Lab14

Pranisaa Charnparttaravanit st121720

Recall that Mnih et al. (2015) obtained superhuman playing ability for some of the Atari games by combining DQN with experience replay and a more sophisticated state representation that what we've seen so far: they stack successive frames as the input state representation so as to give the agent some "velocity" input rather than a static snapshot of the scene.

Try combining the techniques we've developed in the lab with the frame history as state, and get the best Space Invaders player you can. What's your agent's average and best score over 100 games?

In [5]:

```
import math, random
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.autograd as autograd
import torchvision.transforms as T

from PIL import Image
import matplotlib.pyplot as plt
import gym
import numpy as np
from tqdm import trange
from myDQN import DQN, ReplayBuffer, CNNDQN, DDQN, NaivePrioritizedBuffer
```

Below code is taken from the following link:

[https://github.com/akraradets (https://github.com/akraradets)](https://github.com/akraradets)

Select GPU or CPU as device

In [6]:

```
device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cpu
```

Plot function

In [7]:

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

Define epsilon as a function of time (episode index)

In [8]:

```
def gen_eps_by_episode(epsilon_start, epsilon_final, epsilon_decay):
    eps_by_episode = lambda episode: epsilon_final + (epsilon_start - epsilon_fi
nal) * math.exp(-1. * episode / epsilon_decay)
    return eps_by_episode

# episodes = 200000
episodes = 5000
batch_size = 64
gamma      = 0.99
min_play_reward = -.15

epsilon_start = 1.0
epsilon_final = 0.01
epsilon_decay = episodes / 10
eps_by_episode = gen_eps_by_episode(epsilon_start, epsilon_final, epsilon_decay)
```

Define a function to return an increasing beta over episodes

In [9]:

```
beta_start = 0.4
beta_episodes = episodes / 10
def gen_beta_by_episode(beta_start, beta_episodes):
    beta_by_episode = lambda episode: min(1.0, beta_start + episode * (1.0 - bet
a_start) / beta_episodes)
    return beta_by_episode

beta_by_episode = gen_beta_by_episode(beta_start, beta_episodes)
```

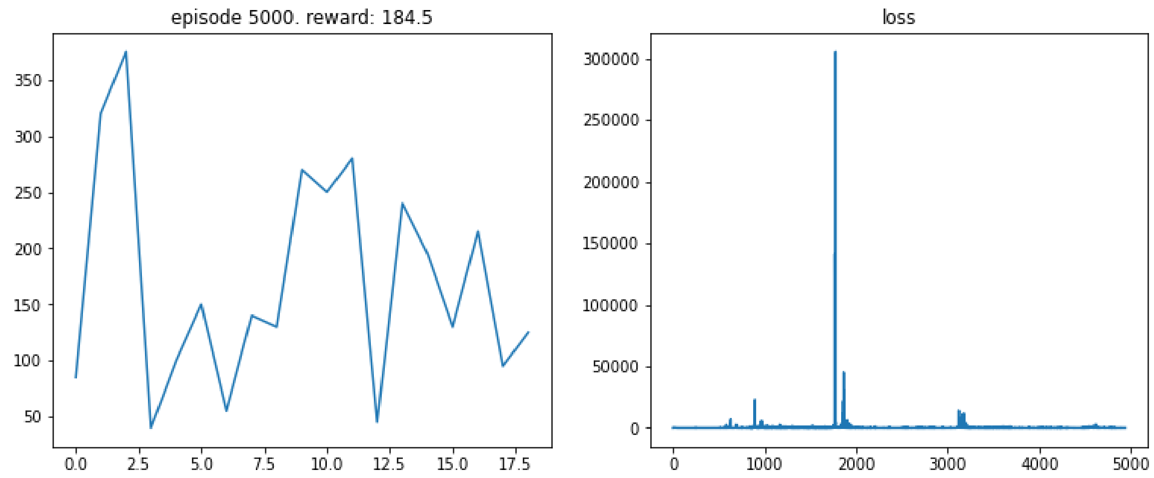
In [1]:

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

env_id = 'SpaceInvaders-v0'
env = gym.make(env_id)

current_model = DDQN(4, env.action_space.n).to(device)
target_model  = DDQN(4, env.action_space.n).to(device)

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

# Change from Normal replay buffer to be prioritize buffer
# replay_buffer = ReplayBuffer(100000)
replay_buffer = NaivePrioritizedBuffer(100000)

update_target(current_model, target_model)

image_size = 84
transform = T.Compose([T.ToPILImage(),
                       T.Grayscale(num_output_channels=1),
                       T.Resize((image_size, image_size), interpolation=Image.CU
BIC),
                       T.ToTensor()])

def get_state2(observation):
    state = observation.transpose((2,0,1))
    state = torch.from_numpy(state)
    state = transform(state)
    return state

def compute_td_loss_DDQN_prior_exp_replay(current_model, target_model, batch_siz
e, gamma=0.99, beta=0.4):
    # get data from replay mode
    # state, action, reward, next_state, done = replay_buffer.sample(batch_size)
    state, action, reward, next_state, done, indices, weights = replay_buffer.sa
mple(batch_size, beta)

    # convert to tensors
    # Autograd automatically supports Tensors with requires_grad set to True.
    state      = autograd.Variable(torch.FloatTensor(np.float32(state))).to(devi
ce)
    next_state = autograd.Variable(torch.FloatTensor(np.float32(next_state)), vo
latile=True).to(device)
    action     = autograd.Variable(torch.LongTensor(action)).to(device)
    reward     = autograd.Variable(torch.FloatTensor(reward)).to(device)
    done       = autograd.Variable(torch.FloatTensor(done)).to(device)
    weights    = autograd.Variable(torch.FloatTensor(weights)).to(device)

    # calculate q-values and next q-values from deeplearning
    q_values      = current_model(state)
    next_q_values = current_model(next_state)

    # double DQN add here
    #next_q_state_values = target_model(next_state)

    # get q-value from propagated action in each step
    q_value          = q_values.gather(1, action.unsqueeze(1)).squeeze(1)
    # double DQN different here
    #next_q_value     = next_q_state_values.gather(1, torch.max(next_q_values,
```

```
 1)[1].unsqueeze(1)).squeeze(1)
    next_q_value     = next_q_values.max(1)[0]
    # calculate expected q-value from q-function
    expected_q_value = reward + gamma * next_q_value * (1 - done)

    # calculate loss value
    # loss = (q_value - autograd.Variable(expected_q_value.data)).pow(2).mean()
    loss = (q_value - expected_q_value.detach()).pow(2).mean()
    prios = loss + 1e-5
    loss  = loss.mean()

    optimizer.zero_grad()
    loss.backward()
    replay_buffer.update_priorities(indices, prios.data.cpu().numpy())
    optimizer.step()

    return loss

def train_DDQN_prior_exp_replay(env, current_model, target_model, eps_by_episode
, optimizer, replay_buffer, beta_by_episode, episodes = 10000, batch_size=32, ga
mma = 0.99, min_play_reward=-.15):
    losses = []
    all_rewards = []
    episode_reward = 0

    obs = env.reset()
    state = get_state3(obs)
    tot_reward = 0
    tr = trange(episodes+1, desc='Agent training', leave=True)
    for episode in tr:
        avg_reward = tot_reward / (episode + 1)
        tr.set_description("Agent training (episode{}) Avg Reward {}".format(epi
sode+1,avg_reward))
        tr.refresh()

        # get action with q-values
        epsilon = eps_by_episode(episode)
        action = current_model.act(state, epsilon, env, device)

        # input action into state
        reward = 0
        for i in range(3):
            next_obs, i_reward, done, _ = env.step(action)
            reward += i_reward
            if(done): break
        next_state = get_state3(next_obs)
        # save data into buffer
        replay_buffer.push(state, action, reward, next_state, done)

        tot_reward += reward

        state = next_state
        obs = next_obs
        episode_reward += reward

        if done:
            obs = env.reset()
            state = get_state3(obs)
            all_rewards.append(episode_reward)
            episode_reward = 0
```

```
        if len(replay_buffer) > batch_size:
            beta = beta_by_episode(episode)
            loss = compute_td_loss_DDQN_prior_exp_replay(current_model, target_m
odel, batch_size, gamma, beta)
            losses.append(loss.item())

        if episode % 500 == 0:
            update_target(current_model, target_model)

    plot(episode, all_rewards, losses)
    return current_model, target_model, all_rewards, losses

current_model, target_model, all_rewards, losses = train_DDQN_prior_exp_replay(e
nv, current_model, target_model, eps_by_episode, optimizer, replay_buffer, beta_
by_episode, episodes = episodes, batch_size=batch_size, gamma = gamma, min_play_
reward = min_play_reward)
torch.save(current_model.state_dict(), 'checkpoints/spaceInvaders-hw-phi-skip-1
M.pth')
torch.save(target_model.state_dict(), 'checkpoints/spaceInvaders-target-hw-phi-s
kip-1M.pth')
```

```
cpu

/Users/apple/opt/anaconda3/lib/python3.8/site-packages/torchvision/t
ransforms/transforms.py:257: UserWarning: Argument interpolation sho
uld be of type InterpolationMode instead of int. Please, use Interpo
lationMode enum.
  warnings.warn(
Agent training (episode45) Avg Reward 0.0:   1%|           | 40/5001
[00:00<01:47, 45.96it/s]/Users/apple/Documents/AIT/Sem II/RTML/RTML_
Labs/Lab14/myDQN.py:46: UserWarning: volatile was removed and now ha
s no effect. Use `with torch.no_grad():` instead.
  state   = autograd.Variable(torch.FloatTensor(state).unsqueeze(0),
volatile=True).to(device)
Agent training (episode65) Avg Reward 0.7692307692307693:   1%||
| 63/5001 [00:01<01:49, 44.99it/s] <ipython-input-1-fb77817a7b14>:11
4: UserWarning: volatile was removed and now has no effect. Use `wit
h torch.no_grad():` instead.
  next_state = autograd.Variable(torch.FloatTensor(np.float32(next_s
tate)), volatile=True).to(device)
Agent training (episode5001) Avg Reward 0.650869826034793: 100%|████
██████| 5001/5001 [13:54<00:00,  5.99it/s]
```



In [ ]:

```
import math, random
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.autograd as autograd
import matplotlib.pyplot as plt
import gym
import numpy as np
from tqdm import trange
from myDQN import DQN, ReplayBuffer, CNNDQN,DDQN

import torchvision.transforms as T
from PIL import Image

image_size = 84
transform = T.Compose([T.ToPILImage(),
                       T.Grayscale(num_output_channels=1),
                       T.Resize((image_size, image_size), interpolation=Image.CU
BIC),
                       T.ToTensor()])
```

Convert to RGB image (3 channels)

In [ ]:

```
import queue
state_buffer = queue.Queue()
def get_state3(observation):

    # First time, repeat the state for 3 times
    if(state_buffer.qsize() == 0):
        for i in range(4):
            state = get_state2(observation)
            state_buffer.put(state)
        # print(observation.shape, state.shape)
    else:
        state_buffer.get()
        state = get_state2(observation)
        state_buffer.put(state)
    # for i in state_buffer.queue:
    #     print(i.shape)
    rep = torch.cat(list(state_buffer.queue), dim=0)
    # print("rep=====",rep.shape)
    return rep

def get_state2(observation):
    state = observation.transpose((2,0,1))
    state = torch.from_numpy(state)
    state = transform(state)
    return state
```

Select GPU or CPU as device

In [4]:

```
device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cpu
```

Define Epsilon annealing schedule generator

In [ ]:

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

# Epsilon annealing schedule generator

def gen_eps_by_episode(epsilon_start, epsilon_final, epsilon_decay):
    eps_by_episode = lambda episode: epsilon_final + (epsilon_start - epsilon_fi
nal) * math.exp(-1. * episode / epsilon_decay)
    return eps_by_episode

epsilon_start = 1.0
epsilon_final = 0.01
epsilon_decay = 500
eps_by_episode = gen_eps_by_episode(epsilon_start, epsilon_final, epsilon_decay)
```

Define game

In [ ]:

```
env_id = 'SpaceInvaders-v0'
env = gym.make(env_id)

model = DDQN(4, env.action_space.n).to(device)

model.load_state_dict(torch.load('checkpoints/spaceInvaders-hw-phi-skip-1M.pth',
map_location=torch.device('cpu') ),)
model.eval()
```

In [3]:

```python
# replay_buffer = ReplayBuffer(1000)

import time

def play_game_CNN(model):
    done = False
    obs = env.reset()
    state = get_state3(obs)
    round_reward = 0
    while(not done):
        action = model.act(state, epsilon_final,env,device)
        reward = 0
        for i in range(3):
            next_obs, i_reward, done, _ = env.step(action)
            # next_obs, reward, done, _ = env.step(action)
            reward += i_reward
            if(done): break
        round_reward += reward
        next_state = get_state3(next_obs)
        env.render()
        time.sleep(0.1)
        state = next_state
    return round_reward

reward_list = []

for i in range(100):
    reward = play_game_CNN(model)
    print(f"round {i}: {reward}")
    reward_list.append(reward)
# time.sleep(3)
env.close()

print("Reward:", sum(reward_list)/len(reward_list))
```

```
cpu
round 0: 285.0
round 1: 250.0
round 2: 245.0
round 3: 245.0
round 4: 275.0
round 5: 190.0
round 6: 230.0
round 7: 175.0
round 8: 85.0
round 9: 175.0
round 10: 105.0
round 11: 160.0
round 12: 165.0
round 13: 170.0
round 14: 220.0
round 15: 175.0
round 16: 270.0
round 17: 230.0
round 18: 220.0
round 19: 215.0
round 20: 230.0
round 21: 175.0
round 22: 215.0
round 23: 215.0
round 24: 220.0
round 25: 325.0
round 26: 445.0
round 27: 165.0
round 28: 245.0
round 29: 175.0
round 30: 100.0
round 31: 85.0
round 32: 235.0
round 33: 220.0
round 34: 230.0
round 35: 220.0
round 36: 235.0
round 37: 190.0
round 38: 220.0
round 39: 220.0
round 40: 230.0
round 41: 165.0
round 42: 225.0
round 43: 215.0
round 44: 105.0
round 45: 200.0
round 46: 415.0
round 47: 215.0
round 48: 215.0
round 49: 230.0
round 50: 175.0
round 51: 220.0
round 52: 230.0
round 53: 215.0
round 54: 245.0
round 55: 185.0
round 56: 105.0
round 57: 230.0
round 58: 120.0
round 59: 245.0
```

```
round 60: 215.0
round 61: 445.0
round 62: 430.0
round 63: 85.0
round 64: 220.0
round 65: 220.0
round 66: 85.0
round 67: 240.0
round 68: 130.0
round 69: 85.0
round 70: 445.0
round 71: 105.0
round 72: 230.0
round 73: 190.0
round 74: 245.0
round 75: 245.0
round 76: 130.0
round 77: 245.0
round 78: 190.0
round 79: 220.0
round 80: 190.0
round 81: 275.0
round 82: 230.0
round 83: 130.0
round 84: 160.0
round 85: 245.0
round 86: 430.0
round 87: 215.0
round 88: 245.0
round 89: 245.0
round 90: 85.0
round 91: 90.0
round 92: 185.0
round 93: 265.0
round 94: 220.0
round 95: 210.0
round 96: 215.0
round 97: 185.0
round 98: 245.0
round 99: 230.0
Reward: 213.3
```

What I have learnt

In this lab we used a more sophisticated state representation by stacking successive frames as the input state representation so as to give the agent some "velocity" input rather than a static snapshot of the scene.

By doing so, the spaceinvader can achieve the average reward of 213.3 and the best score of 445 after training for 5000 epochs. I suspected that my spaceinvader was not able to even get one ufo during the fight which consequently suggests that more training is neeeded.