DQN 실습 with Atari BreakOut

Author : nemo
2020-08-31

Contents

이번 글에서는 Atari-BreakoutDeterministic-v4을 이용해 DQN을 실습해보고자 한다.

---

1. Environment

```
state, reward_step, done, info = env.step(action)
```

info["ale.lives"]에는 수명이 5개 들어있다. 한 수명이 깎일 때마다 게임을 다시 시작하는 행동(1)을 취해줘야 한다...만 구현에서 쓰이진 않는다.

done은 에피소드 종료, 즉 info["ale.lives"] == 0인 상황에서 True가 반환된다.

1.1. State Space

상태 공간은 210x160x3의 BreakOut 게임 화면이다. 3은 RGB채널을 의미한다.

1.2. Action Space

행동 공간은 4개로 0은 가만히 있기, 1은 게임 시작(라이프가 깎일 경우), 2와 3은 각각 왼쪽 또는 오른쪽으로 움직이는 행동이다.

2. Implement

2.1. Setting

적당히 필요한 모듈들을 import 하자.

```
import numpy as np
import gym
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
from PIL import Image

from copy import deepcopy as c
from tqdm import tqdm
import IPython.display as display
import os
```

우성님이 알려준 wandb도 써보자. wandb를 쓰려면 터미널에서 로그인해둬야 한다.

```
import wandb
wandb.init(project="dqn-atari-breakout", name="0831-Fly-epsilon_step=1e-7")
```

기다리는 시간을 단축해줄 소중한 GPU ~ 사실 DQN은 병렬 연산이 그리 많이 필요하진 않다.

```
USE_CUDA = torch.cuda.is_available()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

환경을 불러오고, Resize할 이미지 크기에 대한 정보도 적어두자.

```
env = gym.make("BreakoutDeterministic-v4")
state = env.reset()
Height = 84
Width = 84
```

2.2. Image Preprocessing

BreakOut의 상태는 210x160x3의 크기로 상당히 큰 편이다.

주변의 벽을 잘라주고, 84x84 크기로 조절하자.

RGB라는 색깔에 대한 정보는 불필요하므로 흑백으로 변환하자.

```
def preprocessing(image):

    image = image[30:-17, 7:-7, :]
    image = Image.fromarray(image)
    image = image.resize((84, 84))
    gray_filter = np.array([0.299, 0.587, 0.114])
    image = np.einsum("...i,i->...", image, gray_filter)
    image = image * 2 / 255 - 1

    return image
```

여담으로 numpy, torch 등에 모두 구현되어 있는 einsum이라는 연산은 여러모로 편하다. 더 알아보고 싶으면 공식 문서를 읽어보자.

2.3. CNN

Q를 근사하기 위한 적당한 신경망을 짜자.

Flatten 이후 선형 레이어에 들어갈 인풋을 계산하기 귀찮으니, 대신 계산해주는 함수를 짜자.

나머지 등의 부분은 프레임워크의 CNN 구현에 따라 다를 수 있으니, 직접 계산하지 말고 공식 문서에서 제시하는 공식을 참고해서 짜자.

```
def calculate_Conv2d_dimension(input_size, kernel_size, stride, padding):
    return ((input_size - kernel_size + 2 * padding) // stride) + 1


def calculate_MaxPool2d_dimension(input_size, max_pool):
    return input_size // max_pool
```

논문대로 CNN을 구현하자.

```
class CNN(nn.Module):

    def __init__(self, input_size, output_size):  # input size는 3차원

        super(CNN, self).__init__()

        calculate_H, calculate_W, channel = input_size

        calculate_H, calculate_W = [calculate_Conv2d_dimension(x, 8, 4, 0) for x in (calculate_H, calculate_W)]
        calculate_H, calculate_W = [calculate_Conv2d_dimension(x, 4, 2, 0) for x in (calculate_H, calculate_W)]
        calculate_H, calculate_W = [calculate_Conv2d_dimension(x, 3, 1, 0) for x in (calculate_H, calculate_W)]

        self.layers = nn.Sequential(
            nn.Conv2d(in_channels=channel, out_channels=32, kernel_size=8, stride=4, padding=0),
            nn.ReLU(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2, padding=0),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=0),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(in_features=calculate_H * calculate_W * 64, out_features=512),
            nn.ReLU(),
            nn.Linear(in_features=512, out_features=output_size)
        )

    def forward(self, x):
        if len(x.shape) == 3:
            x = x.unsqueeze(0)

        return self.layers(x)
```

2.4. History

공의 움직임에 대한 정보를 포함하기 위해, 4개 프레임을 하나의 상태로 취급하고자 한다.

크기 5의 배열을 만들어, 앞의 4칸은 현재 상태를, 뒤의 4칸은 다음 상태를 표현하게 하자.

```
class HISTORY:

    def __init__(self, H, W):
        self.history = np.zeros((5, H, W))

    def start(self, x):
        for i in range(5):
            self.history[i] = c(x)

    def update(self, x):
        self.history[0:4, :, :] = c(self.history[1:5, :, :])
        self.history[4] = c(x)
```

2.5. DATA 자료형

하나의 경험(Experience)을 나타내는 DATA 자료형을 만들자.

```
class DATA():

    def __init__(self, state, action, reward, done):
        self.state = c(state)  # np array: 5 by 84 by 84
        self.action = c(action)
        self.reward = c(reward)
        self.done = c(done)
```

2.6. REPLAY_MEMORY

Experience Replay에 관한 정보를 관리하는 Class인 REPLAY_MEMORY를 짜자.

```
class REPLAY_MEMORY():

    def __init__(self, capacity):

        self.replay = []
        self.capacity = c(capacity)
        self.time = 0

    def update(self, x):

        if len(self.replay) < self.capacity:
            self.replay.append(c(x))

        else:
            pass

        self.replay[self.time] = c(x)
        self.time = (self.time + 1) % self.capacity

```

```
    def sample(self, sample_size):

        assert sample_size <= len(self.replay), "Error !! sample_size > length or capacity"

        sample_data = np.random.choice(self.replay, size=sample_size, replace=False)

        states = np.zeros((sample_size, 4, 84, 84))
        actions = np.zeros((sample_size), dtype=np.int64)
        rewards_step = np.zeros((sample_size))
        states_next = np.zeros((sample_size, 4, 84, 84))
        dones = np.zeros((sample_size))

        for i in range(sample_size):
            states[i] = sample_data[i].state[:4]
            actions[i] = sample_data[i].action
            rewards_step[i] = sample_data[i].reward
            states_next[i] = sample_data[i].state[1:]
            dones[i] = sample_data[i].done

        return states, actions, rewards_step, states_next, dones
```

2.7. Train

DQN 구조에 맞게 열심히 구현하자 !!

```
def train(env, episodes, learning_rate=0.0001, epsilon=1.0, gamma=0.99, min_epsilon=0.10, epsilon_step=1e-7,
          reset=False, replay_capacity = 10000):
    main_cnn = CNN((Height, Width, 4), 4).to(device)  # Q initialization
    target_cnn = CNN((Height, Width, 4), 4).to(device)
    target_cnn.load_state_dict(main_cnn.state_dict())
    target_cnn.eval()

    criterion = nn.MSELoss().to(device)
    optimizer = torch.optim.RMSprop(main_cnn.parameters(), lr=learning_rate)

    if reset == False:
        try:
            main_cnn.load_state_dict(torch.load("target_cnn_one2.pkl"))
            target_cnn.load_state_dict(main_cnn.state_dict())
            optimizer.load_state_dict(torch.load("optimizer_one2.pkl"))
        except:
            pass

    wandb.watch(main_cnn)

    step = 0
    history = HISTORY(Height, Width)
    replay_memory = REPLAY_MEMORY(replay_capacity)

    reward_history = []
    count_action_history = []
```

```
27
28      for episode in tqdm(range(episodes)):
29
30          state = env.reset()
31          state = preprocessing(state)
32          history.start(state)
33          reward = 0
34
35          count_action = [0, 0, 0, 0]
36
37          while True:
38
39              state = c(history.history[1:])
40
41              # Choose Action
42              if np.random.random() < 1 - epsilon:
43                  action =
    target_cnn(torch.from_numpy(state).float().to(device)).to("cpu")
44                  action = torch.argmax(action).item()
45              else:
46                  action = np.random.randint(0, 4)
47
48              count_action[action] += 1
49              epsilon = max(min_epsilon, epsilon - epsilon_step)
50
51              # Step
52              step += 1
53              state_next, reward_step, done, info = env.step(action)
54              state_next = preprocessing(state_next)
55              history.update(state_next)
56
57              reward += reward_step
58              replay_memory.update(DATA(history.history, action,
    reward_step, done))
59
60              if step >= replay_capacity and step % 10 == 0:
61
62                  main_cnn.train()
63                  states, actions, rewards_step, states_next, dones =
    replay_memory.sample(32)
64
65                  states = torch.from_numpy(states).float().to(device)
66                  actions = torch.from_numpy(actions).to(device)
67                  rewards_step =
    torch.from_numpy(rewards_step).float().to(device)
68                  states_next =
    torch.from_numpy(states_next).float().to(device)
69                  dones = torch.from_numpy(dones).to(device)
70
71                  Q_main = torch.sum(main_cnn(states) * F.one_hot(actions,
    4), dim=-1) # main: for training
72
73                  with torch.no_grad():
74                      Q_target = rewards_step + gamma *
    torch.max(target_cnn(states_next), dim=-1)[0].detach()
75
76                  optimizer.zero_grad()
77
```

```
                loss = criterion(Q_main, Q_target)
                loss.backward()
                optimizer.step()
                wandb.log({"Loss": loss.to("cpu").item()})

            if step % 10000 == 0:
                target_cnn.load_state_dict(main_cnn.state_dict())

            if done:
                break

        if episode % 300 == 0:
            display.clear_output()
            print(step, episode)
            plt.title("reward_history, episode : {} epsilon : {}".format(episode, epsilon))
            plt.plot(reward_history)
            plt.show()
            plt.title("count_action_history, episode : {} epsilon : {}".format(episode, epsilon))
            plt.plot(count_action_history)
            plt.show()

        reward_history.append(reward)
        count_action_history.append(count_action)
        wandb.log({"Reward": reward, "count_action_0": count_action[0], "count_action_1": count_action[1],
                   "count_action_2": count_action[2], "count_action_3": count_action[3], "Step": episode,
                   "epsilon": epsilon, "step": step})

        if episode % 1000 == 0:
            torch.save(target_cnn.state_dict(), "cnn2.pkl")
            torch.save(optimizer.state_dict(), "optimizer2.pkl")
            torch.save(target_cnn.state_dict(), os.path.join(wandb.run.dir, 'model.pt'))
            torch.save(optimizer.state_dict(), os.path.join(wandb.run.dir, 'optimizer.pt'))

    return cnn, reward_history


cnn, reward_history = train(env, 300000, epsilon=1.0, reset=True, replay_capacity=10000)
```