강화학습으로 풀어보는 슈퍼마리오 part 2.*

오늘은 네이쳐(Nature)지의 표지를 장식한 딥마인드(Deepmind)의 Deep Q-Network 알고리듬과 텐서플로우(Tensorflow)로 구현된 OpenAI baselines 코드[1]를 보면서 딥마인드의 DQN(Deep Q-Network) 알고리듬이 학습하는 과정을 순서대로 하나 하나 리뷰해보려 한다.

사람마다 공부하는 성향이 다르다. 기본기부터 하나씩 쌓으며 조심스럽게 응용기술을 공부하는 사람이 있고, 응용기술 구현으로 시작해 기본기를 채워 나가는 스타일이 있다. 필자는 후자다. 동작하는 프로그램을 만들고 개선을 위해 해당 구성 기술 요소에 대해 하나씩 이해해 나가기 시작한다. 필자의 첫 번째 튜토리얼을 따라했다면(카카오 AI 리포트 7호 참고)[2], 응용 기술부터 시작한 것이다. 그렇다면, 우리는 일단 동작하는 강화학습 프로그램을 돌려본 적이 있는 셈이다.

자, 이제 핵심 기술을 이해를 할 시간이다. 각 구성 요소와 프로그램이 동작하는 원리를 이해하고, 무엇을 개선해야 더 나은 학습이 되는지 이해해 보도록 하자.

글 | 송호연 chris.song@kakaocorp.com

강화학습과 썸을 타다가 최근에 연애를 시작했습니다. 자기 전에도, 아침에 일어났을 때도 강화학습 생각이 납니다. 앞으로 세상을 파괴적으로 혁신시킬 범용인공지능(Artificial General Intelligence)에 완전히 빠져 있습니다. 카카오에서는 대규모 데이터 유저 프로파일링, 머신러닝 기술을 활용해 현실의 문제를 해결하는 데이터 엔지니어로 일하고 있습니다.

***감사 인사**

이웅원(RLCode 리더), 유승일(Google Senior Software Engineer), 민규식(한양대학교 박사 과정)
본문 튜토리얼의 내용에 대한 감수와 조언에 깊은 감사의 말씀을 전하고 싶습니다.

이번 튜토리얼의 목적은 간결하지만 도전적이다. 바로 DQN(Deep Q-Network) 알고리듬을 공식적으로 세상에 알리고 네이쳐지의 표지를 장식한 딥마인드의 'Playing Atari game with deep reinforcement learning' 논문을 이해하는 것이 첫 번째 목표이다.

두 번째 목표는 논문에 적힌 알고리듬이 실제로 구현된 OpenAI의 baselines 오픈소스에 있는 DQN 코드를 읽고 하나씩 이해하는 것이다. 전설적인 과학자 리차드 파인만(Richard Feynman)이 말한 명언과 함께 공부를 시작해보자. 논문을 읽은 것 만으로는 이해했다고 할 수 없다. 논문의 내용과 구현체를 같이 살펴보면서 실제로 어떻게 DQN(Deep Q-Network)이 작동하게 되는지 이해해 보는 것이다.

> "내가 만들지 못하는 것은,
> 내가 아직 이해하지 못한 것이다."
>
> - 리차드 파인만(Richard Feynman)

Deep Q-Network 알고리듬의 특징

DQN은 총 세가지 특징으로 설명할 수 있다. 첫 번째 특징은 타겟 Q함수다. 두 번째 특징은 게임 플레이 기록을 저장하는 리플레이 메모리(replay memory, replay buffer)다. 그리고 세번째 특징은 가장 중요한 두뇌의 역할을 하는 Q함수(Q function)를 인공신경망(artificial neural network, ANN)으로 만들었다는 점이다. 그리고 이 알고리듬을 실제로 동작하게 만드는 시뮬레이션 환경이 필요하다.

1) 강화학습을 위한 시뮬레이션 환경

딥마인드의 논문에서 사용되었던 환경은 아타리(atari) 게임들이다. 우리가 사용할 게임은 바로 슈퍼마리오다. 우리가 강화학습에서 다루어보는 슈퍼마리오 학습환경은 OpenAI에서 개발한 gym과 게임 에뮬레이터인 fceux를 활용한다.

OpenAI의 gym은 생각보다 아주 심플하고 단순한 함수들로 이루어져 있다. 가장 중요한 3개의 함수를 소개하겠다.

(1) reset() 함수

reset() 함수는 게임 환경을 초기화하는 일을 한다. 주로 맨 처음 게임을 시작할 때, 그리고 에이전트가 죽어서 게임이 끝났을 때 게임을 초기화하는 역할을 담당한다. reset() 함수는 게임 환경을 초기화하면서 observation 변수를 같이 반환한다.

(2) step() 함수

step() 함수는 에이전트에게 명령을 내리는 함수이다. 사실상, 가장 많이 호출하게 되는 함수이다. 이 함수로 action 명령을 보내고, 환경에서 observation 변수, 보상(reward), 게임 종료 여부 등 변수를 반환한다.

(3) render() 함수

render() 함수는 화면에 해당 상태를 표시하는 역할을 한다. Gym 환경에서는 render() 함수로 현재 게임 상태를 화면에 출력할 수 있다. 조금 헷갈릴 수 있는데, 슈퍼마리오에서는 이 함수를 호출하지 않더라도 화면이 표시된다. 그러니 슈퍼마리오 환경에서는 이 함수를 신경쓰지 않아도 된다.

2) 타겟 Q함수(Q-function)

타겟 Q함수는 학습의 대상이 되는 Q함수가 학습이 되면서 계속 바뀌는 문제를 해결하기 위해 딥마인드에서 제시한 해법이다. 타겟 Q함수가 추가로 생기면서, DQN의 Q 함수는 총 두 개가 된다. 이 기법은 우선 학습을 시키는 기준으로 타겟 Q함수를 사용하며, 실제 그라디언트 디센트(gradient descent)로 Q 함수를 학습시킨다. 일정 스텝이 될 때마다 Q함수의 가중치(weights)들을 타겟 Q함수에 업데이트한다.

3) 리플레이 메모리(Replay Memory)

리플레이 메모리는 강화학습에서 학습의 재료가 되는 중요한 샘플(sample)을 저장해두는 저장소다. 딥마인드에서 제안한 DQN 논문 'Playing Atari with deep reinforcement learning' 에서는 샘플들 간의 상관관계(correlation)으로 인해 학습 속도가 느려지는 문제를 해결하기 위하여 Long-Ji Lin이 'Reinforcement learning for robots using neural networks'[3] 에서 제안한 experienced replay를 이용한 학습법을 사용했다. 즉 게임을 플레이하면서 모든 과정이 들어오는 즉시 훈련시키지 않고, 일단 메모리에 저장해뒀다가 나중에 일정 수의 샘플을 랜덤으로 꺼내서 학습을 시키는 방식을 사용하였다. 플레이 기록을 저장해두고 미니배치(minibatch)를 활용해 학습시킨다.

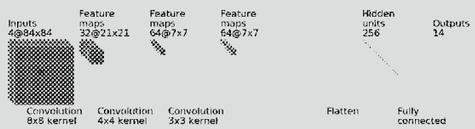
우리는 gym agent에서 매번 명령을 내리면서, state(observation), reward, done 세 가지 정보를 받아올 것이다. 달리 표현하면, 한 번 명령을 내릴 때 SARS(S : state, A : action, R : reward, S' : next state) 이 네 가지 정보를 확보할 수 있다는 것이다.

4) CNN과 MLP을 활용한 Q함수

딥마인드 논문에서 딥러닝 기술이 발전하면서 딥러닝과 강화학습을 접목시키는 아이디어가 조명받고 있다고 했다. DQN은 인공신경망으로 정책 함수(policy function)를 근사(approximate)했다. 인공신경망의 아키텍쳐는 튜토리얼 1편(카카오 AI 리포트 7호 참고)[4]에서 설명했듯이

CNN(convolutional neural network)과 MLP(multi layer perceptron)로 이루어져 있다.

[그림 1] 오픈소스를 활용해 작성한 Q 네트워크의 구조



조금 더 깊은 이해를 위해, 우리가 만들 DQN의 Q 함수 특징을 한번 설명해 보겠다.

(1) 모델 프리(model-free)

근사함수를 활용한 방식은 에이전트가 행동한 샘플을 가지고 직접적으로 정책을 근사(approximate)시킨다. 이렇게 함으로써, 우리는 다이나믹 프로그래밍(dynamic programming)에서 겪었던 차원의 저주에서 벗어날 수 있게 되었다. 강화학습을 차원의 저주에서 벗어나게 해준 대표적인 기법은 인공신경망(artificial neural network, ANN)을 활용한 근사(approximation)다. 최근 딥러닝 시대에서는 대부분 인공신경망을 활용해 정책 네트워크를 훈련시키고 있다.

[그림 2] 차원의 저주[*5]

Tip. Model-based 강화학습과 Model-free 강화학습이란?

딥마인드의 알파고(AlphaGo) 팀 리더이자 강화학습의 전설인 데이비드 실버(David Silver)의 정의로 model-based 강화학습과 model-free 강화학습을 설명해 보겠다.[*6]

Model-free 강화학습:

· 단어 그대로, 모델(model)이 없다.

· 샘플로부터 직접적으로 가치 함수(value function) 혹은 정책 함수(policy function)를 훈련시킨다.

Model-based 강화학습:

· 샘플로부터 모델을 학습시킨다.

· 모델로부터 가치 함수(value function) 혹은 정책 함수(policy function)를 계획(plan)한다.

· 장점 : 지도 학습(supervised learning)을 통해 효과적으로 모델을 훈련시킬 수 있다. 모델의 불확실성에 대해 논할 수 있다.

· 단점 : 먼저 모델을 학습시킨 후, 모델로부터 가치 함수(value function)를 만들어낸다. → 추정의 에러(approximation error)가 모델과 가치 함수 두 영역에서 동시에 생긴다.

(2) 오프폴리시(Off-policy)

DQN은 오프폴리시 방식을 사용했다. 오프폴리시는 단순하게 말하면, 다른 에이전트가 하는 행동을 보고 배운다는 뜻이다. DQN 알고리듬은 자신이 플레이했던 기록을 리플레이 메모리에 넣어두고 랜덤으로 샘플링하여 학습을 진행한다. 이렇게 리플레이 메모리를 쓰는 경우, DQN 에이전트는 과거의 정책이 만들어냈던 샘플들로부터 배우게 된다. 즉, DQN 자신의 정책과 자신이 학습을 하는 기준이 되는 샘플을 만들어낸 정책이 다르게 된다.

Tip. 온폴리시(on-policy)와 오프폴리시(off-policy)는 어떤 차이일까?

딥마인드의 데이비드 실버의 표현에 따르면, 온폴리시와 오프폴리시는 이렇게 표현할 수 있다.[*7]

온폴리시(on-policy):

· 자신이 하는 행동으로부터 배운다

· 자신의 정책 π 로부터 나온 sample들로 자신의 정책 π 를 학습시킨다.

오프폴리시(off-policy):

· 다른 에이전트가 하는 행동을 어깨 넘어 보고 배운다

· 행동 정책(behavior policy) b 로부터 나온 sample들로 자신의 타겟 정책(target policy) π 를 학습시킨다.

조금 더 구체적으로 설명하기 위해 서튼(RS Sutton) 교수님의 'Reinforcement learning : an introduction' 책에 있는 설명으로 온폴리시와 오프폴리시에 대해 부연설명을 해보겠다[*8].

강화학습에서 정책을 정하는 함수를 업데이트 해나가는 방법에는 크게 두 가지 방법이 있다. 하나는 온폴리시 방식, 다른 하나는 오프폴리시 방식이다. 강화학습에서 정책 함수를 업데이트시킬 때 딜레마가 하나 존재한다. 그것은 바로, 탐험(exploration)을 해야 하기 때문에 에이전트가 최적의 경로로 움직이지 않는다는 점이다. 탐험을 하려면 새로운 시도를 해야 하는데, 이는 최적화되지 않은 행동(non-optimal)이다. 이 관점에서 온폴리시와 오프폴리시 방식을 각각 설명해 보겠다.

온폴리시 방식은 모델이 오프폴리시에 비해 단순하고 구현하기 쉬운 반면 사실 약간의 타협을 한 방식이다. 온폴리시 방식은 최적의 정책(optimal policy)을 보고 배운다기 보다는 최적에 가까운 정책(near-optimal policy)을 보고 배운다. 온폴리시 방식은 단 하나의 정책으로 행동을 하는 동시에 같은 정책을 학습시킨다. 즉, 여전히 탐험을 하고 있는 정책으로부터 배운다.

오프폴리시 방식은 이 문제를 해결하기 위해 직접적인 해결책을 제시한다. 바로 타겟 정책과 행동 정책을 나누는 것이다. 타겟 정책은 우리가 강화학습 에이전트에게 가르치기 위한 기준이 되는 정책이다. 그리고 행동 정책은 탐험을 하며 새로운 행동을 만들어내는 정책이다. 이 경우에 우리는 에이전트를 타겟 정책으로부터 떨어진(off) 데이터로 부터 학습을 시키며, 이 전체적인 프로세스를 오프폴리시 학습(off-policy learning)이라고 한다. 일반적으로 오프폴리시 방식이 좀 더 강력하고 범용적인 성능을 보인다. 다만, 샘플링을 해야 하며 두가지 폴리시를 다뤄야 하기에 구현하기는 좀 더 어렵다.

(3) 미니배치(Minibatch)

미니배치란 많은 데이터 중 임의로 샘플을 뽑아 학습시키는 것을 의미한다. DQN은 리플레이 메모리와 미니배치 기법을 활용하여 연속적인 샘플들(samples) 간의 강한 상관관계를 제거하고자 하였다.

(4) 가치 기반(Value-Based) 강화학습

DQN은 대표적인 가치 기반(value-based) 강화학습으로서 가치 함수(value function)를 먼저 근사(approximate)시켜서 정책을 만들어내는 방식으로 학습을 시킨다.

(5) 감소하는 입실론 그리디(Decaying Epsilon-Greedy)

탐험과 활용(exploration & exploitation)은 강화학습이라는 시스템에서 해결해야 하는 아주 중요한 문제 중 하나다. DQN 강화학습 에이전트는 처음에는 랜덤으로 움직인다. 그러다가 어느 정도 학습을 하면 Q 함수 혹은 정책 네트워크가 알려주는 행동(action)을 취한다. 예를 들어, DQN에서 강화학습 에이전트는 처음에는 99%의 확률로 랜덤 행동을 취하도록 되어 있고, 특정 시기에 도달하기까지 랜덤 액션을 취하는 확률 ε(입실론)은 지속적으로 줄어든다. 그리고 ε이 1%가 되면 감소를 멈추고 1%로 고정되게 된다. 이러한 방법론을 바로 '감소하는 입실론 그리디(decaying epsilon-greedy) 방식'이라고 한다. 감소하는 입실론 그리디(decaying epsilon-greedy)에 나오는 두 가지 하이퍼 파라미터는 아래와 같다.

exploration_fraction: 언제까지 감소시킬 것인가? 0.5가 기본값으로 설정되어 있으며, 이 경우에는 총 timesteps의 50%가 될 때까지 랜덤액션을 취할 확률 ε이 계속 줄어들게 된다.

exploration_final_eps: 최종 ε값이다. 기본 값은 0.01로 설정되어 있다. 이 의미는 ε 이 계속 줄어 들다가 0.01이 되면 감소를 멈추고 ε 값이 0.01로 유지되는 것이다. 훈련이 완료되더라도 1%의 랜덤액션을 계속 취하게 되는 셈이다.

Tip. 연속적인 샘플들(samples)간의 강한 상관관계를 줄인다?

"연속적인 샘플들 간의 강한 상관관계를 줄인다"라는 말이 어려울 수 있을 것이다. 좀 더 쉽게 설명을 해보겠다. 게임을 할 때마다 다른 패턴을 발견하게 된다. 예를 들기 위해 문제를 아주 쉽게 제한해 보겠다. 플레이 방식에 따라 10가지 플레이 패턴이 나온다고 하자. 그 중에 7번째 패턴으로 플레이하면 게임 점수가 100점으로 가장 좋다. 그런데 1번째 패턴으로 플레이하면 성능이 50점 정도가 나온다. 그리고 나머지 8개 패턴은 다 10점 이하의 점수를 내놓는다고 치자.

강화학습을 진행할 때 리플레이 메모리가 없이 들어오는 순서대로 바로 학습을 시키는 상황을 가정하겠다. 에이전트가 1번 패턴에서부터 학습을 하게 된다고 하면, 에이전트의 머릿 속에는 금방 1번 패턴만 가득차게 될 것이다. 그러면서 자연스럽게 Q 함수는 1번 패턴이 최고라고 익히게 되고, 결국 에이전트는 1번 패턴의 지역최적해(local minima)에 갇히게 된다.

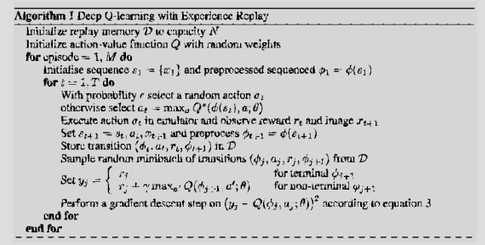
이러한 문제는 Long-ji Lin의 'Reinforcement learning for robots using neural networks'에서 다루어진 바가 있고, 이를 해결하기 위해 경험 재생(experience replay)라는 솔루션이 제시되었다.

DQN에서는 이런 연속적인 기록의 강한 상관관계를 미니배치와 리플레이 메모리를 통해 해결했고, 향후에 나오게 되는 Actor-Critic 유형의 알고리듬은 멀티쓰레딩(multi-threading)[*9] 환경에서 다양한 에이전트를 동시에 띄워서 다양한 샘플들(samples)을 동시에 얻게 만들었고 이를 통해 샘플들 간의 강한 상관관계를 줄였다.

Deep Q-Network 알고리듬 구현체 설명

딥마인드 논문에 적혀있는 DQN의 알고리듬을 살펴보겠다. 그리고 알고리듬의 각 라인에 해당하는 OpenAI의 DQN 구현체 코드를 확인하며 구현체를 이해해 보도록 하겠다.

[그림 3] Open AI의 DQN 구현체 코드



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 ϵ 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

Line 1) Initialize replay memory D to capacity N

리플레이 메모리 D를 N의 크기로 초기화한다. mario-rl-tutorial 프로젝트의 해당 위치로 이동해보자. deepq/deepq.py(폴더명/파일명) 파일을 열고 198라인으로 이동한다.

```
replay_buffer = ReplayBuffer(buffer_size)
beta_schedule = None
```

deepq/deepq.py 198라인부터 199라인 두 줄이 바로 리플레이 메모리를 생성하는 곳이다.

Line 2) Initialize action-value function Q with random weights

Q 함수를 초기화하는 부분으로, 행동 가치 함수(action-value function) Q를 임의의 가중치(weight)로 초기화한다. 우리는 CNN과 fully connected layer를 조합해서 Q 함수를 구현한다. 텐서플로우 명령어로 모델의 구조를 설정한다.

슈퍼마리오 튜토리얼 1편[*10]에서 우리는 Q 함수의 구성을 정의한 적이 있다. 여기에서 우리는 매개 변수 만을 설정했다.

```
# 4. Create a CNN model for Q-Function
model = cnn_to_mlp(
 convs=[(32, 8, 4), (64, 4, 2), (64, 3, 1)],
 hiddens=[256],
 dueling=FLAGS.dueling
)
```

위 코드는 3개의 CNN(convolutional neural network)의 구조를 설정하고, CNN의 결과값을 fully connected layer에 연결하는 형태로 Q 함수를 구성한다. 우리가 원하는 모델의 매개변수를 받아서 진짜로 구현하는 구현체를 살펴보도록 하자.

deepq/models.py 파일의 33라인부터 90라인까지 전체가 Q 함수 설계를 정의하는 스크립트다. 지면 상 코드를 다 실을 수 없어 링크와 QR코드로 대체한다. deepq/models.py 링크: https://goo.gl/E53fM3

[그림 4] Q 함수 설계를 정의하는 스크립트

여기에서 cnn_to_mlp() 라는 함수는 람다(lambda) 기법으로 구현이 되어 있다. cnn_to_mlp 함수는 convs, hiddens, dueling 파라미터를 사용해 CNN과 MLP의 구조를 설정한다. 하지만, cnn_to_mlp 함수는 실제 텐서플로우 그래프를 만들어서 반환하지는 않는다. cnn_to_mlp() 함수는 실제 텐서플로우 레이어들을 반환하지 않고 람다 함수를 반환한다. 람다 함수는 CNN과 MLP 구조에 대한 값만 받은 상태이고 아직 inpt, num_actions, scope, reuse, layer_norm 같은 값들을 받지 못한 상태다. 이 람다 함수는 향후에 build_graph.py 에서 inpt, num_actions, scope, reuse, layer_norm 값들과 함께 호출되어 실제 텐서플로우 그래프를 생성한다.

Tip. Python lambda 문법을 활용하여 함수의 일부만 구현하기

Python에서는 람다 문법을 활용해 좀 더 자유로운 표현을 할 수 있다. 람다 기법에 익숙하지 않은 분들은 cnn_to_mlp() 함수를 보면 이해하는 데 어려움이 있을 것 같다. 최소한의 예시로 이해를 돕도록 해보겠다. 예시 코드는 mario-rl-tutorial Github에 lambda_ex1.py 라는 파일에 구현해두었다.

t2() 함수는 람다 함수를 반환한다. t2() 함수는 1이라는 값을 받아서 a라는 매개변수에 값을 설정한 상태로 람다 함수를 반환한다. t2(1) 함수의 반환값을 lambda_fun 이라는 변수에 할당한다. lambda_fun 변수는 특정한 스칼라 값이 아니라 람다 함수가 된다. 그래서 우리는 다시 lambda_fun 함수를 호출할 수 있다. 이 때, 우리는 t2() 에서 설정하지 않았던 매개변수 b와 c 값을 설정하여 호출한다.

```
def t(a,b,c):
 print(a,b,c)
```

~

```
def t2(n):
 return lambda *args, **kwargs: t(a=n, *args, **kwargs)

lambda_fun = t2(1)
lambda_fun(b=2, c=3)
# 1 2 3
```

cnn_to_mlp() 함수에 CNN의 구조와 MLP의 구조를 매개변수로 전달하면, q_func 람다 함수가 반환된다. 하지만, 아직 모델 생성이 끝나지 않았다. 힘들더라도 조금만 더 힘내기 바란다.

deepq/deepq.py 파일의 168라인부터 187라인까지는 우리가 만든 q_func 람다 함수와 여러 매개변수를 활용해 act, train, update_target, debug 함수를 만든다.

```
# Create all the functions necessary to train the model

sess = U.make_session(num_cpu=num_cpu)
sess.__enter__()

def make_obs_ph(name):
 return U.BatchInput((84,84,4), name=name)

act, train, update_target, debug = build_graph.build_train(
 make_obs_ph=make_obs_ph,
 q_func=q_func,
 num_actions=env.action_space.n,
 optimizer=tf.train.AdamOptimizer(learning_rate=lr),
 gamma=gamma,
 grad_norm_clipping=10,
 param_noise=param_noise
)
act_params = {
 'make_obs_ph': make_obs_ph,
 'q_func': q_func,
 'num_actions': env.action_space.n,
}
```

Line 3) for episode = 1, M do

에피소드를 1부터 M까지 반복한다. 여기서 에피소드란 게임 한 판을 의미한다. 우리가 실행하고 있는 mario-rl-tutorial 예제에서는 M 대신 max_timesteps 라는 변수로 최대 실행 횟수를 설정해 두었다. deepq/deepq.py 224라인에 해당 반복문이 설정되어 있다.

```
for t in range(max_timesteps):
```

이 train.py 훈련 스크립트를 실행할 때 max_timesteps를 설정할 수 있게 구현했다. 예를 들어 1000스텝만 실행하고 싶다면 이렇게 훈련 스크립트를 실행하면 된다.


```
python train.py --timesteps=1000
```

Line 4) Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\varphi_1 = \varphi(s_1)$

가장 처음의 상태(state, s_1)를 설정한 후 전처리(preprocess)한다.

```
reshape_obs = np.reshape([obs], (1, 84, 84, 1))
history = np.append(reshape_obs, history[:, :, :, :3], axis=3)
processed_obs = np.reshape([history], (84, 84, 4))
```

Line 5) for t = 1, T do

t가 1부터 T가 될때까지 반복한다. 여기서 T는 게임의 한 에피소드가 끝나는 시점을 의미한다. 즉, 게임이 끝날 때까지 반복한다는 뜻이다. 우리가 실행해본 mario-rl-tutorial 구현체는 한 에피소드를 위한 반복문을 따로 설정하지는 않았다. 대신, done 변수의 결과에 따라 에피소드가 종료될 때 환경을 reset하는 등의 처리를 해주고 있다.

```
if done:
 obs = env.reset()
 episode_rewards.append(0.0)
 reset = True
 history = np.stack(
  (next_state, next_state, next_state, next_state), axis=2)
 history = np.reshape([history], (1, 84, 84, 4))
else:
 history = next_history
```

그리고 에피소드가 끝날 때마다 tensorboard에 로그를 남긴다.

```
if done and print_freq is not None and len(episode_rewards) % print_freq == 0:
 logger.record_tabular("steps", t)
 logger.record_tabular("episodes", num_episodes)
 logger.record_tabular("reward", epi_reward)
 logger.record_tabular("mean 100 episode reward", mean_100ep_reward)
 logger.record_tabular("% time spent exploring", int(100 * exploration.value(t)))
 logger.dump_tabular()
```

Line 6) With probability ε select a random action a_t

ε의 확률로 랜덤한 액션 a_t를 선택한다.

Line 7) otherwise select $a_t = max_a Q^*(\varphi(s_t), a; \theta)$

그렇지 않으면 $a_t = max_a Q^*(\varphi(s_t), a; \theta)$ 공식으로 다음 a_t를 선택한다. 랜덤한 행동을 취하는 ε은 가장 먼저 LinearSchedule 이라는 클래스로 정의되고, 위치는 deepq/deepq.py 파일의 201라인이다.

```
# Create the schedule for exploration starting from 1.
exploration = LinearSchedule(schedule_timesteps=int(exploration_fraction *
                max_timesteps),
                 initial_p=1.0,
                 final_p=exploration_final_eps)
```

그리고 t 스텝에서의 ε 값을 조회하는 코드는 231라인에 있다.

```
update_eps = exploration.value(t)
update_param_noise_threshold = 0.
```

그리고, 247라인에서 우리가 만들어놨던 action() 함수에 현재 상태값을 넣으면서 ε 값을 같이 넣어준다.

```
action = act(np.array(processed_obs)[None], update_eps=update_eps, **kwargs)[0]
```

이렇게 act() 함수에 전처리된 상태값과 ε 값을 같이 넣어주면, ε 값에 따라 랜덤한 행동이 반환되거나 e-greedy 방식에 따른 최적의 행동이 반환된다. 여기에서 사용된 act() 함수가 어떻게 구현되어 있는지 build_graph.py 안의 정의를 확인해보자.

build_graph.py의 build_act() 함수

```
def build_act(make_obs_ph, q_func, num_actions, scope="deepq", reuse=None):
 with tf.variable_scope(scope, reuse=reuse):
  observations_ph = U.ensure_tf_input(make_obs_ph("observation"))
  stochastic_ph = tf.placeholder(tf.bool, (), name="stochastic")
  update_eps_ph = tf.placeholder(tf.float32, (), name="update_eps")

  eps = tf.get_variable("eps", (), initializer=tf.constant_initializer(0))

  q_values = q_func(observations_ph.get(), num_actions, scope="q_func")
  deterministic_actions = tf.argmax(q_values, axis=1)

  batch_size = tf.shape(observations_ph.get())[0]
  random_actions = tf.random_uniform(tf.stack([batch_size]), minval=0,
maxval=num_actions, dtype=tf.int64)
  chose_random = tf.random_uniform(tf.stack([batch_size]), minval=0, maxval=1,
dtype=tf.float32) < eps
  stochastic_actions = tf.where(chose_random, random_actions, deterministic_
actions)

  output_actions = tf.cond(stochastic_ph, lambda: stochastic_actions, lambda:
deterministic_actions)
  update_eps_expr = eps.assign(tf.cond(update_eps_ph >= 0, lambda: update_
eps_ph, lambda: eps))
  act = U.function(inputs=[observations_ph, stochastic_ph, update_eps_ph],
           outputs=output_actions,
           givens={update_eps_ph: -1.0, stochastic_ph: True},
           updates=[update_eps_expr])
  return act
```

우리가 build_act() 함수를 실행해서 얻는 act() 함수는 U.function()

이라는 함수를 활용해 만들어낸다. U.function()은 원래 텐서플로우에서 지원하는 함수는 아니고 OpenAI에서 텐서플로우 연산을 단순한 함수 형식으로 변환한 것으로, 필요할 때 함수만 호출하면 텐서플로우 연산을 돌릴 수 있도록 만든 모듈이다.

여기에 정의된 act()함수는 ε 값과 4 프레임의 상태값을 받아서 다음에 취할 행동(action)을 반환하는 함수다. U.function을 활용해 입력 값을 넣으면 행동을 반환해주는 함수를 만들어 낸 디자인 패턴이 필자는 너무 마음에 든다. 텐서플로우의 복잡한 연산을 거치지 않고 심플하게 함수로만 호출해서 처리할 수 있게 만들다니 개발자로서 그저 대단하다는 생각만 든다.

Line 8) Execute action a_t in emulator and observe reward r_t and image x_{t+1}

에뮬레이터에서 다음 행동 a_t를 실행하고 다음 보상 r_t와 다음 상태 x_{t+1}을 받는다. deepq/deepq.py 파일에서 249 라인에 있다. 환경에서 action을 실행한 후에 새로운 state(new_obs)와 reward(rew), done을 반환 받는다.

```
new_obs, rew, done, _ = env.step(action)
```

Line 9) Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\varphi_{t+1} = \varphi(s_{t+1})$

$s_{t+1} = s_t, a_t, x_{t+1}$을 세팅하고 $\varphi_{t+1} = \varphi(s_{t+1})$을 전처리한다. deepq/deepq.py 파일의 251라인부터 255라인까지이다. 새로운 상태를 4프레임 stack에 쌓아 넣는다.

```
obs = new_obs

next_state = np.reshape([new_obs], (1, 84, 84, 1))
next_history = np.append(next_state, history[:, :, :, :3], axis=3)
processed_new_obs = np.reshape([history], (84, 84, 4))
```

먼 길을 달려왔다. [그림 3]의 DQN 알고리듬을 다시 한 번 확인해 보기를 바란다.

Line 10) Store transition (φ_t, a_t, r_t, φ_{t+1}) in D

(φ_t, a_t, r_t, φ_{t+1}) 기록(trajectory)를 리플레이 메모리 D에 넣는다.

```
# Store transition in the replay buffer.
replay_buffer.add(processed_obs, action, rew, processed_new_obs, float(done))
```

Line 11) Sample random minibatch of transitions (φ_j, a_j, r_j, φ_{j+1}) from D

리플레이 메모리 D에서 랜덤으로 미니배치 기록(minibatch trajectories)인 φ_j, a_j, r_j, φ_{j+1}를 추출한다. deepq/deepq.py 파일에서 278,279 라인에 있다. replay_buffer.sample(batch_size)가 리플레이 메모리에서 batch_size 만큼의 기록을 미니배치로 추출하는 함수다.

```
obses_t, actions, rewards, obses_tp1, dones = replay_buffer.sample(batch_size)
weights, batch_idxes = np.ones_like(rewards), None
```

Line 12) Set $y_j = r_j$ (for terminal φ_{j+1}) $y_j = r_j + \gamma \max_{a'} Q(\varphi_{j+1}, a';\theta)$ (for non-terminal φ_{j+1})

$y_j = r_j$ (게임이 끝인 경우 φ_{j+1}) / $y_j = r_j + \gamma \max_{a'} Q(\varphi_{j+1}, a';\theta)$ (게임이 아직 안끝난 경우 φ_{j+1})

Line 13) Perform a gradient descent step on $(y_j - Q(\varphi_j, a_j;\theta))^2$ according to equation 3

공식 3에 따라 $(y_j - Q(\varphi_j, a_j;\theta))^2$ 그라디언트 디센트를 실행한다.

```
td_errors = train(obses_t, actions, rewards, obses_tp1, dones, weights)
```

여기에서 호출된 train() 함수가 구현된 build_graph.py의 build_train() 함수로 다시 가보겠다.

지면 상 코드를 다 실을 수 없어 링크와 QR코드로 대체하겠다.
deepq/build_graph.py 코드 링크 : https://goo.gl/2Kgi3S

[그림 5] build_graph.py의 build_train()함수

build_graph.build_train() 함수에 여러 가지 매개변수를 넣으면 조건에 맞는 모델을 생성함과 동시에, 상태를 넣으면 행동을 알려주는 act() 함수, 모델을 학습시키는 train() 함수, 타겟 네트워크를 업데이트 시키는 update_target() 함수, 디버깅을 위한 debug 변수를 반환한다. Debug 변수는 트레이닝 중간에 생성되는 q_value 등 중간 산출물을 넣어서 같이 보내기 위한 통로로 쓰인다. build_train() 함수는 Q함수를 업데이트 시키는 로직이 구현되어 있다.
(1) 튜토리얼 초반에 선언했던 q_func 람다 함수에 나머지 값들을 마저 넣어서 Q 네트워크 모델과 타겟 Q 네트워크 모델을 만들어낸다.

```
# q network evaluation
q_t = q_func(obs_t_input.get(), num_actions, scope="q_func", reuse=True) #
reuse parameters from act
```

~

```
q_func_vars = U.scope_vars(U.absolute_scope_name("q_func"))

# target q network evaluation
q_tp1 = q_func(obs_tp1_input.get(), num_actions, scope="target_q_func")
target_q_func_vars = U.scope_vars(U.absolute_scope_name("target_q_func"))
```

(2) 그리고 Q 네트워크를 구성하는 변수들을 조회한 후 q_func_vars 저장한다.

```
q_func_vars = U.scope_vars(U.absolute_scope_name("q_func"))
```

(3) 현재 Q 네트워크와 업데이트의 타겟이 되는 값을 구한다.

```
# q scores for actions which we know were selected in the given state.
q_t_selected = tf.reduce_sum(q_t * tf.one_hot(act_t_ph, num_actions), 1)

q_tp1_best = tf.reduce_max(q_tp1, 1)
q_tp1_best_masked = (1.0 - done_mask_ph) * q_tp1_best

# compute RHS of bellman equation
 q_t_selected_target = rew_t_ph + gamma * q_tp1_best_masked
```

(4) 에러 값을 구한다.

```
 # compute the error (potentially clipped)
 td_error = q_t_selected - tf.stop_gradient(q_t_selected_target)
 errors = U.huber_loss(td_error)
 weighted_error = tf.reduce_mean(importance_weights_ph * errors)
```

(5) 그라디언트 디센트를 수행한다. 여기서 grad_norm_clipping 이라는 값이 있는데, 학습을 시키다 보면 종종 급격한 그라디언트(gradient) 값으로 인해 모델이 망가지는 경우가 있다. 이를 막기 위해 tf.clip_by_norm() 함수를 사용한다. tf.clip_by_norm() 함수는 그라디언트를 정규화한 특정 값을 넘어가면 그라디언트를 잘라준다.

```
# compute optimization op (potentially with gradient clipping)
 if grad_norm_clipping is not None:
  optimize_expr = U.minimize_and_clip(optimizer,
                                      weighted_error,
                                      var_list=q_func_vars,
                                      clip_val=grad_norm_clipping)
 else:
  optimize_expr = optimizer.minimize(weighted_error, var_list=q_func_vars)
```

이렇게 DQN 알고리듬 구현체의 코드 리뷰를 마쳤다. 상당히 길고 험한 과정이었다. 그리고 OpenAI의 baselines 오픈소스 코드의 추상화 레벨이 아주 높기 때문에 초심자의 경우에는 이해하기가 상당히 어려울 것이라 생각한다.

최고의 작가가 되는 첫 걸음은 필사(베껴 쓰기)라고 배웠다. 최고의 개발자가 되기 위한 첫 걸음 역시 우수한 코드를 보고 따라서 작성해보는 것이 아닐까 생각된다. 조금은 벅차더라도, 잘 정돈된 코드를 보고 익히며 소화해 보도록 하자.

이번 튜토리얼에서는 알고리듬의 구현체에 대한 코드레벨 관련 설명을 하였다. 다음 편에서는 Actor-Critic 강화학습 알고리듬을 소개하도록 하겠다.

***1** 참고 | OpenAI, Github baselines : https://github.com/openai/baselines ***2** 참고 | https://brunch.co.kr/@kakao-it/144 ***3** 논문 | Lin, L. (1993). Reinforcement Learning for Robots Using Neural Networks, CMU ***4** 참고 | https://brunch.co.kr/@kakao-it/144 ***5** 참고 | https://haifengl.wordpress.com/2016/02/29/there-is-no-big-data-in-machine-learning/ ***6** 참고 | RL Course by David Silver - Lecture 8: Integrating Learning and Planning. https://youtu.be/ItMutbeOHtc ***7** 참고 | RL Course by David Silver - Lecture 5: Model Free Control. https://youtu.be/0g4j2k_Ggc4 ***8** 참고 | Sutton, R. & Barto, A. (2017). Reinforcement Learning: An Introduction 2nd edition ***9** 설명 | 멀티쓰레딩은 프로세스(현재 실행중인 프로그램)에 두 개 이상의 쓰레드(thread)를 할당하여 하나의 프로세스에서 병렬적으로 여러 개 작업을 처리하는 것을 일컫는다. 예컨대 멀티쓰레딩 환경을 통해 카카오톡에서 채팅을 하면서 동시에 파일을 다운로드 받을 수 있다. ***10** 참고 | https://brunch.co.kr/@kakao-it/144|

참고자료
[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning, NIPS 2013 Workshop, 2013 **[2]** Chris Hoyean Song, Github mario-rl-tutorial : https://github.com/chris-chris/mario-rl-tutorial