# DQN으로 Atari 게임 강화학습

먼저 여러가지 설정 변수 정의

```
import numpy as np
import gym
import torch
import torch.nn as nn
import torchvision.transforms as T

# Configuration paramaters for the whole setup
seed = 42
gamma = 0.99  # Discount factor for past rewards
epsilon = 1.0  # Epsilon greedy parameter
epsilon_min = 0.1  # Minimum epsilon greedy parameter
epsilon_max = 1.0  # Maximum epsilon greedy parameter
epsilon_interval = (
    epsilon_max - epsilon_min
)  # Rate at which to reduce chance of random action being taken
batch_size = 64  # Size of batch taken from replay buffer
max_steps_per_episode = 1000
```

## Atari 게임 환경

Gym env 의 observation

- Observation의 모양: (210, 160, 3)

액션 정의

- 0: NOOP
- 1: FIRE
- 2: RIGHT
- 3: LEFT

```
# Use the Baseline Atari environment because of Deepmind helper functions
env = gym.make("BreakoutNoFrameskip-v4")
```

```
A.L.E: Arcade Learning Environment (version 0.8.1+53f58b7)
[Powered by Stella]
```

## 네트워크 정의하기

참고: Conv2d 파라미터

- in_channels (int) – Number of channels in the input image
- out_channels (int) – Number of channels produced by the convolution
- kernel_size (int or tuple) – Size of the convolving kernel
- stride (int or tuple, optional) – Stride of the convolution. Default: 1
- padding (int, tuple or str, optional) – Padding added to all four sides of the input. Default: 0
- padding_mode (str, optional) – 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

```
num_actions = env.action_space.n

class QModel(nn.Module):
    def __init__(self, num_actions):
        super(QModel, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=8, stride=4)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.dropout = nn.Dropout(p=0.3)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(3136, 512)
        self.fc2 = nn.Linear(512, num_actions)

    def forward(self, x):
        x = nn.functional.relu(self.conv1(x))
        x = nn.functional.relu(self.conv2(x))
        x = self.dropout(x)
        x = nn.functional.relu(self.conv3(x))
        x = self.flatten(x)
        x = nn.functional.relu(self.fc1(x))
        x = self.dropout(x)
        action = self.fc2(x)
        return action
```

## 모델 빌딩 & 로스 및 최적화 계산기 만들기

```
# The first model makes the predictions for Q-values which are used to
# make a action.
model = QModel(num_actions)
```

```
# Build a target model for the prediction of future rewards.
# The weights of a target model get updated every 10000 steps thus when the
# loss between the Q-values is calculated the target Q-value is stable.
model_target = QModel(num_actions)

loss_function = nn.SmoothL1Loss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.00025)
```

### Replay Buffer 정의

```
# Experience replay buffers
action_history = []
state_history = []
state_next_history = []
rewards_history = []
done_history = []
episode_reward_history = []
running_reward = 0
episode_count = 0
frame_count = 0

# Number of frames to take random action and observe output
epsilon_random_frames = 50000
# Number of frames for exploration
epsilon_greedy_frames = 100000.0
# Maximum replay length
# Note: The Deepmind paper suggests 1000000 however this causes memory issues
max_memory_length = 500000
# Train the model after 4 actions
update_after_actions = 4
# How often to update the target network
update_target_network = 10000
```

### 전처리

Observation 을 QModel의 입력 타입으로 전처리

- PIL Image 객체로 바꾸고
- 그레이 스케일로 바꾸고
- 84 * 84 짜리 이미지로 리사이징
- 지금까지 np.array이니 torch.tensor로 캐스팅
- 마지막으로 batch 축 추가

```
preprocess = T.Compose([
    T.ToPILImage(),
    T.Grayscale(),
    T.Resize((84, 84)),
    T.ToTensor()
])

# Function to preprocess the state
def preprocess_state(state):
    state = preprocess(state).unsqueeze(0)
    return state
```

### Epsilon-greedy 액션 선택 함수

학습시 에피소드 생성하면서 사용 (주의: 입력은 batch axis 없음)

```
# Function to select an action
def get_greedy_epsilon(model, state):
    global epsilon

    #if frame_count < epsilon_random_frames or np.random.rand(1)[0] < epsilon:
    if np.random.rand(1)[0] < epsilon:
        action = np.random.randint(num_actions)
    else:
        with torch.no_grad():
            # add a batch axis
            #state = state.unsqueeze(0)
            # compute the q-values
            q_values = model(state)
            # the action of maximum q-value
            action = q_values.argmax().item()

    # decay epsilon
    epsilon -= epsilon_interval / epsilon_greedy_frames
    epsilon = max(epsilon, epsilon_min)

    return action
```

### Greedy 액션 선택 함수

나중에 evaluation 시 사용

```
def get_greedy_action(model, state):
    global epsilon

    with torch.no_grad():
        #state = state.unsqueeze(0) # batch dimension
        q_values = model(state)
        action = q_values.argmax().item()

    return action
```

### Update 파트

- Replay buffer 에서 batch하나를 샘플링하고,
- model을 update한다.

```
# sample a batch of _batch_size from replay buffers
# return numpy.ndarrays
def sample_batch(_batch_size):
    # Get indices of samples for replay buffers
    indices = np.random.choice(range(len(done_history)), size=_batch_size, replace=False)

    state_sample = np.array([state_history[i].squeeze(0).numpy() for i in indices])
    state_next_sample = np.array([state_next_history[i].squeeze(0).numpy() for i in indices])
    rewards_sample = np.array([rewards_history[i] for i in indices], dtype=np.float32)
    action_sample = np.array([action_history[i] for i in indices])
    done_sample = np.array([float(done_history[i]) for i in indices])

    return state_sample, state_next_sample, rewards_sample, action_sample, done_sample
```

```
# Function to update the Q-network
def update_network():
    # sample a batch of ...
    state_sample, state_next_sample, rewards_sample, action_sample, done_sample = \
        sample_batch(batch_size)

    # Convert numpy arrays to PyTorch tensors
    state_sample = torch.tensor(state_sample, dtype=torch.float32)
    state_next_sample = torch.tensor(state_next_sample, dtype=torch.float32)
    action_sample = torch.tensor(action_sample, dtype=torch.int64)
    rewards_sample = torch.tensor(rewards_sample, dtype=torch.float32)
    done_sample = torch.tensor(done_sample, dtype=torch.float32)

    # Compute the target Q-values for the states
    with torch.no_grad():
        future_rewards = model_target(state_next_sample)

        # compute the q-value for the next state and the action maximizing the q-value
        max_q_values = future_rewards.max(dim=1).values

        # compute the target q-value
        # if the step was final, max_q_values should not be added
        target_q_values = rewards_sample + gamma * max_q_values * (1. - done_sample)

    # It's forward propagation! Compute the Q-values for the taken actions
    q_values = model(state_sample)
    q_values_action = q_values.gather(dim=1, index=action_sample.unsqueeze(1)).squeeze(1)

    # Compute the loss
    loss = loss_function(q_values_action, target_q_values)

    # Perform the optimization step
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

## Run DQN Tranining

```
while True:  # Run until solved
    state, info = env.reset()
    state, reward, done, _, info = env.step(1)
    state = preprocess_state(state)
    episode_reward = 0

    for timestep in range(1, max_steps_per_episode):
        frame_count += 1

        # Select an action
        action = get_greedy_epsilon(model, state)

        # Take the selected action
        state_next, reward, done, _, info = env.step(action)
        state_next = preprocess_state(state_next)

        episode_reward += reward
```

```
        # Store the transition in the replay buffer
        action_history.append(action)
        state_history.append(state)
        state_next_history.append(state_next)
        rewards_history.append(reward)
        done_history.append(done)

        state = state_next

        # Update every fourth frame and once batch size is over 32
        if frame_count % update_after_actions == 0 and len(done_history) > batch_size:
            update_network()

        if frame_count % update_target_network == 0:
            model_target.load_state_dict(model.state_dict())

        # Limit the state and reward history
        if len(rewards_history) > max_memory_length:
            del rewards_history[:1]
            del state_history[:1]
            del state_next_history[:1]
            del action_history[:1]
            del done_history[:1]

        if done:
            break

    episode_count += 1
    episode_reward_history.append(episode_reward)

    # Update running reward to check condition for solving
    if len(episode_reward_history) > 100:
        del episode_reward_history[:1]
    running_reward = np.mean(episode_reward_history)

    if episode_count % 10 == 0:
        print(f"Episode: {episode_count}, Frame count: {frame_count}, Running reward: {running_reward}")

    if episode_count % 5000 == 0:
        torch.save(model, 'model.{}'.format(episode_count))
    if running_reward > 20:
        print(f"Solved at episode {episode_count}!")
        break
    if episode_count % 200 == 0:
        break


torch.save(model, 'model.final')
```

```
Episode: 10, Frame count: 6499, Running reward: 1.0
Episode: 20, Frame count: 13280, Running reward: 1.05
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[19], line 30
     28 # Update every fourth frame and once batch size is over 32
     29 if frame_count % update_after_actions == 0 and len(done_history) > batch_size:
---> 30     update_network()
     32 if frame_count % update_target_network == 0:
     33     model_target.load_state_dict(model.state_dict())

Cell In[15], line 16, in update_network()
     14 # Compute the target Q-values for the states
     15 with torch.no_grad():
---> 16     future_rewards = model_target(state_next_sample)
     18     # compute the q-value for the next state and the action maximizing the q-value
     19     max_q_values = future_rewards.max(dim=1).values

File ~/miniconda3/envs/torch/lib/python3.10/site-packages/torch/nn/modules/module.py:1501, in Module._call_impl(
   1496 # If we don't have any hooks, we want to skip the rest of the logic in
   1497 # this function, and just call forward.
   1498 if not (self._backward_hooks or self._backward_pre_hooks or self._forward_hooks or self._forward_pre_hool
   1499         or _global_backward_pre_hooks or _global_backward_hooks
   1500         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1501     return forward_call(*args, **kwargs)
   1502 # Do not call functions when jit is used
   1503 full_backward_hooks, non_full_backward_hooks = [], []

Cell In[3], line 18, in QModel.forward(self, x)
     16 x = nn.functional.relu(self.conv2(x))
     17 x = self.dropout(x)
---> 18 x = nn.functional.relu(self.conv3(x))
     19 x = self.flatten(x)
     20 x = nn.functional.relu(self.fc1(x))

File ~/miniconda3/envs/torch/lib/python3.10/site-packages/torch/nn/modules/module.py:1501, in Module._call_impl(
   1496 # If we don't have any hooks, we want to skip the rest of the logic in
```

```
   1497 # this function, and just call forward.
   1498 if not (self._backward_hooks or self._backward_pre_hooks or self._forward_hooks or self._forward_pre_hool
   1499         or _global_backward_pre_hooks or _global_backward_hooks
   1500         or _global_forward_hooks or _global_forward_pre_hooks):
→ 1501     return forward_call(*args, **kwargs)
   1502 # Do not call functions when jit is used
   1503 full_backward_hooks, non_full_backward_hooks = [], []

File ~/miniconda3/envs/torch/lib/python3.10/site-packages/torch/nn/modules/conv.py:463, in Conv2d.forward(self, i
    462 def forward(self, input: Tensor) → Tensor:
──> 463     return self._conv_forward(input, self.weight, self.bias)

File ~/miniconda3/envs/torch/lib/python3.10/site-packages/torch/nn/modules/conv.py:459, in Conv2d._conv_forward(s
    455 if self.padding_mode ≠ 'zeros':
    456     return F.conv2d(F.pad(input, self._reversed_padding_repeated_twice, mode=self.padding_mode),
    457                     weight, bias, self.stride,
    458                     _pair(0), self.dilation, self.groups)
──> 459 return F.conv2d(input, weight, bias, self.stride,
    460                 self.padding, self.dilation, self.groups)

KeyboardInterrupt:
```

# Evaluation

```python
import time, sys
from IPython.display import clear_output
from matplotlib import animation
import matplotlib.pyplot as plt
import glob
import imageio

anim_file = 'atari.gif'

turn =  0
board, info = env.reset()
state = preprocess_state(board)
board, reward, done, _, info = env.step(1)
state = preprocess_state(board)
plt.imshow(board)
plt.savefig('image_at_turn_{:04d}.png'.format(turn))

for timestep in range(1, 100):
    turn += 1
    action = get_greedy_action(model, state.to(device))
    print(action)
    board, reward, done, _, info = env.step(action)
    state = preprocess_state(board)
    plt.imshow(board)
    plt.savefig('image_at_turn_{:04d}.png'.format(turn))

    if done:
        break
```

```python
# generate animated gif file
with imageio.get_writer(anim_file, mode='I') as writer:
    filenames = glob.glob('image_at_turn_*.png')
    filenames = sorted(filenames)
    for filename in filenames:
        print(filename)
        image = imageio.imread(filename)
        writer.append_data(image)
    image = imageio.imread(filename)
    writer.append_data(image)
```