Deep Q Learning with Gym and Tensorflow

Byeongchang Kim

May 19, 2017

https://bckim92.github.io/DQN-with-Gym-talk

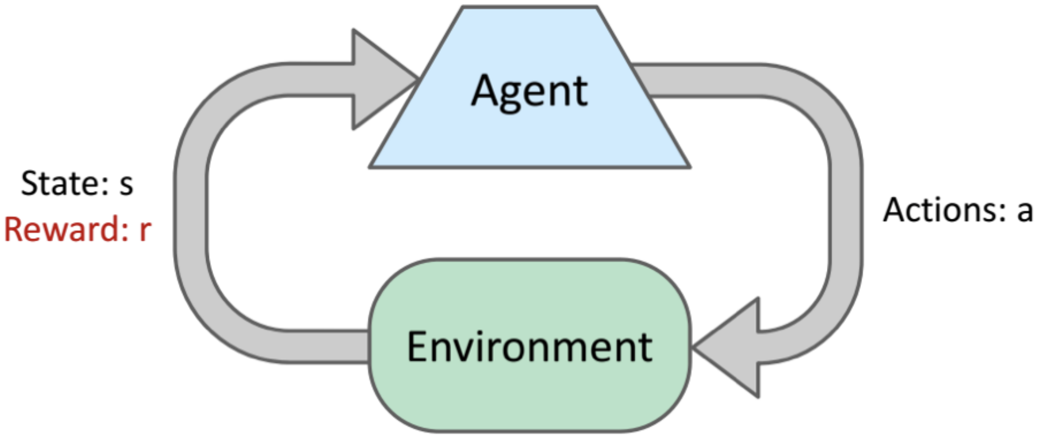
About

이번 실습에서는 Deep Q-Learning 에 대해 간략히 살펴본 후,

이를 Tensorflow와 OpenAI Gym을 이용해서 구현해보는 것을 목표로 합니다.

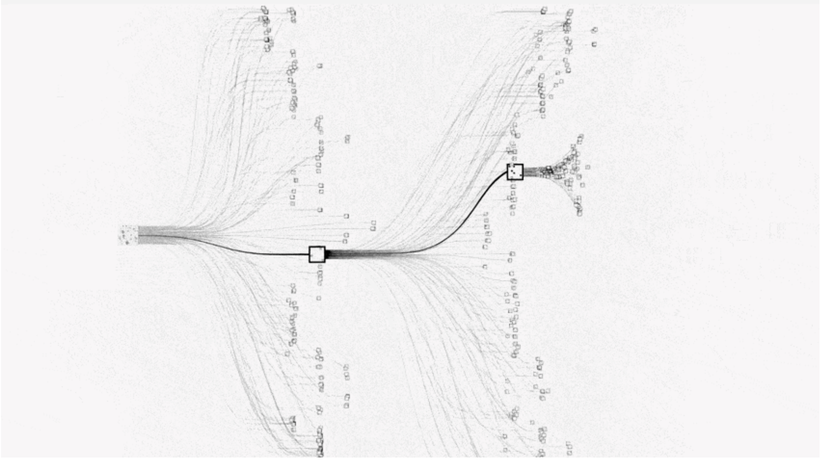
Deep Q-Learning

What is Reinforcement Learning



- RL is a general-purpose framework for artificial intelligence
  - RL is for an agent with the capacity to act
  - Each action a_t, influences the agent's future state s_t
  - Success is measured by a *scalar* reward r_t
  - Must (learn to) act so as to maximize expected rewards

Examples of Reinforcement Learning



Terrain-Adaptive Locomotion Skills using Deep Reinforcement Learning
Xue Bin Peng, Glen Berseth, Michiel van de Panne
University of Biritish Columbia
includes audio

(Clip credit: Google DeepMind, X. B. Peng)

Policy and Value Functions

- Policy π is a behaviour function selection actions given states

$$a = \pi(s)$$

- Value function $Q^{\pi}(s, a)$ is expected total reward from state s and action a under policy π

$$Q^{\pi}(s, a) = \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s, a]$$

  - means, "How good is action a in state s?"

Approaches to Reinforcement Learning

- Value-based RL
  - Estimate the optimal value function $Q^*(s, a)$
  - This is the maximum value achievable under any policy
  - The approach we took
- Policy-based RL
  - Search directly for the optimal policy $\pi^*(s)$
  - This is the policy achieving maximum future reward
    - e.g. Actor-Critic Model, TRPO
- Model-based RL
  - Build a transition model of the environment
  - Modeling an environment
  - Plan (by lookahead) using model

Deep Reinforcement Learning

- Use deep (neural) network to represent value function / policy / model
- Optimize function end-to-end
    - Using stochastic gradient descent

Optimize Value Function

- Bellman's Principle of Optimality

  Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision. (See Bellman, 1957, Chap. 3.3.)

- Value function can be unrolled recursively

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s, a] \\ &= \mathbb{E}_{s'}\left[r + \gamma Q^{\pi}(s', a') | s, a\right] \end{aligned}$$

- Value iteration algorithms solve the Bellman equation

$$Q_{i+1}(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q_i(s', a') | s, a\right]$$

Deep Q-Learning

- Find value function $Q(s, a)$ with Q-network with weights w

$$Q(s, a, w) \approx Q^{\pi}(s, a)$$

- Define objective function by mean-squared error in Q-values

$$L(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max a' Q(s', a', w)}_{target} - Q(s, a, w)\right)^2\right]$$

Stability Issues with Deep Q-Learning

- Naive Q-learning oscillates or diverges with neural nets
- **Issue 1 : Data is sequential**
  - Successive samples are correlated, not independent
  - $\rightarrow$ Use experience replay
- **Issue 2 : Policy changes rapidly with slight changes to Q-values**
  - Policy may oscillate
  - Distribution of data can swing from one extreme to another
  - $\rightarrow$ Freeze target Q-network
- **Issue 3: Scale of rewards and Q-values is unknown**
  - Naive Q-learning gradients can be largely unstable when backpropagated
  - $\rightarrow$ Clip rewards or normalize network adaptively

Stable Deep RL (1) : Experience Replay

- One of the most valuable techniques
- To remove correlations, build data-set(memory!) from agent's own experience
    - Take action a_t according to ϵ-greedy policy
    - Store transition (s_t, a_t, r_t, s_{t+1}) in replay memory D
    - Sample random mini-batch of transitions (s, a, r, s') from D
    - Optimize MSE between Q-network and Q-learning targets, e.g.

$$L(w) = \mathbb{E}_{s,a,r,s' \ D}\left[\left(r + \gamma \max_{a'} Q(s', a', w) - Q(s, a, w)\right)^2\right]$$

- Training can be done independently from execution
- Parallelism

Stable Deep RL (2) : Fixed target Q-Network

- To avoid oscillations, fix parameters used in Q-learning target
  - Compute Q-learning targets w.r.t. old, fixed parameters w^-
  - Optimize MSE between Q-networks and Q-learning targets

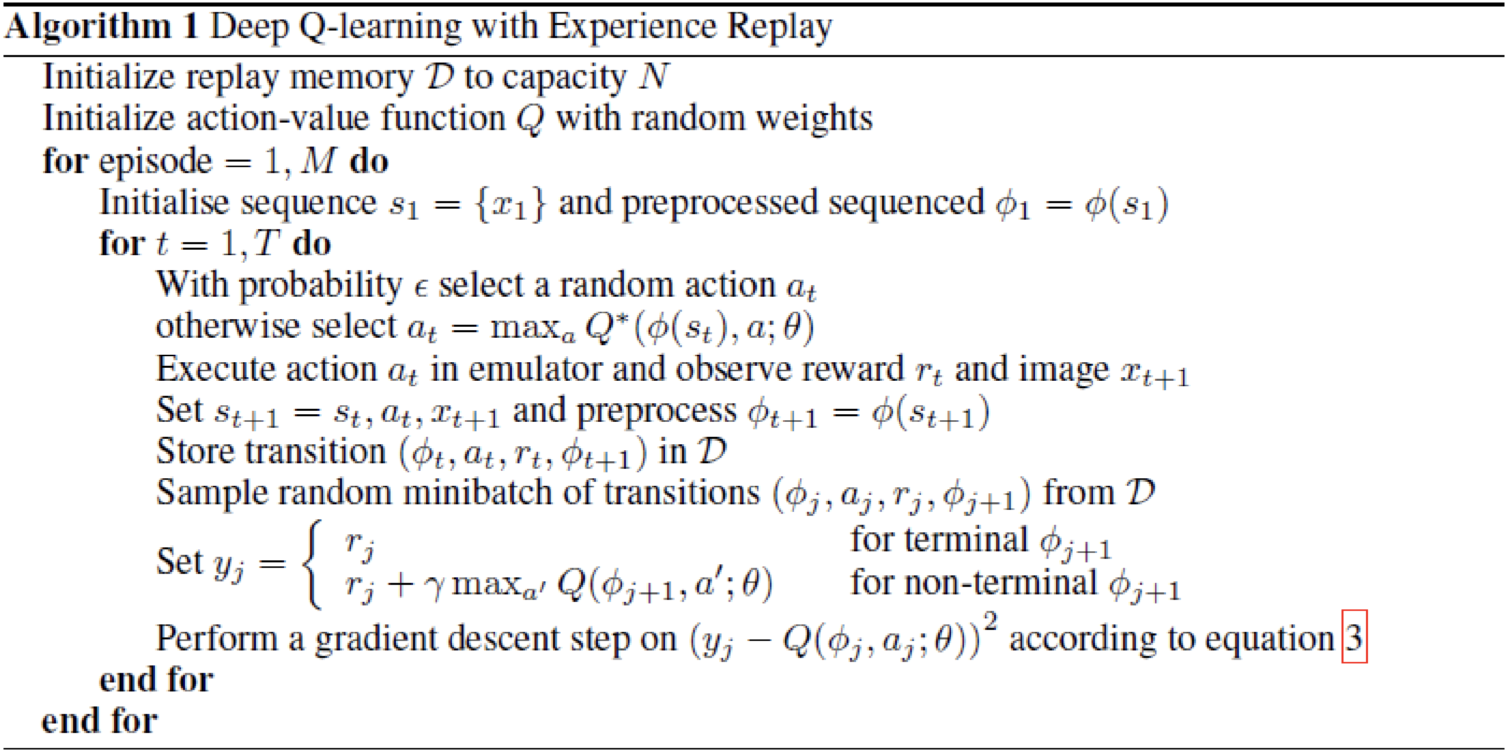
$$L(w) = \mathbb{E}_{s,a,r,s' \ D}\left[\left(r + \gamma \max_{a'} Q(s', a', w^-) - Q(s, a, w)\right)^2\right]$$

  - periodically update fixed parameters $w^- \leftarrow w$

Stable Deep RL (3) : Reward / Value range

- Clips the rewards to range
    - In general cases, we lose some information
        - Can't tell difference between small and large rewards
    - In our case, our rewards naturally clipped to $[-1, 1]$
- Better solution?
    - Use Huber loss! (we will cover this later)

