DQN

AILAB
Hanyang Univ.

공지

공지

다음주 6/8, 6/9 이론 수업 두 번

6/15 기말고사

오늘 실습 내용

1. DQN 구현

1. DQN 구현

DQN 이란

Deep Q-Network

Q Learning을 Neural Network로 구현

DQN 이란

DQN Algorithm

Initialise an empty replay memory.
Initialise the DQN with random (small) weights.

1. Choose an action a to perform in the current state, s, using an ε-greedy strategy (with ε annealed from 1.0 to 0.1).
2. Perform a and receive reward $\mathcal{R}(s, a)$.
3. Observe the new state, $\mathcal{S}(s, a)$.
4. Add $(s, a, \mathcal{R}(s, a), \mathcal{S}(s, a))$ to the replay memory.
5. Sample a minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from the replay memory, and perform stochastic gradient descent on the DQN, based on the loss function

$$\left(Q'(s_i, a) - \left(r_i + \gamma \max_{\alpha} Q'(s_{i+1}, \alpha)\right)\right)^2$$

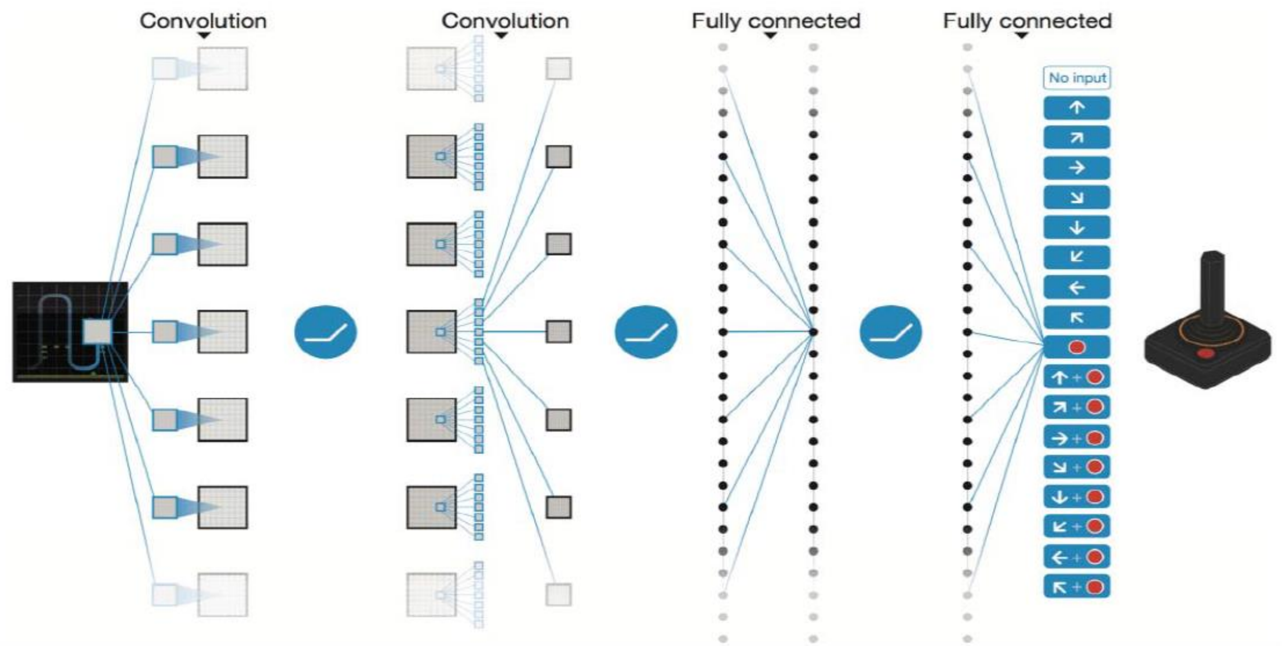
Variable

Target

   where $Q'(s, \cdot)$ is computed by feeding s into the DQN.
6. If the next state is not terminal, go back to step 1.

