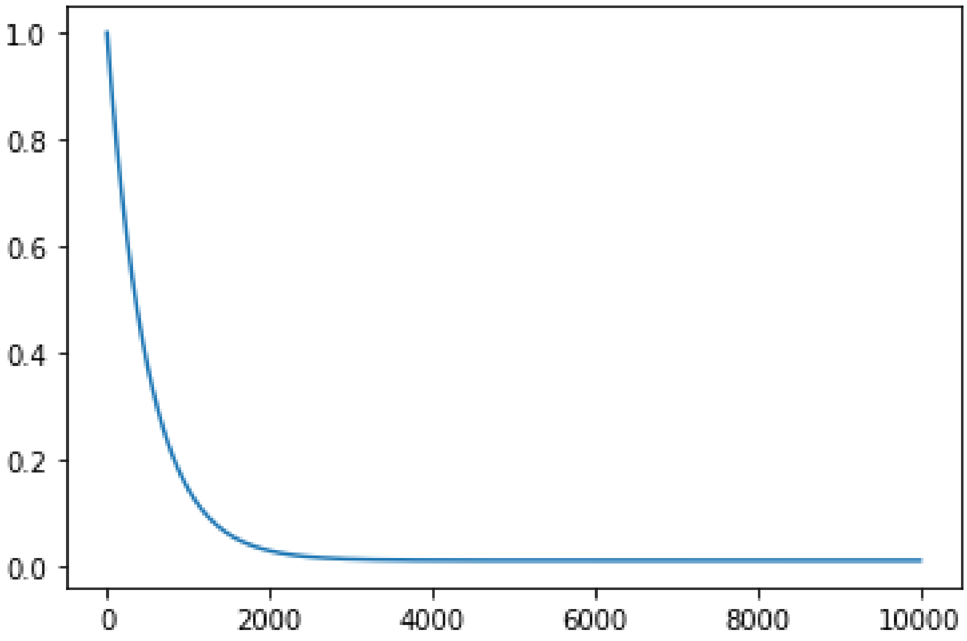
```
epsilon_start = 1.0
epsilon_final = 0.01
epsilon_decay = 500

epsilon_by_frame = lambda frame_idx: epsilon_final + (epsilon_start -␣
 ↪epsilon_final) * math.exp(-1. * frame_idx / epsilon_decay)
```

[7]:

```
plt.plot([epsilon_by_frame(i) for i in range(10000)])
```

[7]:

```
[<matplotlib.lines.Line2D at 0x7f92940aed68>]
```



Dueling Deep Q Network

[8]:

```
class DuelingDQN(nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(DuelingDQN, self).__init__()


        self.feature = nn.Sequential(
            nn.Linear(num_inputs, 128),
            nn.ReLU()
        )

        self.advantage = nn.Sequential(
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, num_outputs)
        )

        self.value = nn.Sequential(
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )

    def forward(self, x):
        x = self.feature(x)
        advantage = self.advantage(x)
        value     = self.value(x)
        return value + advantage  - advantage.mean()

    def act(self, state, epsilon):
        if random.random() > epsilon:
#             state   = Variable(torch.FloatTensor(state).unsqueeze(0),␣
 ↪volatile=True)
            with torch.no_grad():
                state = Variable(torch.FloatTensor(state).unsqueeze(0))
            q_value = self.forward(state)
#             action  = q_value.max(1)[1].data[0]
            action = q_value.max(1)[1].item()
        else:
            action = random.randrange(env.action_space.n)
        return action
```

[9]:

```
current_model = DuelingDQN(env.observation_space.shape[0], env.action_space.n)
target_model  = DuelingDQN(env.observation_space.shape[0], env.action_space.n)

if USE_CUDA:
    current_model = current_model.cuda()
    target_model  = target_model.cuda()

optimizer = optim.Adam(current_model.parameters())

replay_buffer = ReplayBuffer(1000)
```

Synchronize current policy net and target net

[10]:

```
def update_target(current_model, target_model):
    target_model.load_state_dict(current_model.state_dict())
```