Deep Q-Learning Algorithm



Algorithm 1 Deep Q-learning with Experience Replay

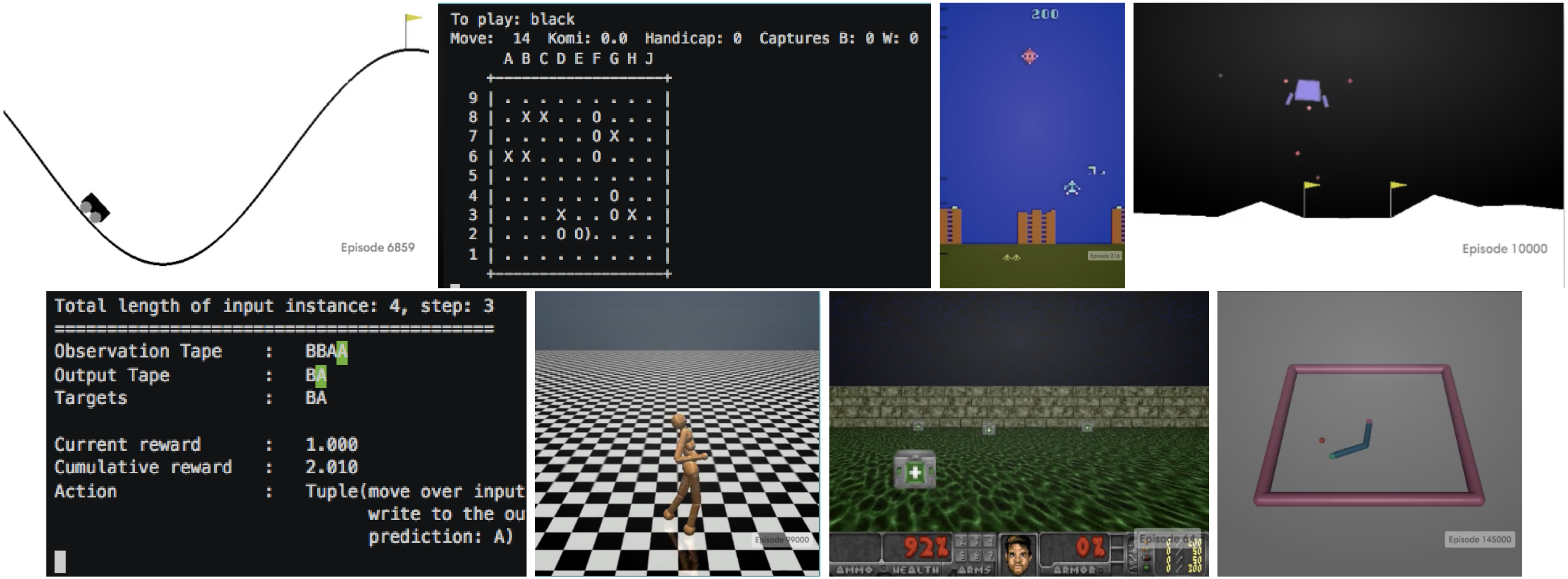
```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

V. Mnih et al. Playing Atari with Deep Reinforcement Learning. In *NIPS*, 2013

OpenAI Gym

OpenAI Gym

- A toolkit for developing and comparing reinforcement learning algorithms
- It supports teaching agents everything from walking to playing games like Pong or Go



```
To play: black
Move:  14  Komi: 0.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E F G H J
    +-------------------+
  9 | . . . . . . . . . |
  8 | . X X . . O . . . |
  7 | . . . . . O X . . |
  6 | X X . . . O . . . |
  5 | . . . . . . . . . |
  4 | . . . . . . O . . |
  3 | . . . X . . O X . |
  2 | . . . O O). . . . |
  1 | . . . . . . . . . |
    +-------------------+
```

```
Total length of input instance: 4, step: 3
==========================================
Observation Tape    :   BBAA
Output Tape         :   BA
Targets             :   BA

Current reward      :   1.000
Cumulative reward   :   2.010
Action              :   Tuple(move over input
                              write to the ou
                              prediction: A)
```

G. Rockman et al. OpenAI Gym. In *arXiv:1606.01540*, 2016

Getting Started with OpenAI Gym

Installation (https://github.com/openai/gym)

1. Install all dependencies

```
apt-get install -y python-numpy python-dev cmake zlib1g-dev libjpeg-dev xvfb libav-to
```

2. Install OpenAI Gym

```
pip install 'gym[all]'
```

Getting Started with OpenAI Gym

Run Gym environment (https://gym.openai.com)

```python
import gym
env = gym.make("Taxi-v1")
observation = env.reset()
for _ in range(1000):
  env.render()
  action = env.action_space.sample() # your agent here (this takes random actions)
  observation, reward, done, info = env.step(action)
```

Getting Started with OpenAI Gym

Upload your results (https://gym.openai.com)

```
import gym
from gym import wrappers

env = gym.make("FrozenLake-v0")
env = wrappers.Monitor(env, "/tmp/gym-results")
observation = env.reset()
for _ in range(1000):
  env.render()
  action = env.action_space.sample() # your agent here (this takes random actions)
  observation, reward, done, info = env.step(action)
  if done:
    env.reset()

env.close()
gym.upload("/tmp/gym-results", api_key="YOUR_API_KEY")
```

DQN in Tensorflow

Disclaimer

이후 슬라이드의 코드는 **가독성**을 위해 많은 코드를 생략했습니다.

(실제 코드와는 다릅니다.)

Disclaimer

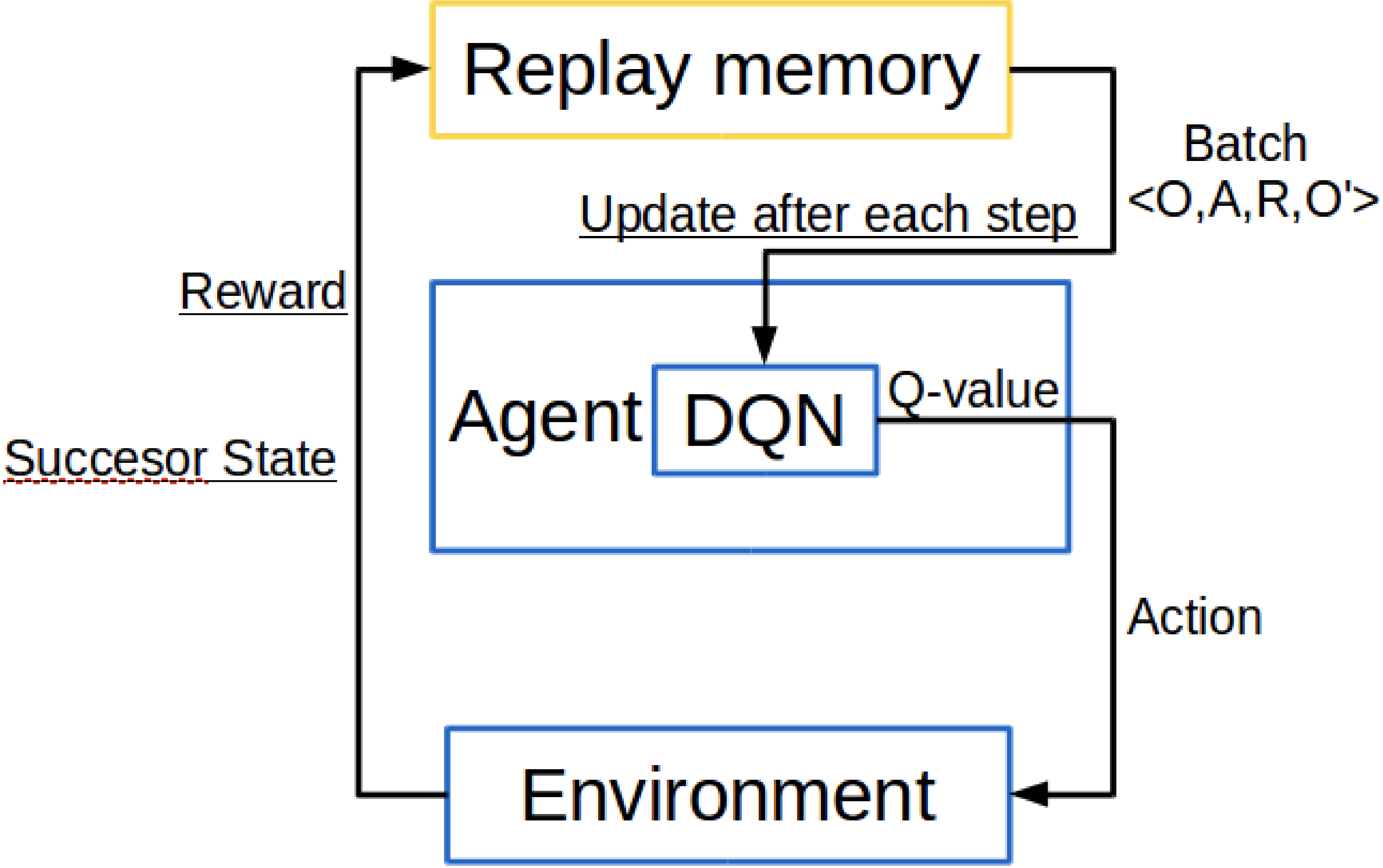
이후 슬라이드의 코드는 **가독성**을 위해 많은 코드를 생략했습니다.

(실제 코드와는 다릅니다.)

또한, 기존에 DQN을 구현한 훌륭한 코드들이 많으니, 아래 링크들을 참고하시면 공부하는데 도움이 많이 될 것 같습니다.

https://github.com/carpedm20/deep-rl-tensorflow
https://github.com/nivwusquorum/tensorflow-deepq

System Overview



(Image credit: modulabs)

Code Structure

```
./
├── requirements.txt
├── main.py
├── dqn
|    ├── agent.py
|    └── replay_memory.py
└── utils
     └── utils.py
```

- requirements.txt

코드 실행에 필요한 패키지 리스트들을 적어놓은 파일입니다.

`pip install -r requirements.txt`로 적혀져있는 패키지들을 한번에 설치할 수 있습니다.

Code Structure

```
./
├── requirements.txt
├── main.py
├── dqn
|    ├── agent.py
|    └── replay_memory.py
└── utils
     └── utils.py
```

- main.py

DQN agent와 Gym environment가 실행되는 부분이 구현되어 있는 파일입니다.

- utils/utils.py

필요한 utility method들이 구현되어 있는 파일입니다.

Code Structure

```
./
├── requirements.txt
├── main.py
├── dqn
|   ├── agent.py
|   └── replay_memory.py
└── utils
    └── utils.py
```

- dqn/agent.py

DQN agent가 구현되어 있는 파일입니다.

- dqn/replay_memory.py

Experience replay에 필요한 replay memory가 구현되어 있는 파일입니다.

Environment Setup (main.py)

```python
def main():
  env = gym.make("SpaceInvaders-v0")
  agent = Agent(FLAGS, env.action_space.n)
  for step in tqdm(range(FLAGS.num_steps), ncols=70):
    if done: env.reset()

    reward = 0.
    for _ in xrange(FLAGS.action_repeat):
      observation, reward_, done, info = env.step(action)
      reward += reward_
      if done: reward -= 1.; break

    observation = atari_preprocessing(observation, width, height)

    action = agent.train(observation, reward, done, step)
```

Agent와 Gym environment를 만들어줍니다.

Environment Setup (main.py)

```
def main():
  env = gym.make("SpaceInvaders-v0")
  agent = Agent(FLAGS, env.action_space.n)
  for step in tqdm(range(FLAGS.num_steps), ncols=70):
    if done: env.reset()

    reward = 0.
    for _ in xrange(FLAGS.action_repeat):
      observation, reward_, done, info = env.step(action)
      reward += reward_
      if done: reward -= 1.; break

    observation = atari_preprocessing(observation, width, height)

    action = agent.train(observation, reward, done, step)
```

매 frame을 보는 대신 k번마다 frame을 보는 frame-skipping을 적용시켜줍니다.

DQN paper 에서는 4번마다 frame을 봅니다.

Environment Setup (main.py)

```
def main():
  env = gym.make("SpaceInvaders-v0")
  agent = Agent(FLAGS, env.action_space.n)
  for step in tqdm(range(FLAGS.num_steps), ncols=70):
    if done: env.reset()

    reward = 0.
    for _ in xrange(FLAGS.action_repeat):
      observation, reward_, done, info = env.step(action)
      reward += reward_
      if done: reward -= 1.; break

    observation = atari_preprocessing(observation, width, height)

    action = agent.train(observation, reward, done, step)
```

그레이 스케일로 변환하고, 작은 사이즈로 줄여줍니다 (utils/utils.py)

```
def atari_preprocessing(raw_image, width, height):
  gray_image = np.dot(raw_image[..., :3], [0.299, 0.587, 0.114])
  return scipy.misc.resize(gray_image / 255, [width, height])
```

Environment Setup (main.py)

```
def main():
  env = gym.make("SpaceInvaders-v0")
  agent = Agent(FLAGS, env.action_space.n)
  for step in tqdm(range(FLAGS.num_steps), ncols=70):
    if done: env.reset()

    reward = 0.
    for _ in xrange(FLAGS.action_repeat):
      observation, reward_, done, info = env.step(action)
      reward += reward_
      if done: reward -= 1.; break

    observation = atari_preprocessing(observation, width, height)

    action = agent.train(observation, reward, done, step)
```

현재 프레임을 보고, Q 값을 최대화 시키는 action을 예측합니다.

또한 agent를 학습시킵니다.

Environment Setup (main.py)

```
def main():
  env = gym.make("SpaceInvaders-v0")
  agent = Agent(FLAGS, env.action_space.n)
  for step in tqdm(range(FLAGS.num_steps), ncols=70):
    if done: env.reset()

    reward = 0.
    for _ in xrange(FLAGS.action_repeat):
      observation, reward_, done, info = env.step(action)
      reward += reward_
      if done: reward -= 1.; break

    observation = atari_preprocessing(observation, width, height)

    action = agent.train(observation, reward, done, step)
```

Build Input Pipeline for Model (dqn/agent.py)

```
def __init__(self):
  self.replay_memory = ReplayMemory()
  self.history = History()

  # Build placeholders
  self.state = tf.placeholder(tf.float32, [None, height, width, history_length])
  self.next_state = tf.placeholder(tf.float32, [None, height, width, history_length])
  self.action = tf.placeholder(tf.int32, [None])
  self.reward = tf.placeholder(tf.float32, [None])
  self.done = tf.placeholder(tf.float32, [None])
```

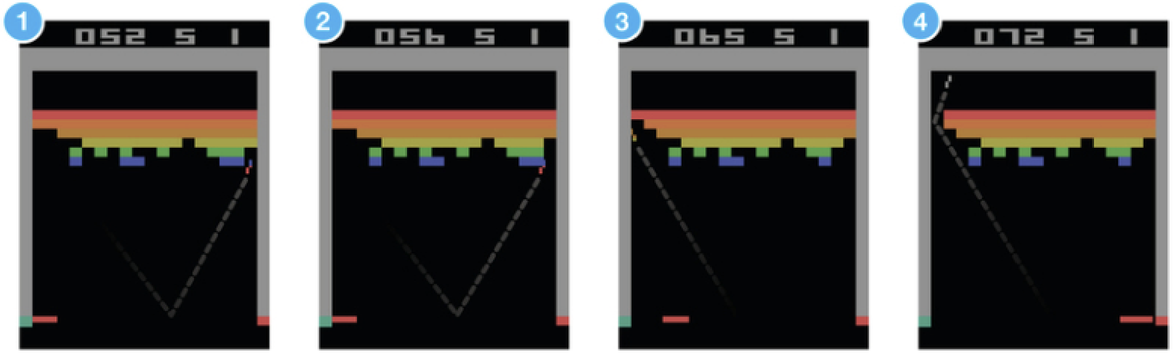
Experience replay를 위한 replay memory를 만들어줍니다.

Build Input Pipeline for Model (dqn/agent.py)

```python
def __init__(self):
  self.replay_memory = ReplayMemory()
  self.history = History()

  # Build placeholders
  self.state = tf.placeholder(tf.float32, [None, height, width, history_length])
  self.next_state = tf.placeholder(tf.float32, [None, height, width, history_length])
  self.action = tf.placeholder(tf.int32, [None])
  self.reward = tf.placeholder(tf.float32, [None])
  self.done = tf.placeholder(tf.float32, [None])
```

네 장의 이미지를 붙여 하나의 이미지로 만들어 줍니다



(Image credit: nervanasys)

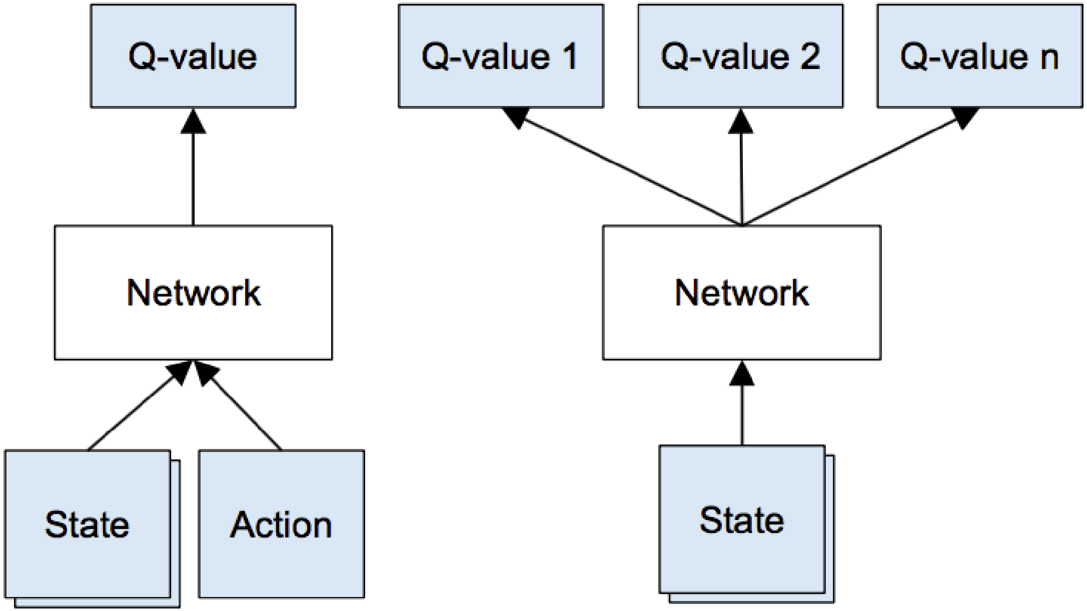
Build Input Pipeline for Model (dqn/agent.py)

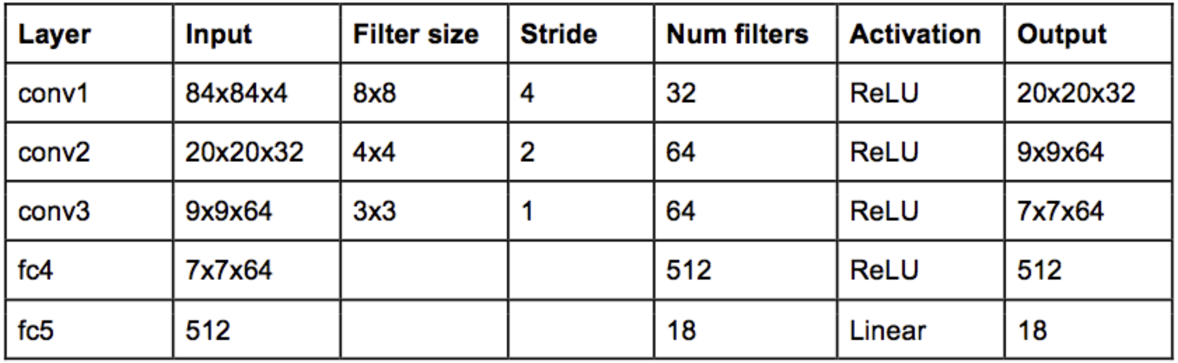
```python
def __init__(self):
  self.replay_memory = ReplayMemory()
  self.history = History()

  # Build placeholders
  self.state = tf.placeholder(tf.float32, [None, height, width, history_length])
  self.next_state = tf.placeholder(tf.float32, [None, height, width, history_length])
  self.action = tf.placeholder(tf.int32, [None])
  self.reward = tf.placeholder(tf.float32, [None])
  self.done = tf.placeholder(tf.float32, [None])
```

(s_t, a_t, r_t, s_{t+1})을 넣어줄 placeholder를 만들어 줍니다.

DQN Architecture





Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

(Image credit: nervanasys)

Build Source/Target Network (dqn/agent.py)

```
def build():
  # Build network
  source_q = _build_net(state, 'source', True)
  target_q = _build_net(state, 'target', False)
  # Compute loss and gradient
  ...

  # Update target network
  ...
```

3 Conv-layer + 2 FC-layer with $[\#\ action\ space]$ outputs

```
def _build_net(observation, name='source', trainable=True):
  with tf.variable_scope(name):
   with arg_scope([layers.conv2d, layers.fully_connected], trainable=trainable, ...):
     conv1 = layers.conv2d(observation, num_outputs=32, kernel_size=8, stride=4k
     conv2 = layers.conv2d(conv1, num_outputs=64, kernel_size=4, stride=2)
     conv3 = layers.conv2d(conv2, num_outputs=64, kernel_size=3, stride=1)
     conv3_flat = tf.reshape(conv3, [-1, reduce(lambda x, y: x * y, conv3.get_shape().as_
     fc4 = layers.fully_connected(conv3_flat, 512)
     q = layers.fully_connected(fc4, self.action_space)
  return q
```

Build Inference Op (dqn/agent.py)

```python
def build():
  # Build network
  source_q = _build_net(state, 'source', True)
  target_q = _build_net(state, 'target', False)
  inference_action_op = tf.argmax(source_q, dimension=1)

  # Compute loss and gradient
  ...

  # Update target network
  ...
```

$\underset{a}{\operatorname{argmax}}\, Q(s, a, w)$ 를 계산해주는 `inference_action_op`을 만들어줍니다.

Compute Loss and Gradient (dqn/agent.py)

```
def build():
  # Build network
  ...

  # Compute loss and gradient
  action_one_hot = tf.one_hot(current_action, self.action_space, 1.0, 0.0)
  q_acted = tf.reduce_sum(source_q * action_one_hot, reduction_indices=1)
  max_target_q = tf.reduce_max(target_q, axis=1)
  delta = (1 - done) * self.config.gamma * max_target_q + current_reward - q_acted
  loss = tf.reduce_mean(clipped_error(delta))
  train_op = tf.train.RMSPropOptimizer(lr, momentum=0.95, epsilon=0.1).minimize(loss)

  # Update target network
  ...
```

Delta 값인 $\underbrace{r + \gamma \max a' Q(s', a', w^-)}_{target} - Q(s, a, w)$ 를 계산해 줍니다.

Compute Loss and Gradient (dqn/agent.py)

```python
def build():
  # Build network
  ...

  # Compute loss and gradient
  action_one_hot = tf.one_hot(current_action, self.action_space, 1.0, 0.0)
  q_acted = tf.reduce_sum(source_q * action_one_hot, reduction_indices=1)
  max_target_q = tf.reduce_max(target_q, axis=1)
  delta = (1 - done) * self.config.gamma * max_target_q + current_reward - q_acted
  loss = tf.reduce_mean(clipped_error(delta))
  train_op = tf.train.RMSPropOptimizer(lr, momentum=0.95, epsilon=0.1).minimize(loss)

  # Update target network
  ...
```

Delta를 $[-1, 1]$로 clipping 해줍니다.

Issue with Delta Clipping

그런데, 이 때 치명적인 실수가 발생할 수 있습니다.
(https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b)

Issue with Delta Clipping

그런데, 이 때 치명적인 실수가 발생할 수 있습니다.
(https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b)

일반적으로 clipping에는 `tf.clip_by_value`를 사용합니다.

```
clipped_delta = tf.clip_by_value(delta, clip_value_min=-1.0, clip_value_max=1.0)
```

Issue with Delta Clipping

그런데, 이 때 치명적인 실수가 발생할 수 있습니다. (https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b)

일반적으로 clipping에는 `tf.clip_by_value`를 사용합니다.

```
clipped_delta = tf.clip_by_value(delta, clip_value_min=-1.0, clip_value_max=1.0)
```

하지만, 이를 사용하게 되면 clip 되었을 때 scalar 값이 나오게 됩니다.

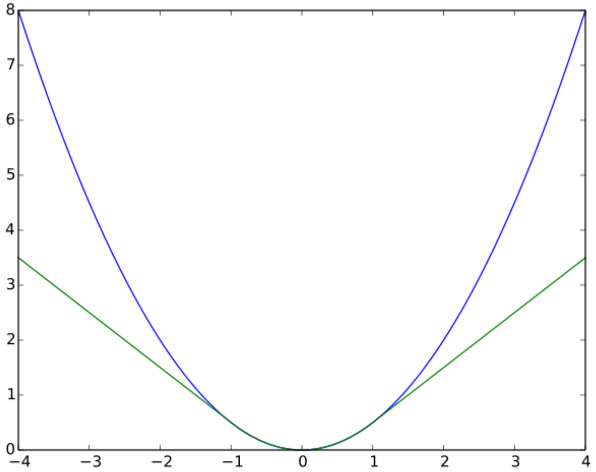
따라서, 미분값이 0이 나오게 되며, gradient가 0인채로 backpropagation을 하게 됩니다.

이는 학습에 치명적인 영향을 미치게 됩니다.

Issue with Delta Clipping

그래서 다음과 같이 Huber loss를 사용해줘야 합니다.

```
def clipped_error(x):
  """Huber loss"""
  try:
    return tf.select(tf.abs(x) < 1.0, 0.5 * tf.square(x), tf.abs(x) - 0.5)
  except:
    return tf.where(tf.abs(x) < 1.0, 0.5 * tf.square(x), tf.abs(x) - 0.5)
```



(Image credit: Wikipedia)

Huber loss (green, $\delta = 1$) and squared error loss (blue)

Compute Loss and Gradient (dqn/agent.py)

```
def build():
  # Build network
  ...

  # Compute loss and gradient
  action_one_hot = tf.one_hot(current_action, self.action_space, 1.0, 0.0)
  q_acted = tf.reduce_sum(source_q * action_one_hot, reduction_indices=1)
  max_target_q = tf.reduce_max(target_q, axis=1)
  delta = (1 - done) * self.config.gamma * max_target_q + current_reward - q_acted
  loss = tf.reduce_mean(clipped_error(delta))
  train_op = tf.train.RMSPropOptimizer(lr, momentum=0.95, epsilon=0.1).minimize(loss)

  # Update target network
  ...
```

RMSPropOptimizer를 이용하여 `train_op`을 만들어 줍니다.

Update Target Q Network (dqn/agent.py)

```python
def build():
  # Build network
  ...

  # Compute loss and gradient
  ...

  # Update target network
  target_update_op = []
  source_variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope='source')
  target_variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope='target')
  for source_variable, target_variable in zip(source_variables, target_variables):
    target_update_op.append(target_variable.assign(source_variable.value()))
  target_update_op = tf.group(*target_update_op)
```

주기적으로 target network를 업데이트 해주기 위해, source network의 파라미터를 target network에 할당하는 `target_update_op`을 만들어 줍니다.

Train and Run Agent (dqn/agent.py)

```
def train(new_state, reward, done):
  # Update history
  self.history.add(new_state)

  # Predict action using epsilon-greedy policy
  if random.random() < epsilon_greedy():
    action = random.randrange(action_space)
  else:
    action = sess.run(inference_action_op, {self.state: self.history.get()})

  # Update replay memory
  self.replay_memory.add(new_state, reward, action, done)

  # Train source network
  s, a, r, n_s, done = self.replay_memory.sample()
  sess.run(self.train_op,
           {self.state: s,
            self.action: a,
            self.reward: r,
            self.next_state: n_s,
            self.done: done})

  # Periodically update target network
  if update_target:
    sess.run(self.target_update_op)
```

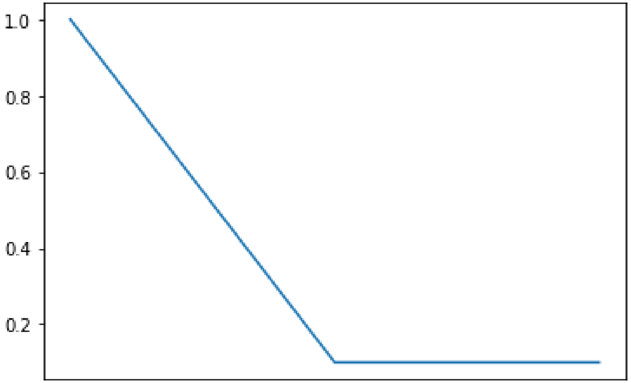
Better Exploration (dqn/agent.py)

```
def train(new_state, reward, done):
  ...

  # Predict action using epsilon-greedy policy
  if random.random() < epsilon_greedy():
    action = random.randrange(action_space)
  else:
    action = sess.run(inference_action_op, {self.state: self.history.get()})

  ...
```

Exploration을 향상시켜주기 위해, 일정 확률로 랜덤하게 움직이는 ϵ-greedy policy를 적용시켜 줍니다.



Can We Do Better?

Yes!

Can We Do Better?

- **Dobule DQN (DDQN)**
  - DQN uses same values to select and to evaluate an action → Resulting overoptimistic value estimates!
  - Then decouple the selection from the evaluation

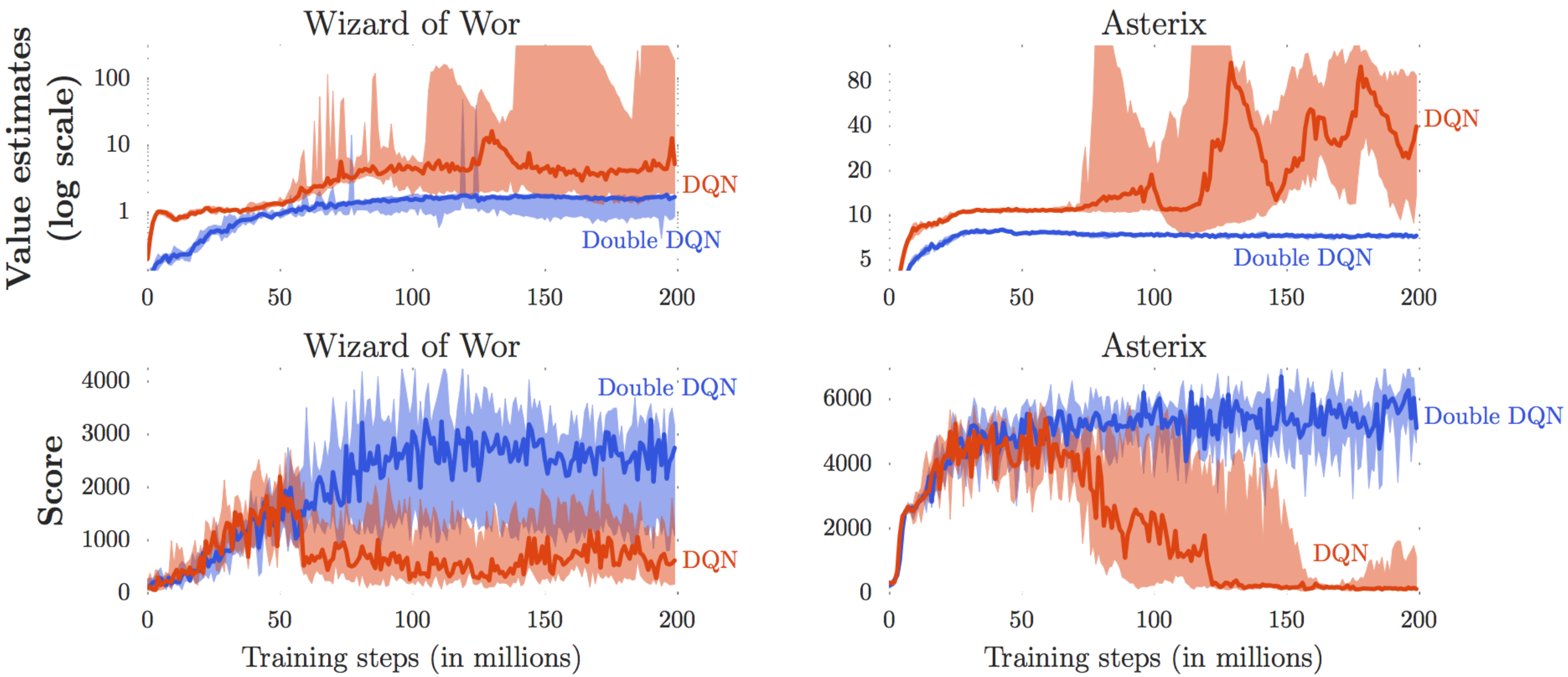
$$y_t^{DQN} = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^-)$$

$$y_t^{DDQN} = R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \theta_t), \theta_t^-)$$

H. Hasselt et al. Deep Reinforcement Learning with Double Q-learning. In *AAAI*, 2016

Can We Do Better?

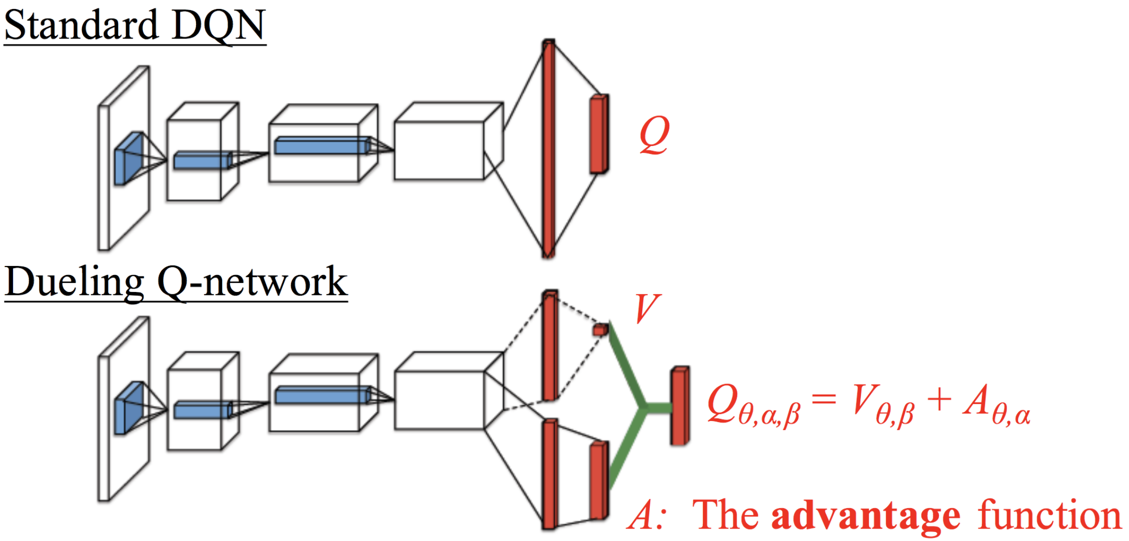
- **Dobule DQN (DDQN)**



H. Hasselt et al. Deep Reinforcement Learning with Double Q-learning. In *AAAI*, 2016

Can We Do Better?

- **Deuling Q-Network**
  - Seperates the representation of state values $\hat{V}(S)$ and action advantages $\hat{A}(S)$


Standard DQN
Q
Dueling Q-network
V
$Q_{\theta,\alpha,\beta} = V_{\theta,\beta} + A_{\theta,\alpha}$
A: The **advantage** function

Z. Wang et al. Dueling Network Architectures for Deep Reinforcement Learning. In *ICML*, 2016

Can We Do Better?

- **Prioritized Experience Replay**
  - **Key idea**
    - Not all transitions are useful
    - Then, RL can learn more effectively from some transitions than others!
  - **Approach**
    - Sampling transitions with high **Temporal-Difference error** δ_t

$$\delta_t = R_t + \gamma_t \max_a Q_{target}(S_t, a) - Q(S_{t-1}, A_{t-1})$$

T. Schaul et al. Prioritized Experience Replay. In *ICML*, 2016

Useful Tips for Designing Your Own RL Agent

(Slide credit: J. Schulman's Talk)

New Algorithm? Use Small Test Problems

- Run experiments quickly
- Do hyperparameter search
- Interpret and visualize learning process: state visitation, value function, etc.
- Useful to have medium-sized problems that you're intimately familira with (Hopper, Atari Pong)

New Task? Make It Easier Until Signs of Life

- Provide good input features
- Shape reward function

Run Your Baselines

- Don't expect them to work with default parameters
- Recommended (rllab, OpenAI lab, keras-rl) :
  - Cross-entropy method
  - Well-tuned policy gradient method
  - Well-tuned Q-learning + SARSA method

Run with More Samples Than Expected

- Early in tuning process, may need huge number of samples
    - Don't be deterred by published work
- Examples:
    - DQN on Atari: update freq=10K, replay buffer size=1M

It Works! But Don't Be Satisfied

- Explore sensitivity to each parameter
  - If too sensitive, it doesn't really work, you just got lucky
- Look for health
  - VF fit quality
  - Policy entropy
  - Standard diagnostics of deep networks

General RL Diagnostics

- Look at min / max /stdev of episode returns, along with mean
- Look at episode lengths: sometimes provides additional information
    - Solving problem faster, losing game slower

Always Whitening / Standardizing Data

- If observations have unknown range, standardize
  - Compute running estimate of mean and standard deviation
  - $x' = clip((x - \mu)/\sigma, -10, 10)$
- Rescale the rewards, but don't shift mean, as that affects agent's will to live
- Standardize prediction targets (e.g. value functions) the same way

Generally Important Parameters

- Discount
    - $Return_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$
    - Effective time horizon: $1 + \gamma + \gamma^2 + \ldots = 1/(1-\gamma)$
        - i.e. $\gamma = 0.99 \rightarrow$ ignore rewards delayed by more than 100 timesteps
    - Low γ works well for well-shaped reward
- Action frequency
    - Solvable with human control (if possible)

Q-Learning Strategies

- Optimize memory usage carefully: you'll need it for replay buffer
- Learning rate schedules
- Exploration schedules
- Be patient. DQN converges slowly
  - On Atari, often 10-40M frames to get policy much better than random

Thank You!

@bckim92

Special thanks to: Jongwook Choi

Slideshow created using remark.