DQN 이란

DQN 구조
CNN + FC, action classification



Conv [16] → ReLU → Conv [32] → ReLU → FC [256] → ReLU → FC [|A|]

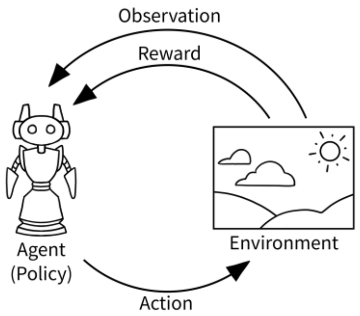
Gym

Gym

https://www.gymlibrary.ml/content/api/#initializing-environments

- env = gym.make

```
import gym
env = gym.make('CartPole-v0')
```



- env.step(action)
  - 현 state에서 action을 진행
- env.reset()
  - 첫 번째 상태로 초기화
- env.render(mode=‘rgb_array’)
  - 현재 화면을 rgb array로 return

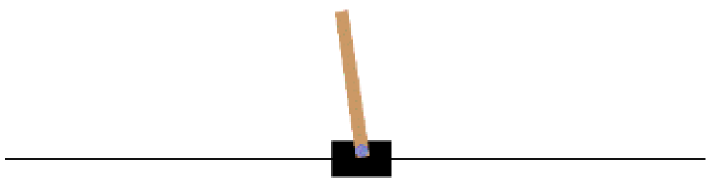
CartPole Problem

CartPole Problem

Cart가 막대를 떨어트리지 않게 하는 것이 목표

- Action: 좌우로 cart로 움직이기, 2개
- State: 스크린 간의 차이값
- Terminate: cart가 선의 끝에 도달하거나, 막대가 아래로 내려갈 때

https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/2b3f06b04b5e96e4772746c20fcb4dcc/reinforcement_q_learning.ipynb (실습코드)
https://www.youtube.com/watch?v=5Q14EjnOJZc (참고영상)



Num	Action
0	Push cart to the left
1	Push cart to the right

DQN 구현

Replay Memory

Queue로 메모리를 만들어서 (s,a,R(s,a),S(s,a)) 관리

Initialise an empty replay memory.
Initialise the DQN with random (small) weights.

1. Choose an action a to perform in the current state, s, using an ε-greedy strategy (with ε annealed from 1.0 to 0.1).
2. Perform a and receive reward $\mathcal{R}(s, a)$.
3. Observe the new state, $\mathcal{S}(s, a)$.
4. Add $(s, a, \mathcal{R}(s, a), \mathcal{S}(s, a))$ to the replay memory.
5. Sample a minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from the replay memory, and perform stochastic gradient descent on the DQN, based on the loss function

$$\left(Q'(s_i, a) - \left(r_i + \gamma \max_{\alpha} Q'(s_{i+1}, \alpha)\right)\right)^2$$

Variable

Target

where $Q'(s, \cdot)$ is computed by feeding s into the DQN.

6. If the next state is not terminal, go back to step 1.

DQN 구현

Replay Memory

```
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))

class ReplayMemory(object):

    def __init__(self, capacity):
        self.memory = deque([],maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

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

4. Add $(s, a, \mathcal{R}(s, a), \mathcal{S}(s, a))$ to the replay memory.

5. Sample a minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from the replay

DQN 구현

DQN 만들기

CNN, FC 구조

Initialise an empty replay memory.
Initialise the DQN with random (small) weights.

1. Choose an action a to perform in the current state, s, using an ε-greedy strategy (with ε annealed from 1.0 to 0.1).
2. Perform a and receive reward $\mathcal{R}(s, a)$.
3. Observe the new state, $\mathcal{S}(s, a)$.
4. Add $(s, a, \mathcal{R}(s, a), \mathcal{S}(s, a))$ to the replay memory.
5. Sample a minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from the replay memory, and perform stochastic gradient descent on the DQN, based on the loss function

$$\left(Q'(s_i, a) - \left(r_i + \gamma \max_{\alpha} Q'(s_{i+1}, \alpha)\right)\right)^2$$

Variable

Target

where $Q'(s, \cdot)$ is computed by feeding s into the DQN.

6. If the next state is not terminal, go back to step 1.

DQN 구현

DQN
CNN+FC 구조
2개의 action 으로 classification

```
class DQN(nn.Module):

    def __init__(self, h, w, outputs):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
        self.bn3 = nn.BatchNorm2d(32)

        # Number of Linear input connections depends on output of conv2d layers
        # and therefore the input image size, so compute it.
        def conv2d_size_out(size, kernel_size = 5, stride = 2):
            return (size - (kernel_size - 1) - 1) // stride  + 1
        convw = conv2d_size_out(conv2d_size_out(conv2d_size_out(w)))
        convh = conv2d_size_out(conv2d_size_out(conv2d_size_out(h)))
        linear_input_size = convw * convh * 32
        self.head = nn.Linear(linear_input_size, outputs)

    # Called with either one element to determine next action, or a batch
    # during optimization. Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = x.to(device)
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        return self.head(x.view(x.size(0), -1))
```

DQN 구현

STATE: get_screen()

현재 상태를 이미지로 가져옴
카트가 좌우로 움직이므로 카트가
가운데로 오게 image 전처리

이미지 위아래 잘라내기

cart의 중심 좌표 가져오기
cart가 이미지의 왼쪽에 있을 때

cart가 이미지의 오른쪽에 있을 때

cart가 가운데로 오게 범위자르기

```python
resize = T.Compose([T.ToPILImage(),
                    T.Resize(40, interpolation=Image.CUBIC),
                    T.ToTensor()])


def get_cart_location(screen_width):
    world_width = env.x_threshold * 2
    scale = screen_width / world_width
    return int(env.state[0] * scale + screen_width / 2.0)  # MIDDLE OF CART

def get_screen():
    # Returned screen requested by gym is 400x600x3, but is sometimes larger
    # such as 800x1200x3. Transpose it into torch order (CHW).
    screen = env.render(mode='rgb_array').transpose((2, 0, 1))
    # Cart is in the lower half, so strip off the top and bottom of the screen
    _, screen_height, screen_width = screen.shape
    screen = screen[:, int(screen_height*0.4):int(screen_height * 0.8)]
    view_width = int(screen_width * 0.6)
    cart_location = get_cart_location(screen_width)
    if cart_location < view_width // 2:
        slice_range = slice(view_width)
    elif cart_location > (screen_width - view_width // 2):
        slice_range = slice(-view_width, None)
    else:
        slice_range = slice(cart_location - view_width // 2,
                            cart_location + view_width // 2)
    # Strip off the edges, so that we have a square image centered on a cart
    screen = screen[:, :, slice_range]
    # Convert to float, rescale, convert to torch tensor
    # (this doesn't require a copy)
    screen = np.ascontiguousarray(screen, dtype=np.float32) / 255
    screen = torch.from_numpy(screen)
    # Resize, and add a batch dimension (BCHW)
    return resize(screen).unsqueeze(0)
```

DQN 구현

Action 선택

주어진 state으로 action을 선택

Initialise an empty replay memory.
Initialise the DQN with random (small) weights.

1. Choose an action a to perform in the current state, s, using an ε-greedy strategy (with ε annealed from 1.0 to 0.1).
2. Perform a and receive reward $\mathcal{R}(s, a)$.
3. Observe the new state, $\mathcal{S}(s, a)$.
4. Add $(s, a, \mathcal{R}(s, a), \mathcal{S}(s, a))$ to the replay memory.
5. Sample a minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from the replay memory, and perform stochastic gradient descent on the DQN, based on the loss function

$$\Big(\underbrace{Q'(s_i, a)}_{\text{Variable}} - \underbrace{\big(r_i + \gamma \max_{\alpha} Q'(s_{i+1}, \alpha)\big)}_{\text{Target}}\Big)^2$$

where $Q'(s, \cdot)$ is computed by feeding s into the DQN.

6. If the next state is not terminal, go back to step 1.

DQN 구현

epsilon decay start->end

Network 선언

Policy Network　　Target Network

$$\left(Q'(s_i, a) - \left(r_i + \gamma \max_{a} Q'(s_{i+1}, a)\right)\right)^2$$

Variable　　Target

where $Q'(s, \cdot)$ is computed by feeding s into the DQN.

Target Network는 학습하지 않고 Policy Network의 Parameter 복제

```
BATCH_SIZE = 128
GAMMA = 0.999
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 200
TARGET_UPDATE = 10

# Get screen size so that we can initialize layers correctly based on shape
# returned from AI gym. Typical dimensions at this point are close to 3x40x90
# which is the result of a clamped and down-scaled render buffer in get_screen()
init_screen = get_screen()
_, _, screen_height, screen_width = init_screen.shape

# Get number of actions from gym action space
n_actions = env.action_space.n

policy_net = DQN(screen_height, screen_width, n_actions).to(device)
target_net = DQN(screen_height, screen_width, n_actions).to(device)
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()

optimizer = optim.RMSprop(policy_net.parameters())
memory = ReplayMemory(10000)


steps_done = 0
```

DQN 구현

ACTION: select_action(state)

State을 DQN에 넣어 left, right 중 값이 높은 것을 action

```
def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * ₩
        math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            # t.max(1) will return largest column value of each row.
            # second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.
            return policy_net(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[random.randrange(n_actions)]], device=device, dtype=torch.long)
```

DQN 구현

DQN Training

Initialise an empty replay memory.
Initialise the DQN with random (small) weights.

1. Choose an action a to perform in the current state, s, using an ε-greedy strategy (with ε annealed from 1.0 to 0.1).
2. Perform a and receive reward $\mathcal{R}(s, a)$.
3. Observe the new state, $\mathcal{S}(s, a)$.
4. Add $(s, a, \mathcal{R}(s, a), \mathcal{S}(s, a))$ to the replay memory.
5. Sample a minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from the replay memory, and perform stochastic gradient descent on the DQN, based on the loss function

$$\left(Q'(s_i, a) - \left(r_i + \gamma \max_{\alpha} Q'(s_{i+1}, \alpha)\right)\right)^2$$

Variable

Target

   where $Q'(s, \cdot)$ is computed by feeding s into the DQN.
6. If the next state is not terminal, go back to step 1.

DQN 구현

DQN Training

Memory에서 sample

Policy network, action에 해당하는 값 추출

Variable

$$\left(Q'(s_i, a) - \left(r_i + \gamma \max_a Q'(s_{i+1}, a)\right)\right)^2$$

Target

where $Q'(s, \cdot)$ is computed by feeding s into the DQN.

Target network

Compute Loss

$$l_n = \begin{cases} 0.5(x_n - y_n)^2 / beta, & \text{if } |x_n - y_n| < beta \\ |x_n - y_n| - 0.5 * beta, & \text{otherwise} \end{cases}$$

```python
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see https://stackoverflow.com/a/19343/3343043 for
    # detailed explanation). This converts batch-array of Transitions
    # to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
    # (a final state would've been the one after which simulation ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)), device=device, dtype=torch.bool
    non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken. These are the actions which would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # Compute V(s_{t+1}) for all next states.
    # Expected values of actions for non_final_next_states are computed based
    # on the "older" target_net; selecting their best reward with max(1)[0].
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0].detach()
    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Compute Huber loss
    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    for param in policy_net.parameters():
        param.grad.data.clamp_(-1, 1)
    optimizer.step()
```

DQN 구현

DQN Training

Initialise an empty replay memory.
Initialise the DQN with random (small) weights.

1. Choose an action a to perform in the current state, s, using an ε-greedy strategy (with ε annealed from 1.0 to 0.1).
2. Perform a and receive reward $\mathcal{R}(s, a)$.
3. Observe the new state, $\mathcal{S}(s, a)$.
4. Add $(s, a, \mathcal{R}(s, a), \mathcal{S}(s, a))$ to the replay memory.
5. Sample a minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from the replay memory, and perform stochastic gradient descent on the DQN, based on the loss function

$$\left(\underbrace{Q'(s_i, a)}_{\text{Variable}} - \underbrace{\left(r_i + \gamma \max_{\alpha} Q'(s_{i+1}, \alpha)\right)}_{\text{Target}}\right)^2$$

   where $Q'(s, \cdot)$ is computed by feeding s into the DQN.
6. If the next state is not terminal, go back to step 1.

DQN 구현

DQN Training

1. Replay Memory에 Transition 추가
2. Training Policy Network
3. Update Target Network

현재 state에서 action 진행

Replay memory에 Transition추가

Policy Network 학습

Target Network parameter update

```python
num_episodes = 50
for i_episode in range(num_episodes):
    # Initialize the environment and state
    env.reset()
    last_screen = get_screen()
    current_screen = get_screen()
    state = current_screen - last_screen
    for t in count():
        # Select and perform an action
        action = select_action(state)
        _, reward, done, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device)

        # Observe new state
        last_screen = current_screen
        current_screen = get_screen()
        if not done:
            next_state = current_screen - last_screen
        else:
            next_state = None

        # Store the transition in memory
        memory.push(state, action, next_state, reward)

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the policy network)
        optimize_model()
        if done:
            episode_durations.append(t + 1)
            plot_durations()
            break
    # Update the target network, copying all weights and biases in DQN
    if i_episode % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict())
```

오늘 실습 내용

1. Cartpole problem DQN 학습

학습시 아래 코드 첫 번째 셀에 추가

```
!apt install xvfb -y
!pip install pyvirtualdisplay
!pip install piglet

from pyvirtualdisplay import Display
display = Display(visible=0, size=(1400, 900))
display.start()
```

오늘 실습 내용

Wandb에서 학습 진행확인

https://docs.wandb.ai/guides/integrations/other/openai-gym

```
env = gym.make('CartPole-v0')
env = gym.wrappers.Monitor(env, f"videos")  # record videos
env = gym.wrappers.RecordEpisodeStatistics(env) # record stats such as returns
# set up matplotlib
is_ipython = 'inline' in matplotlib.get_backend()
if is_ipython:
    from IPython import display

plt.ion()

# if gpu is to be used
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
wandb.init(project="week14_dqn", config=config, monitor_gym=True)
wandb.run.name = "dqn_cartpole"
```