[11]:

```
update_target(current_model, target_model)
```

Computing Temporal Difference Loss

[12]:

```
def compute_td_loss(batch_size):
    state, action, reward, next_state, done = replay_buffer.sample(batch_size)

    state      = Variable(torch.FloatTensor(np.float32(state)))
    next_state = Variable(torch.FloatTensor(np.float32(next_state)))
    action     = Variable(torch.LongTensor(action))
    reward     = Variable(torch.FloatTensor(reward))
    done       = Variable(torch.FloatTensor(done))

    q_values      = current_model(state)
    next_q_values = target_model(next_state)

    q_value          = q_values.gather(1, action.unsqueeze(1)).squeeze(1)
    next_q_value     = next_q_values.max(1)[0]
    expected_q_value = reward + gamma * next_q_value * (1 - done)

    loss = (q_value - expected_q_value.detach()).pow(2).mean()

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

#     return loss
    return loss
```

[13]:

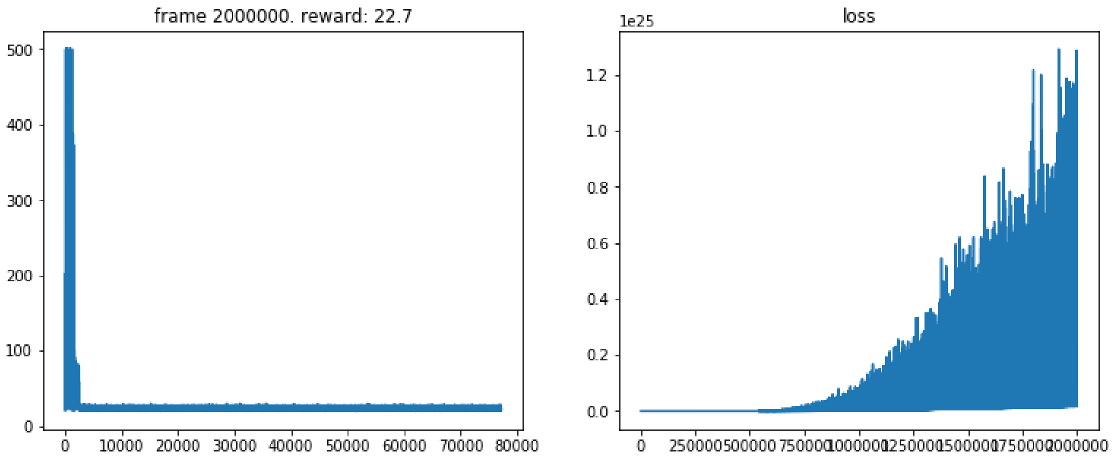
```
def plot(frame_idx, rewards, losses):
    clear_output(True)
    plt.figure(figsize=(20,5))
    plt.subplot(131)
    plt.title('frame %s. reward: %s' % (frame_idx, np.mean(rewards[-10:])))
    plt.plot(rewards)
    plt.subplot(132)
    plt.title('loss')
    plt.plot(losses)
    plt.show()
```

Training

[14]:

```
# num_frames = 10000
num_frames = 2000000
batch_size = 32
gamma      = 0.99

losses = []
all_rewards = []
episode_reward = 0
```

```
state = env.reset()
for frame_idx in range(1, num_frames + 1):
    epsilon = epsilon_by_frame(frame_idx)
    action = current_model.act(state, epsilon)

    next_state, reward, done, _ = env.step(action)
    replay_buffer.push(state, action, reward, next_state, done)

    state = next_state
    episode_reward += reward

    if done:
        state = env.reset()
        all_rewards.append(episode_reward)
        episode_reward = 0

    if len(replay_buffer) > batch_size:
        loss = compute_td_loss(batch_size)
#         losses.append(loss.data[0])
        losses.append(loss.item())

    if frame_idx % 200 == 0:
        plot(frame_idx, all_rewards, losses)

    if frame_idx % 100 == 0:
        update_target(current_model, target_model)
```



Atari Environment

[15]:

```
# from common.wrappers import make_atari, wrap_deepmind, wrap_pytorch
from wrappers import make_atari, wrap_deepmind, wrap_pytorch
```

[16]:

```
# env_id = "PongNoFrameskip-v4"
env_id = "BreakoutNoFrameskip-v4"
env    = make_atari(env_id)
env    = wrap_deepmind(env)
env    = wrap_pytorch(env)
```

[17]:

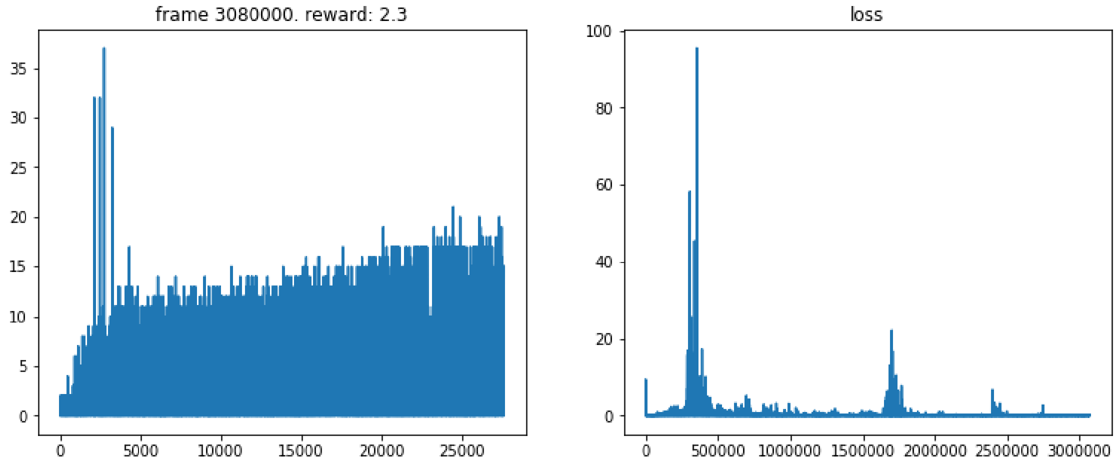
```
class DuelingCnnDQN(nn.Module):
    def __init__(self, input_shape, num_outputs):
        super(DuelingCnnDQN, self).__init__()


        self.input_shape = input_shape
        self.num_actions = num_outputs

        self.features = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )

        self.advantage = nn.Sequential(
            nn.Linear(self.feature_size(), 512),
            nn.ReLU(),
            nn.Linear(512, num_outputs)
        )

        self.value = nn.Sequential(
            nn.Linear(self.feature_size(), 512),
            nn.ReLU(),
            nn.Linear(512, 1)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        advantage = self.advantage(x)
        value     = self.value(x)
        return value + advantage  - advantage.mean()

    def feature_size(self):
        return self.features(autograd.Variable(torch.zeros(1, *self.
 ↪input_shape))).view(1, -1).size(1)

    def act(self, state, epsilon):
        if random.random() > epsilon:
#             state   = Variable(torch.FloatTensor(np.float32(state)).
 ↪unsqueeze(0), volatile=True)
            with torch.no_grad():
                state = Variable(torch.FloatTensor(np.float32(state)).
 ↪unsqueeze(0))
            q_value = self.forward(state)
#             action  = q_value.max(1)[1].data[0]
            action  = q_value.max(1)[1].item()
        else:
            action = random.randrange(env.action_space.n)
        return action
```

[18]:

```
current_model = DuelingCnnDQN(env.observation_space.shape, env.action_space.n)
target_model  = DuelingCnnDQN(env.observation_space.shape, env.action_space.n)

if USE_CUDA:
    current_model = current_model.cuda()
    target_model  = target_model.cuda()

optimizer = optim.Adam(current_model.parameters(), lr=0.0001)

replay_initial = 10000
replay_buffer = ReplayBuffer(300000)
# replay_initial = 500
# replay_buffer = ReplayBuffer(1000)

update_target(current_model, target_model)
```

[ ]:

```
num_frames = 5000000
# num_frames = 10000
batch_size = 32
gamma      = 0.99

losses = []
all_rewards = []
episode_reward = 0

state = env.reset()
for frame_idx in range(1, num_frames + 1):
    epsilon = epsilon_by_frame(frame_idx)
    action = current_model.act(state, epsilon)

    next_state, reward, done, _ = env.step(action)
    replay_buffer.push(state, action, reward, next_state, done)

    state = next_state
    episode_reward += reward
    if done:
        state = env.reset()
        all_rewards.append(episode_reward)
        episode_reward = 0

    if len(replay_buffer) > replay_initial:
        loss = compute_td_loss(batch_size)
        losses.append(loss.item())

    if frame_idx % 10000 == 0:
        plot(frame_idx, all_rewards, losses)

    if frame_idx % 1000 == 0:
        update_target(current_model, target_model)
```



[ ]: