<table>
<tr><td rowspan="11">Model
-Free</td><td rowspan="10">Policy Optimization
(최근에 많은 모델 소개)
1.Continous action 가능
Value-based는 net의 output이 action의 수로 산출되지만 policy-based에서는 distribution의 parameter가 산출, 심지어 value-based에서 net의 output으로 나온 것 중 argmax로 action찾는 것도 optimization
2. stochastic policy가 가능해서 partially observable환경에 좋다.</td><td>REINFORCEMENT</td><td colspan="2">$\Delta\theta_t = \alpha\nabla_\theta \log \pi_\theta(s_t, a_t) v_t$</td></tr>
<tr><td rowspan="5">Actor-critic</td><td colspan="2">$\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
$\theta = \theta + \alpha\nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$</td></tr>
<tr><td>A2C
(Synchronous Advantage AC)</td><td>$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a)\ A^{\pi_\theta}(s, a)]$
Actor 기대출력 = Advatage -> var을 AC에 비해 더 줄일 수 있다.</td></tr>
<tr><td>A3C(2016)
(Asynchronous A2C)</td><td>여러 agent 주기적 실행 + 비동기적 공유 네트워크 업데이트</td></tr>
<tr><td>ACER(2017)
(Actor-Critic with Experience Replay)</td><td></td></tr>
<tr><td>ACKTR(2017)
Actor-Critic using Kronecker-factored Trust Region)</td><td></td></tr>
<tr><td>DPG(2014) ★
Model-free
Off-policy
Actor-critic algo.</td><td colspan="2">$\delta_t = R_t + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t)$
$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w Q_w(s_t, a_t)$
$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_a Q_w(s_t, a_t) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}$
(아직 이해를 잘 못함..)
$\nabla_{\theta^\mu} J \approx \mathrm{E}_{s_t \backsim \rho^\beta}[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)}] = \mathrm{E}_{s_t \backsim \rho^\beta}[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s|Q^\mu)|_{s=s_t}]$</td></tr>
<tr><td rowspan="5">DDPG(2016)
Model-free
Off-policy
Actor-critic algo.</td><td colspan="2">DQN + DPG</td></tr>
<tr><td>D4PG</td><td>DDPG + Distribution</td></tr>
<tr><td>MADDPG(2017)</td><td>DDPG + Multi-agent</td></tr>
<tr><td>TD3(2018)</td><td>Value function overestimation 방지위해 DDPG에 몇가지 기법 적용</td></tr>
<tr><td>SAC(2018)</td><td>exploration을 더 잘하기 위해 reward에 policy에 대한 entropy measure 추가</td></tr>
<tr><td>NPG</td><td colspan="2"></td></tr>
<tr><td>TRPO(2015)</td><td colspan="2">PPO의 전신(높은 차원의 action space, 광범위한 control parameter 등 두루 좋은 성능을 나타냄)</td></tr>
<tr><td>PPO(2017)</td><td colspan="2">1. Agent가 환경과 interaction 통해 나온 data sampling + stochastic gradient ascent를 사용하여 “surrogate” objective function optimizing 하는 것을 번갈아 한다.
2. Data sample마다 one gradient update하는 것이 아니라, minibatch update를 통해 multiple epoch를 가능하게 하는 새로운 objective function 소개.
3. 장점 : TRPO의 장점(알고리즘으로 학습하기에 간단하고 일반적이고 더 좋은 sample complexity 가짐)</td></tr>
<tr><td>Q-learning
: 고차원 nn썼을 때 불안정하며 수렴안됨, continuous</td><td>MC-Control
(on policy)</td><td colspan="2">$N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t))$
Bias : low, var : high</td></tr>
</table>

	space에서 악화	SARSA (on policy)	$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma Q(S',A') - Q(S,A)\right)$ Bias : high, var : low		
		Q-learning (off-policy)	$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$		
		Double Q-learning (off-policy) (overestimation)	Q-learning : $Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(s_{t+1}, a; \theta_t); \theta_t)$ Double Q-learning : $Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(s_{t+1}, a; \theta_t); \theta'_t)$		
		DQN(2013)★ Online Approximate continuous state Discrete action Continuous control(X)	Deep Learning + Function Approximation using both Experience Replay and Target Network		
			Rainbow	DDQN(Double DQN)(2015) DQN의 target 수정	$y_i = \mathrm{E}_{s' \sim \varepsilon}[r + \gamma \max Q(s', arg\max Q(s', a; \theta_{i-1}); \theta_{i-1}) \mid s, a]$
				Dueling DQN(2015) DQN의 architecture 수정	1. V, A network 분리 2. 분리한 것을 조건 만족하도록 summation ⇨ DDQN + advanced network
				DDDQN	Double DQN + Dueling DQN Dueling DQN자체가 Q network 안에서 일어나는 거라 다른 알고리즘 쉽게 적용 가능 Dramatic performance X
				PER(2015) (Prioritized Experience Replay)	가장 효율적인 replay memory 사용법
				Multi-step learning	
				Noisy Nets	정책 신경망의 가중치(w,b)에 잡음 부여 -> action random성이 state, t에 따라 자동 적응
			REM	Randomly combining multiple Q-value estimates	
		Deep SARSA	$Loss = (R + \gamma Q(s',a') - Q(s,a))^2$ DQN과 마찬가지로, Q값을 Q-network로 DL 신경망 적용		
		Distributional RL	Q : scalar -> distribution 이에 따라 output of network = value distribution w.r.t action Distribution 이유 : Random 환경인 경우, 동일 action, state에서 reward가 다름		
			C51(2017)	Support 개념 도입 + DQN과 유사한 algorithm	
			QR-DQN(2017) (online)	C51의 한계점 개선(parameter 수, 수렴성 보장)	
			IQN(2018) ★	QR-DQN을 random sampling	
		UVFA	DQN에서 multi goal		
		HER	UVFA처럼 multi goal에 관심, Off policy, sparse reward 정의		

<table>
<tr><td rowspan="7">Model
-Based</td><td rowspan="7">Learn</td><td>Worlds model
(2018)</td><td>뇌의 동작방식과 매우 유사하게 학습할 수 있는 DL방법
Vision -> memory -> controller
DQN : input = 현재, 과거화면 -> 미래 일 모름, Q-learning
Worlds model : 다음 화면 예상, 근거로 행동 a.k.a 진화전략(or 유전전략 : 환경에 fitting한 자손만 계속 번식)
꿈을 꾸는 이유 중 하나가 경험했던 일들을 머릿 속에서 반복적으로 수행함으로써 학습을 좀 더 빠르게한다는 가설과 Worlds model 유사하다
나의 생각 : 인풋이 image로 인코딩되어야 하지만 memory(with RNN)이 미래를 예측하는데 이 net을 활용한다면 충분히 매력적인 알고리즘이라고 생각됨</td></tr>
<tr><td>I2A(2018)
Imagination-Augmented Agents for DRL</td><td>Model-based는 보통 function approximation사용해서 imperfection prediction이다. 즉, robust planning, model-based method가 없다. => imperfection prediction을 “learning to interpret(해석하여 학습)”해서 approximate 환경의 model 사용
규칙만 입력하고 모델의 방정식은 만들지 않는다. = policy set으로만 구성된 model free model만들고, policy set의 경우의 수에 가중치를 둬서 시뮬레이션 돌리는 작업(이유는 DQN의 NN의 아웃풋은 action의 수이므로 discrete하기 때문에 모든 종류의 불확실성을 다 고려할 수가 없다.)(이 문제가 policy-based에는 해결되는데.. 둘의 차이점은..?)</td></tr>
<tr><td>MBMF(2017)
Model-based RL with model-free fine-tunning</td><td>DRL에서 몇몇 표준 benchmark task를 통해 학습된 env. Model을 사용해서 MPC를 탐색하는 기법(Model Predictive Control)
MPC기법이란 : 현재에 미래 몇 step model 예측 -> 임시 plan -> agent 다음 action,
단, 신뢰성이 낮으므로 다다음 action은 선택하지 않음</td></tr>
<tr><td>MBVE</td><td></td></tr>
<tr><td>PLICO
(Probability Inference for learning control
가우시안 프로세스 역할 모델(기본 제공)을 기반으로 정책 검색 방법(필수적으로 정책 방복)을 제안.</td><td>a practical, data-efficient model-based policy search method. Pilco reduces model bias, one of the key problems of model-based reinforcement learning, in a principled way. By learning a probabilistic dynamics model and explicitly incorporating model uncertainty into long-term planning, pilco can cope with very little data and facilitates learning from scratch in only a few trials. Policy evaluation is performed in closed form using state-of-the- art approximate inference. Furthermore, policy gradients are computed analytically for policy improvement. We report unprecedented learning efficiency on challenging and high-dimensional control tasks.</td></tr>
<tr><td rowspan="2">Plan</td><td>Alphazero</td><td>Approximate policy 학습, planning algorithm(MCTS 등) 사용되며 planning의 전문성이 향상</td></tr>
<tr><td>MB-MPO(Model based-Meta policy optimization)</td><td>Learned model에 강한 의존, learned model의 앙상블로 meta policy만듬
메타 학습으로 앙상블의 어느 역할 모델이 정책을 최적화, 편향 완화를 할 것인지</td></tr>
<tr><td>PETs
Trajectory를 이용한 확률적 앙상블</td><td colspan="2">Model-based RL algorithm이 높은 차원 파라미터의 함수 때문에 점근적 성능의 관점에서 model-free보다 안 좋을때가 있음.
역할은 combines uncertainty-aware deep network dynamics models with sampling-based uncertainty propagation.
세 부분을 하나의 기능 알고리즘으로 결합(1. 여러 개의 무작위로 초기화된 신경망, 2. 입자 기반 전파 알고리즘 및 3. 간단한 모델 예측 컨트</td></tr>
</table>

		롤러)
	ME-TRPO Model ensemble Trust Region Policy optimization	환경의 기본 진실로 간주되는 모델 앙상블에 trpo 적용
	simPLe	Model-based의 많은 트릭을 픽셀의 변형 자동 인코더 모델링 역학과 결합

Multi-agent★	COMA	
	QMIX	
	LIIR	
	MAAC	
	SEAC	
	G2ANET	
	AI-QMIX	
	ROMA	
	RODE	

Agent의 샘플 효율성	Transfer learning	하나의 문제에서 얻은 지식을 다른 관련 문제에 적용 DL에서 분류문제에 자주 쓰임
	Meta learning	Training time 간 접하지 않은 task, environment 잘 적응

REINFORCEMENT

i. 문제 : variance of returned θ is high(∵Monte Carlo) -> var을 줄이기 위해 actor-critic 도입

- Update parameters by stochastic gradient ascent
- Using policy gradient theorem
- Using return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

function REINFORCE
  Initialise θ arbitrarily
  for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
    end for
  end for
  return θ
end function

ii.

Actor-Critic

i. 환경 : 여러 Action 복합적 선택(power, mode 등 함께 컨트롤), Continuous action에 적용시 우수

ii. 배경 : MC approach인 REINFORCE algorithm이 var이 크다.(value function을 같이 써서 안정성을 높임) → 통신분야 특성상 TD적용이 어려울 수 있으니 θ update시 discount factor를 다소 크게 설정해서 후반부의 var을 줄이는 것도 하나의 방법으로 생각된다.

iii. Dueling DQN과 차이점 :

1. 마지막에 값을 합치는 지의 유무

2. Replay buffer : Dueling DQN과 달리, AC는 replay buffer를 사용하지 않음 -> 잘못된 state로 학습 시 성능 저하. (A3C가 이후 단점 보완)

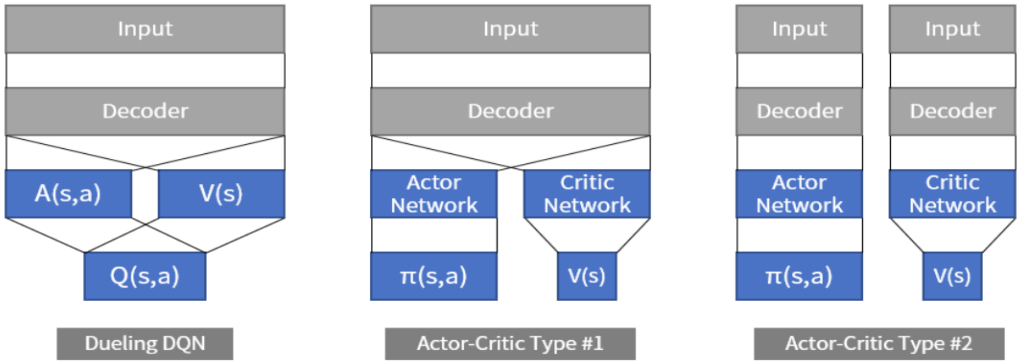


그림 1. 입력을 해석하는 파라미터(Decoder)를 공유(parameter sharing)하느냐 그렇지 않느냐에 따라 actor-critic 알고리즘의 구조는 크게 2가지로 나눌 수 있습니다.

function QAC
  Initialise s, θ
  Sample $a \sim \pi_\theta$
  for each step **do**
    Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$
    Sample action $a' \sim \pi_\theta(s', a')$
    $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
    $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
    $w \leftarrow w + \beta \delta \phi(s, a)$
    $a \leftarrow a', s \leftarrow s'$
  end for
end function

A2C(Synchronous Advantage-AC)

i. 환경 : actor의 기대출력으로 Q대신 A사용 -> Critic Network에서 나온 V(s)가 Actor Network 계산에 영향

$$A(s,a) \simeq R + \gamma V(s') - V(s)$$

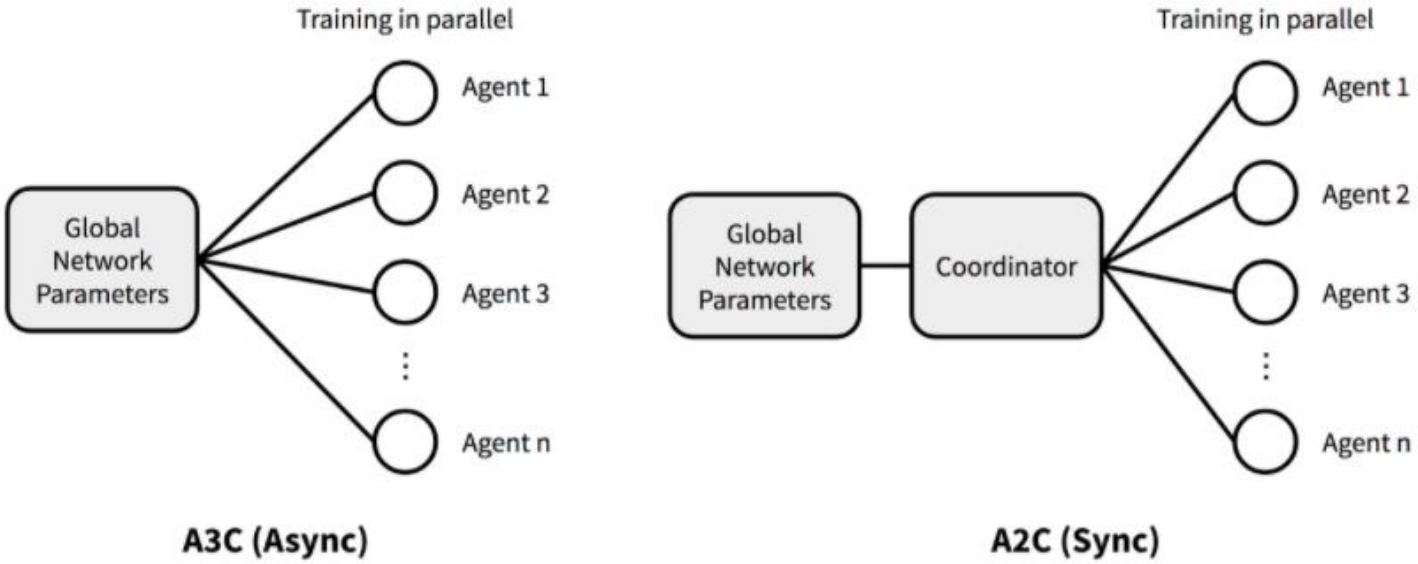
$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ A^{\pi_\theta}(s,a)\right]$$

실제로는 parameter 문제로 A자리에 approximate TD error 대입 $\delta_v = r + \gamma V_v(s') - V_v(s)$

ii. Synchronous A2C

1. Parallel training

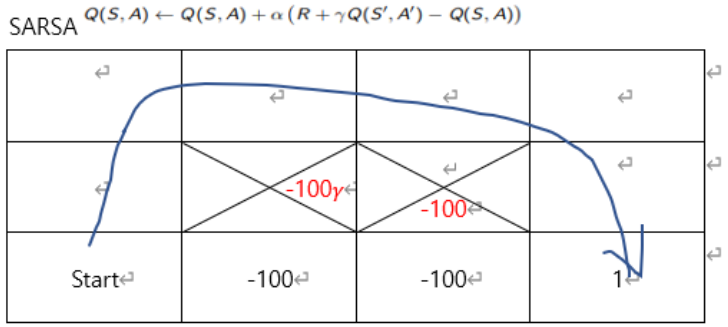
- 여러 workers가 환경과 interact(resulting r,s') -> master node의 actor-critic 전달 -> updated value, updated policy를 각 workers 전달
- synchronous이기 때문에 many workers가 있을 필요 없다는 주장도 있음 -> essentially what we do is to create "multiple versions of the environment"

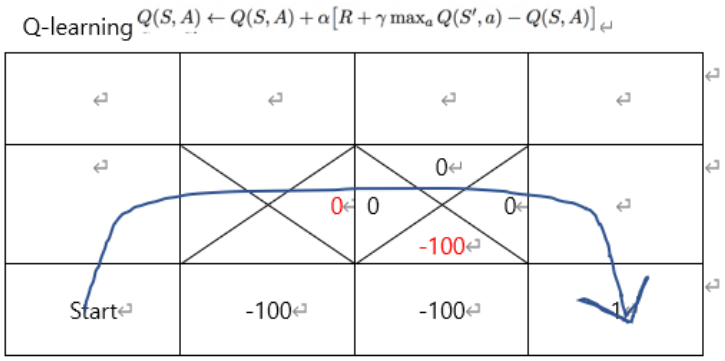


A3C와 A2C 구조 비교

Q-learning vs SARSA, on policy vs off-policy

i. Q-learning vs SARSA : SARSA는 겁쟁이다.(episode가 길어져 discounted factor때문에 sum of reward가 작아 결국 optimal path로 수렴은 하지만 그 과정이 길다.)





ii. On-policy는 local minimum에 빠질 가능성이 높다.(∵안 좋은 길로 가도 agent는 그 길에서의 state, reward만 알기 때문에)

iii. Importance Sampling : on-policy가 off-policy하기 위한 technique

1. Off-policy MC, off-policy TD는 사용안함. (∵MC : computation, TD : var)

2. Off-policy의 basic algorithm : Q-learning

$$\begin{aligned}\mathbb{E}_{X\sim P}[f(X)] &= \sum P(X)f(X) \\ &= \sum Q(X)\frac{P(X)}{Q(X)}f(X) \\ &= \mathbb{E}_{X\sim Q}\left[\frac{P(X)}{Q(X)}f(X)\right]\end{aligned}$$

Double Q-learning

i. Q-learning의 overestimation을 줄임

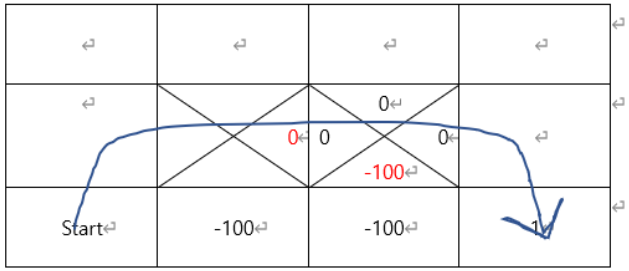
1. Q-learning은 action selection과 evaluation을 같은 parameter 사용했기에 overoptimistic(number of action ∝ overestimation)

$$Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(s_{t+1}, a; \theta_t); \theta_t)$$

2. Parameter θ_t^-, θ_t : switching randomly update

   - Actor-critic algorithm과 실제로 비슷한 구조로 이루어짐 : 차이점은 max연산의 유무

Q-learning $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$



ii. Pseudocode

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q_1(s,a)$ and $Q_2(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q.(terminal,\cdot) = 0$

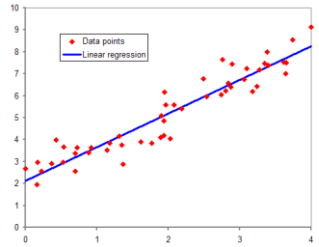
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:
      $Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S',a)\big) - Q_1(S,A)\Big)$
    else:
      $Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S',a)\big) - Q_2(S,A)\Big)$
    $S \leftarrow S'$
  until S is terminal

DQN(Deep Q Network) : Deep Learning + Function Approximation using both Experience Replay and Target Network

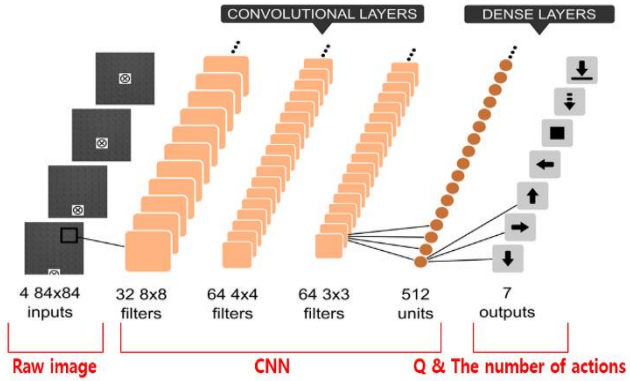
i. Raw pixel(84 x 84 x 4 image)를 directly input data using CNN network

1. Continuous state -> Q-learning(just update Q) 대신 Q_w w(weight)로 w update using regression with batch method

- Regression : dimension of state is high(In atari game, agent position and ball position 생각하면 이해하기 쉬움) -> 1:1대응 불가하기에 regression으로 추정



2. Network Architecture



- 원래는 CNN을 거쳐 Q를 regression하는 것인데 특이하게 action 종류만큼 output으로 나온다. 이유는 Q 구하려면 max 필요한데, CNN 내에서 max 연산 위해 모든 action 연산하는 것은 효율이 떨어짐. 따라서 CNN 통과 후 단순 비교하는 것이 간단.
- 정석DL은 모든 output에 대해 고려하는 것이 맞지만, DQN은 output 중 선택한 action만 고려해서 sample과의 MSE 계산(∵해당 action으로 update해야하기에)
- 참고로 image 크기(84 x 84 x 4)에서 4는 state에서 변화량을 파악하기 위한 것임. 왼쪽 그림에도 표현되어 있음. 단순 한 장의 사진으로는 공이 어디 방향으로 어느 속도로 움직이는지 파악 불가

ii. Contribution

1. Thanks to CNN, 인간처럼 인식 가능(input data = raw pixel image)
2. Experience Replay 잘 사용 using mini-batch SGD
   - Effect of SGD : reduced state correlation (correlation이 높으면 한 frame만 보면 유의미한 관찰 불가, 전체 다 보는 것도 sample이 너무 많다.)
   - Assume of SGD : state is i.i.d(independent identically distributed)(독립항등분포) -> s_t : uniformly randomly sampled -> 후에 PER(Prior Experience Replay에서 개선
3. Target Network 사용 : DQN과 똑 같은 Neural net하나 더 만들어 그 weight값이 가끔씩만 업데이트 되도록.(∵Q 학습 순간 target값도 따라 변해서 성능 저하)

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right] \quad (4)$$

target net의 weight는 DQN값 주기적으로 복사

먼저 나오는 Q가 target net에서 계산, 뒤의 Q가 DQN에서 계산

iii. Pseudocode

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N FIFO(First in, First out) 형식으로 data set 저장
Initialize action-value function Q with random weights Initialize "weight" of CNN
for episode $= 1, M$ **do** image
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$ Image 너무 크면 계산 힘드니 조금 잘라준다
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t e-greedy
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ e-greedy
    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 $\phi_{t+1} = \phi(s_{t+1})$ Frame skipping : frame너무 많으니 skip하겠다. x image를 본 후 이후 몇 구간은 image skip하고 동일 Action취하겠다. 본 논문 : skipping = 4
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$ Phi를 st라고 보면 됨
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ Terminal은 Q 없기때문에
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Y라는 sample을 가지고 그 action에 다가가도록 weight update

DDQN(Double DQN) 2015.

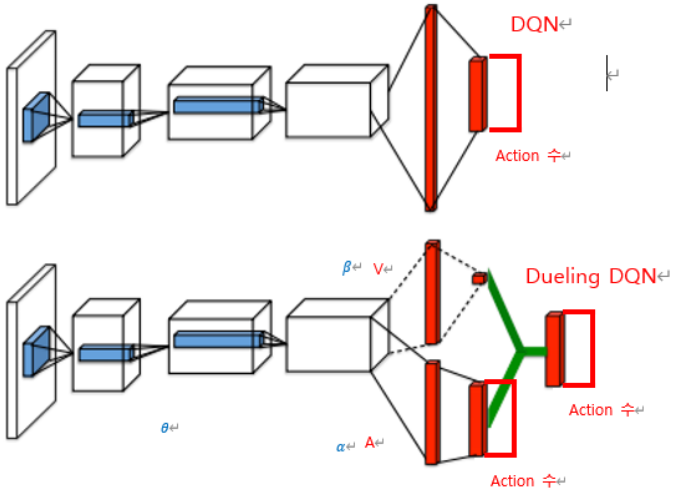
i. 기존 DQN의 target Q value는 overestimation이다.

ii. Theorem1 : $\max_a Q_t(s,a) \geq V_*(s) + \sqrt{\frac{c}{m-1}}$ -> max Q update하는 것은 optimal Q에 대해서 차이가 날 수 밖에 없다. The number of action $\propto$ error ex) cliff walking

iii. TD target : $R_t + \gamma Q_{w^-}(S_{t+1}, argmax Q_w(S_{t+1}, a_{t+1}))$

Target network Main network(DQN network)

Dueling DQN 2015.



i. Fully connected layer 반으로 나눔

ii. 하나의 layer 더 통과해서 network가 더 deep(성능 향상)

ii. 나눈 V, A를 단순하게 더할 것인가? No

1. $\mathbb{E}_{a\sim\pi(s)}\left[A^{\pi}(s,a)\right]=0.$ 이유는 $V^{\pi}(s)=\mathbb{E}_{a\sim\pi(s)}\left[Q^{\pi}(s,a)\right]$

$$Q(s,a;\theta,\alpha,\beta)=V(s;\theta,\beta)+\left(A(s,a;\theta,\alpha)-\frac{1}{|\mathcal{A}|}\sum_{a'}A(s,a';\theta,\alpha)\right)$$

➔

2. Deterministic policy 경우, $Q(s,a^*)=V(s)$ and hence $A(s,a^*)=0.$

$$Q(s,a;\theta,\alpha,\beta)=V(s;\theta,\beta)+\left(A(s,a;\theta,\alpha)-\max_{a'\in|\mathcal{A}|}A(s,a';\theta,\alpha)\right)$$

➔ : DQN은 한 action에 대해서만 update하지만 sum 전에 다른 action에 대해서도 같이 update된다

iv. Implementation : **알짜배기 Learning!**

1. 결과를 학습하지 않아도 어떤 state가 valuable한지 알 수 있다. In normal DQN, Q-value 계산 필요 even state or action이 안 좋더라도. However, in Dueling DQN, If state or action 나쁘면 혹은 valueless 학습하지 않는다. 어차피 Q-value learning 방해

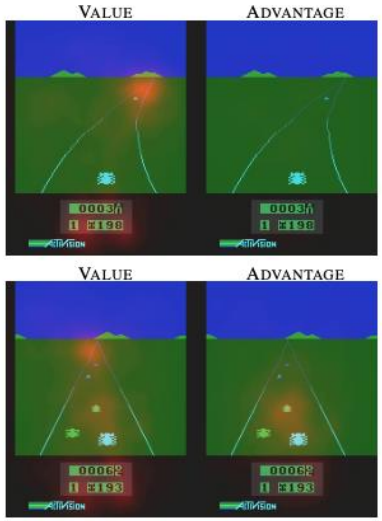
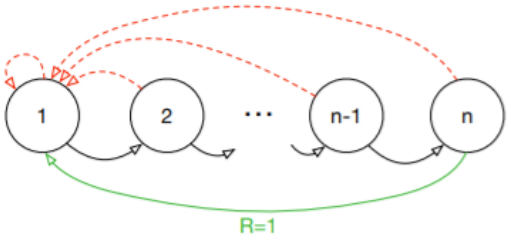


Figure 2. See, attend and drive: Value and advantage saliency maps (red-tinted overlay) on the Atari game Enduro, for a trained dueling architecture. The value stream learns to pay attention to the road. The advantage stream learns to pay attention only when there are cars immediately in front, so as to avoid collisions.

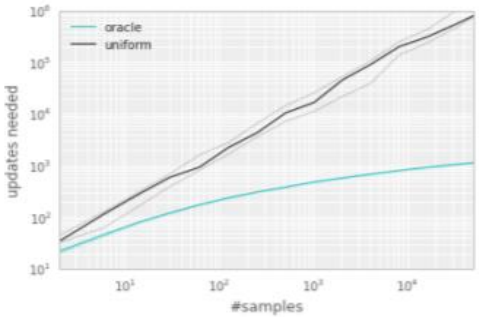
Intuitively, the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way. To illustrate this, consider the saliency maps shown in Figure 2[1]. These maps were generated by computing the Jacobians of the trained value and advantage streams with respect to the input video, following the method proposed by Simonyan et al. (2013). (The experimental section describes this methodology in more detail.) The figure shows the value and advantage saliency maps for two different time steps. In one time step (leftmost pair of images), we see that the value network stream pays attention to the road and in particular to the horizon, where new cars appear. It also pays attention to the score. The advantage stream on the other hand does not pay much attention to the visual input because its action choice is practically irrelevant when there are no cars in front. However, in the second time step (rightmost pair of images) the advantage stream pays attention as there is a car immediately in front, making its choice of action very relevant.

PER(Prioritized Experience Replay) 2015

i. 기존은 uniformly sampled (in order to i.i.d) -> 중요성 무시



Blind Cliffwalk



1. 오직 n state 도착할 때 까지 reward 0, wrong(절벽에서 떨어짐)이면 다시 처음으로
2. N state 보상 = 1
3. 즉, sample이 많아질수록 oracle과의 격차는 커진다.

ii. Prioritization -> state correlation(bias)가 심해지지 않을까? -> importance sampling 도입

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\beta}$$

IS
(Importance Sampling weight)

$$\mathbb{E}_{X\sim P}[f(X)] = \sum P(X)f(X)$$
$$= \sum Q(X)\frac{P(X)}{Q(X)}f(X)$$
$$= \mathbb{E}_{X\sim Q}\left[\frac{P(X)}{Q(X)}f(X)\right]$$

w_i

Uniform : 1/N

$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$

$\beta \to 0 : P(i)\ sampled$
$\beta \to 1 : uniform\ sampled$

iii. 우선순위 지표는 TD error(∵n-1 step에서 TD error는 1이므로)

1. TD error가 큰 sample에 대해서 update를 적극적으로 해라. 단, greedy sampling하면 overfitting -> 확률적으로 뽑자.

To overcome these issues, we introduce a stochastic sampling method that interpolates between pure greedy prioritization and uniform random sampling. We ensure that the probability of being sampled is monotonic in a transition's priority, while guaranteeing a non-zero probability even for the lowest-priority transition. Concretely, we define the probability of sampling transition i as

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \quad (1)$$

where $p_i > 0$ is the priority of transition i. The exponent α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case. $\alpha \propto 0 : uniformly\ random\ sample$ $\alpha \propto 1 : much\ sensitive\ to\ TD\ error(\delta)$ 즉, 하이퍼파라미터 알파로 prioritization 조절 가능

The first variant we consider is the direct, proportional prioritization where $p_i = |\delta_i| + \epsilon$, where ϵ is a small positive constant that prevents the edge-case of transitions not ever being revisited once their error is zero. The second variant is an indirect, rank-based prioritization where $p_i = \frac{1}{\text{rank}(i)}$, where $\text{rank}(i)$ is the rank of transition i when the replay memory is sorted according to $|\delta_i|$. In this case, P

iv. Pseudocode

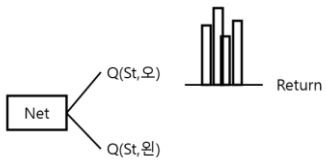
Algorithm 1 Double DQN with proportional prioritization

1: **Input:** minibatch k, step-size η, replay period K and size N, exponents α and β, budget T. 총 time step
2: Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
3: Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
4: **for** $t = 1$ **to** T **do**
5: Observe S_t, R_t, γ_t
6: Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$ 가장 최근 data를 maximal priority
7: **if** $t \equiv 0 \mod K$ **then** K마다 한번씩만 weight update(근현시 k=1로 많이 하는 듯)
8: **for** $j = 1$ **to** k **do**
9: Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
10: Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$ Normalize
11: Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$ Double DQN TD error
12: Update transition priority $p_j \leftarrow |\delta_j|$
13: Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$ Loss(sample)의 MSE 미분
14: **end for**
15: Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
16: From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$ 2015 DQN : weight copy
17: **end if**
18: Choose action $A_t \sim \pi_\theta(S_t)$
19: **end for**

line13 : importance sampling에서 도수를 gradient or error^2라고 보면 되는데(line15처럼 업데이트), minibatch update니까 sigma가 들어감. 예를 들어, sample = 12, (12-Qw)^2 minimize하는 방향으로 업데이트할 것이고 이것을 미분. (12-Qw)^2가 도수이므로 거기에다 importance sampling weight 곱해준다.

C51 2017. : A Distributional Perspective on Reinforcement Learning

$Z(x,a) \overset{D}{=} R(x,a) + \gamma Z(X', A').$: Target Distribution



$\Delta z = \frac{V_{max} - V_{min}}{N-1}$이므로 Support의 수, Support의 max값, Support의 min값 parameter 필요

1. Shift

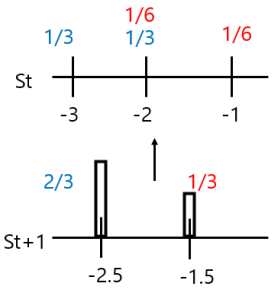
   DQN의 target : $R_t + \gamma max Q_\theta(S_{t+1}, a)$ -> C51의 target : $R_t + \gamma Z(S_{t+1}, a)$

   ① maxQ가 아닌 meanQ

   ② x축이 바뀜(St에서 return distribution의 support 구간과 St+1에서 return distribution의 support 구간이 다름)

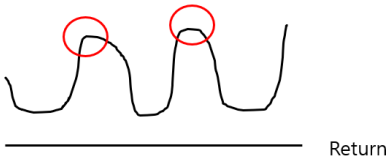
   ③ 우리의 목표는? X축 shift된 target의 분포와 최대한 같게 만들고 싶다! -> 문제점 1. Shift 구간 다름 2. 어떻게 minimize? 단순 차이의 제곱?

2. Projection : 1번 문제점(Shift 구간 다름) 해결



3. Loss : 2번 문제점(Minimiz) 해결

   ① What if 차이의 제곱? then, local optimum



   ② 확률 관련 Minimize = Likelihood 방법 = Cross Entropy minimize = KL Divergence minimize(두 확률분포 차이 계산)

   Loss = $-\sum p_i log q_i$

4. Contribution

    ① Network architecture

        기존 DQN은 num of output = num of action, 하지만 Distributional RL부터는 num of action X num of soft

    ② Loss 바꿈

5. Limitation

    ① Parameter w.r.t support 너무 많다

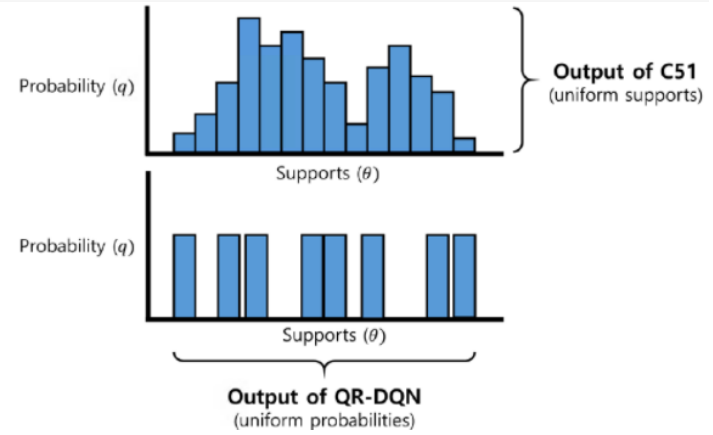
    ② Projection 과정이 포함되어 있다.

    ③ 수렴성 보장이 안된다.(wasserstein distance minimize 못함) : 그 다음 논문(QR-DQN에서 해결)

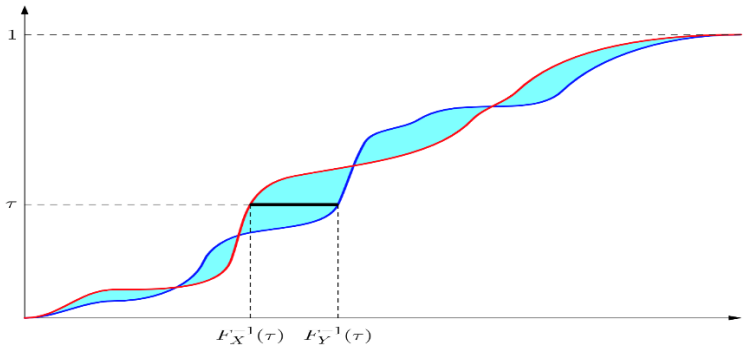
QR-DQN(2017) : Distributional Reinforcement Learning with Quantile Regression

1. C51과 차이점

   C51 = support 동일 간격 고정에 확률 구함, QR-DQN = 확률 동일 고정에 support 간격 구함(parameter : 오직 support 수만 있으면 됨! + projection 필요없음)

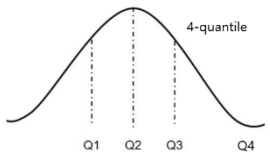


- p-wasserstein distance : $W_p(U,Y) = (\int_0^1 |F_Y^{-1}(w) - F_U^{-1}(w)|^p \, dw)^{\frac{1}{p}}$을 Wasserstein distance 를 줄이는 방향으로 학습을 수행하므로 distributional RL 의 수렴성을 수학적으로 만족함!

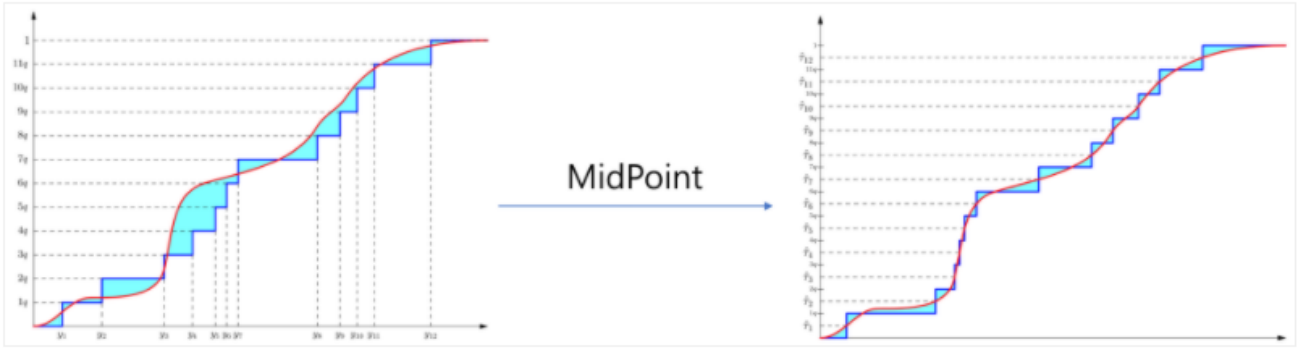


2. Quantile Huber Loss

   ① Quantile(1/4,2/4,3/4,4/4)



   ② Quantile midpoint(1/8,3/8,5/8,7/8)



그냥 quantile쓰는 것보다 midpoint로 하는 것이

Wasserstein distance 최소화)

③ Quantile Regression Loss(CDF에 적용하는 알고리즘)(Loss의 목적 + 단조증가의 특성을 유지하자)

Let target quantile = [2, 4, 5, 8] -> Cross entropy X(확률에 관해서만 사용할 수 있다.) -> MSE : [2, 4, 5, 8]에 대해 [2, 4, 5, 3], [2, 4, 5, 13] 구분이 안됨

Target : 2 4 5 8, Quantile midpoint = [1/8(τ1),3/8(τ2),5/8(τ3),7/8(τ4)]

Cand1 : 2 4 5 3, (8-3)*7/8

Cand2 : 2 4 5 13, (8-13)*(-1/8)

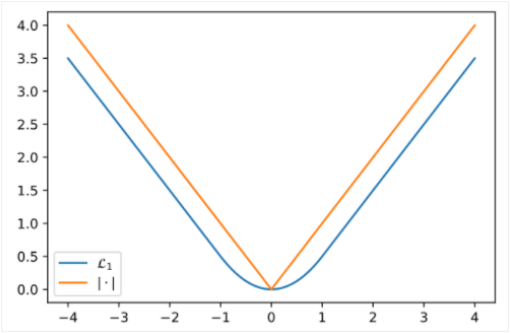
뺀 값이 음수면 $(\tau - 1)$곱, 따라서 단조증가 특성을 따르지 못하는 Cand1이 Loss가 훨씬 크다.

④ Quantile Huber Loss(논문에서 사용하는 Loss)

주황색선 : Quantile Regression Loss -> 0에서 미분불가

파란색선 : Quantile Huber Loss

2차함수 -> grad크기가 천차만별

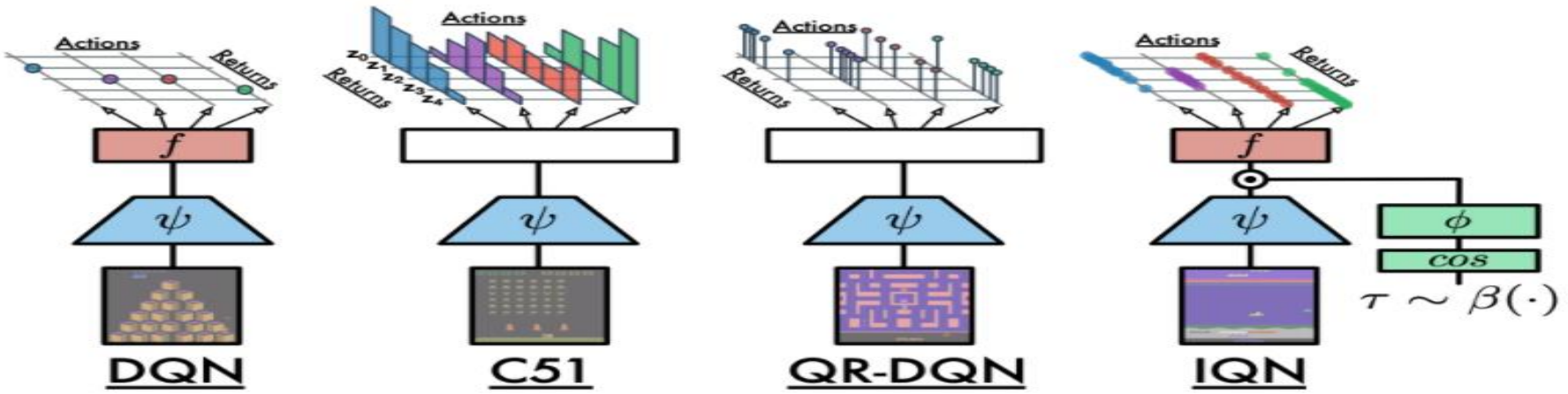


IQN(2018) : Implicit Quantile Networks for Distributional Reinforcement Learning

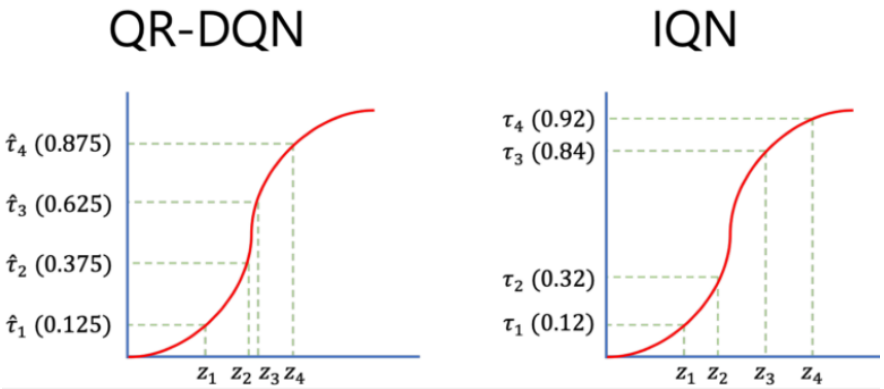
1. QR-DQN과 차이점

   동일한 확률로 나눈 Quantile을 이용하는 대신 확률들을 random sampling하고 해당하는 support 도출

   네트워크 구조



2. Sampling



   Sampling의 이점은? Risk-Sensitive하게 policy 선택 가능(Risk크다 = 분산이 크다, Risk작다 = 분산이 작다) -> Risk averse policy, Risk seeking policy

   베타양수 : Risk-averse, 베타음수 : Risk-seeking, 베타0 : Risk-neutral

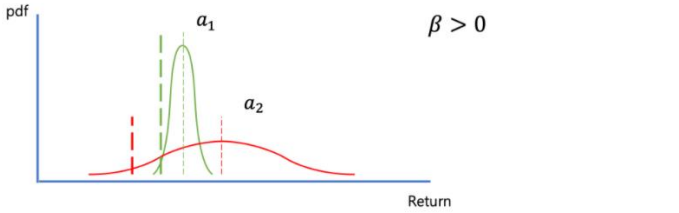
   Sampling 기법 : CPW(Cumulative Probability Weighting), Wang, POW(Power Formula), CVaR(Conditional Value at Risk)

Risk sensitive RL

Risk-averse

$a^* = argmax\, E(R(s,a_i)) - \beta std(R(s,a_i))$

$\beta > 0$



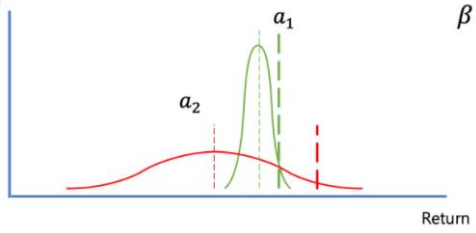
$E(R(s,a_1)) - \beta std(R(s,a_1)) > E(R(s,a_2)) - \beta std(R(s,a_2))$

Risk sensitive RL

Risk-seeking

$a^* = argmax\, E(R(s,a_i)) - \beta std(R(s,a_i))$

$\beta < 0$



$E(R(s,a_2)) + \beta std(R(s,a_2)) > E(R(s,a_1)) + \beta std(R(s,a_1))$

3. Network

   Tau를 embedding하는 function을 제외하고는 DQN(convolution function : psi + fully-connected layer : f)와 동일

   N cosine basis function $\phi_j(\tau) = ReLU(\sum_{i=0}^{n-1} \cos(\pi i \tau)\, w_{ij} + b_j)$

4. Quantile Huber Loss

$$\mathcal{L}(x_t, a_t, r_t, x_{t+1}) = \frac{1}{N'} \sum_{i=1}^{N} \sum_{j=1}^{N'} \rho_{\tau_i}^{\kappa}(\delta_t^{\tau_i, \tau'_j})$$

5. Action

$$a^* \leftarrow argmax_{a'} \frac{1}{K} \sum_{k}^{K} Z_{\tilde{\tau}_k}(x', a'), \qquad \tilde{\tau}_k \sim \beta(\cdot)$$

6. Conclusion : QR-DQN에 비해 뛰어난 성능

   Quantile random sampling하고 그 때의 support 도출 value distribution 획득

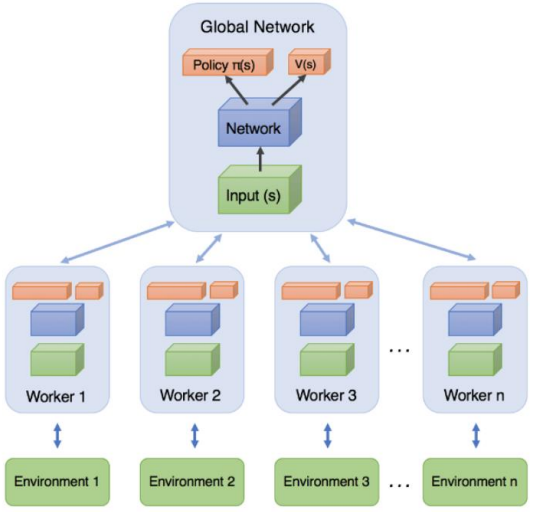
   Random sampling을 이용하여 risk-sensitive policy에 따른 action 선택

   Network 구조 변화

   단일 알고리즘이지만 여러 알고리즘 결합체(Rainbow)에 약간 못 미치는 성능 보임

A3C(Asynchronous Advantage Actor-Critic)(2016)(a.k.a vanilla policy gradient)

1. A2C agents 여러 개 독립 실행 & global network와 update => 다양한 최신 경험 사용
2. AC, A2C : on-policy이기 때문에 observed data가 correlated되어있다.
3. On-policy이기 때문에 data-efficiency, robustness 측면은 떨어짐
4. 절차
   - ① Worker reset to global network
   - ② Worker interacts with environment
   - ③ Worker calculates value and policy loss
   - ④ Worker gets gradients from losses
   - ⑤ Worker updates global network with gradient



Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
// Assume thread-specific parameter vectors θ' *and* θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$. Global network 0으로 초기화
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$ Thread들을 초기 global parameter로 initial update
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t+1$
    $T \leftarrow T+1$
  until terminal s_t **or** $t - t_{start} == t_{max}$ Time step t 끝날 때까지
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$ policy
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i;\theta'_v))^2 / \partial \theta'_v$ value
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

DPG(Deterministic Policy Gradient)(2016) : policy를 deterministic decision하는 것을 모델링

1. Back ground : SPG(Stochastic policy gradient)

   Theorem 1

   ρ: 해당 policy들의 성능을 나타내는 척도 (예. average reward per step)

   Theorem 1 (Policy Gradient) *For any MDP, in either the average-reward or start-state formulations,*

$$\frac{\partial \rho}{\partial \theta} = \sum_s d^\pi(s) \sum_a \frac{\partial \pi(s,a)}{\partial \theta} Q^\pi(s,a) \qquad E_s\left[\sum_a \frac{\partial \pi(s,a)}{\partial \theta} Q^\pi(s,a)\right] = \frac{\partial \rho}{\partial \theta}$$

   - D의 gradient 구하지 않아도 된다 = grad rho에 대한 unbiased estimation을 구할 수 있다 = grad rho을 sampling을 통해서 추정할 수 있다.
   - Exact Q값은 알 수가 없다.

   Theorem 2(fw : Qphi의 function approximation)

$$\sum_s d^\pi(s) \sum_a \pi(s,a)\left[Q^\pi(s_t,a_t) - f_w(s_t,a_t)\right]\frac{\partial f_w(s,a)}{\partial w} = 0 \quad + \quad \frac{\partial f_w(s,a)}{\partial w} = \frac{\partial \pi(s,a)}{\partial \theta}\frac{1}{\pi(s,a)}$$

: Compatibility condition

$$\frac{\partial \rho}{\partial \theta} = \sum_s d^\pi(s) \sum_a \frac{\partial \pi(s,a)}{\partial \theta} f_w(s,a)$$

2. Contribution
   - Var[policy]가 0에 수렴할 경우, DPG는 SPG와 동일 -> Theorem2로 인해 기존의 PG기법들을 DPG 적용 가능
   - 적절한 exploration을 위해 model-free, off-policy actor-critic algorithm 제안
   - DPG는 SPG보다 성능이 좋다.(특히 high dimensional action space에서, computation 양이 적음) 이유는 DPG는 policy gradient의 state space만 계산하기 때문, SPG의 policy gradient는 action space, state space 둘 다 계산

3. Objective function :

$$J(\theta) = \int_s \rho^{\mu}(s) Q(s, \mu_\theta(s)) ds$$

$$\begin{aligned} \nabla_\theta J(\theta) &= \int_s \rho^{\mu}(s) \nabla_a Q^{\mu}(s,a) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)} ds \\ &= E_{s\sim\rho^{\mu}}[\nabla_a Q^{\mu}(s,a) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}] \end{aligned}$$

DDPG(Deep Deterministic Policy Gradient)

1. 배경
   - DQN : raw pixel input으로 high dimensional observation space 가능하지만 only discrete & low dimensional action space -> **continuous action space!!**
   - Discritization을 exponential하게 해도 continuous action 표현 한계
   - 이를 위해, Model-free, off-policy, actor-critic + Based on DPG + Replay buffer + Target Q Network
2. Background
   - Bellman equation
     - 상태 s_t에서 행동 a_t를 취했을 때 Expected return은 다음과 같습니다.

$$Q^{\pi}(s_t, a_t) = \mathbf{E}_{r_{i\geq t}, s_{i\geq t}\sim E, a_{i\geq t}\sim\pi}[R_t | s_t, a_t]$$

     - 벨만 방정식을 사용하여 위의 식을 변형합니다.

$$Q^{\pi}(s_t, a_t) = \mathbf{E}_{r_t, s_t\sim E}[r(s_t, a_t) + \gamma \mathbf{E}_{a_{t+1}\sim\pi}[Q^{\pi}(s_{t+1}, a_{t+1})]]$$

     - Determinsitc policy를 가정합니다.

$$Q^{\mu}(s_t, a_t) = \mathbf{E}_{r_t, s_t\sim E}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1}))]$$

- Q learning

$$L(\theta^Q) = \mathbf{E}_{s_t \backsim \rho^\beta, a_t \backsim \beta, r_t \backsim E}[(Q(s_t, a_t|\theta^Q) - y_t)^2]$$

- 위의 수식에 대한 추가설명

  - β는 behavior policy를 의미합니다.
  - $y_t = r(s_t, a_t) + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1}))$
  - $\mu(s) = argmax_a Q(s, a)$
    - Q learning은 위와 같이 $argmax$라는 deterministic policy를 사용하기 때문에 off policy로 사용할 수 있습니다.

- DPG

$$\nabla_{\theta^\mu} J \approx \mathbf{E}_{s_t \backsim \rho^\beta}[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)}] = \mathbf{E}_{s_t \backsim \rho^\beta}[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s|Q^\mu)|_{s=s_t}]$$

3. Algorithm

- Replay buffer

  DDPG는 DQN에 사용된 Replay buffer를 사용해 online batch update 가능 + NN function approximator to learn in large state space

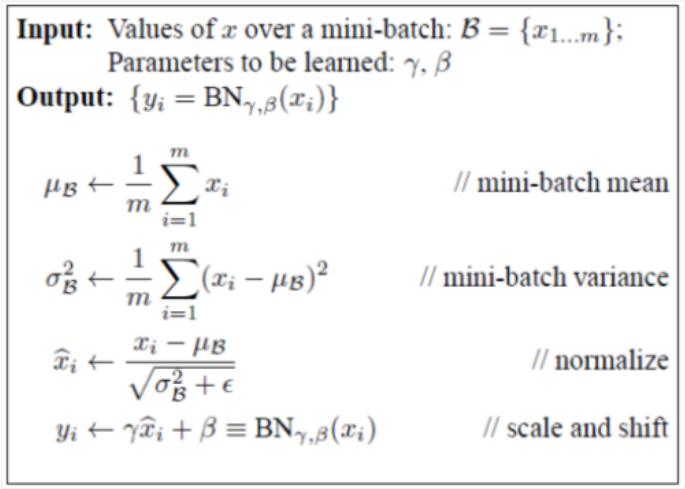
- Soft target update

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  - DQN에서는 일정 주기마다 origin network의 weight를 target network로 직접 복사해서 사용합니다.
  - DDPG에서는 exponential moving average(지수이동평균) 식으로 대체합니다.
  - soft update가 DQN에서 사용했던 방식에 비해 어떤 장점이 있는지는 명확하게 설명되어있지 않지만 stochatic gradient descent와 같이 급격하게 학습이 진행되는 것을 막기 위해 사용하는 것 같습니다.

- Batch normalization



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

  - 서로 scale이 다른 feature를 state로 사용할 때에 Neural Net이 일반화에서 어려움을 겪습니다.
    - 이걸 해결하기 위해서는 원래 직접 스케일을 조정해주었습니다.
  - 하지만 각 layer의 Input을 Unit Gaussian이 되도록 강제하는 BatchNormalization을 사용하여 이 문제를 해결합니다.

- Noise process

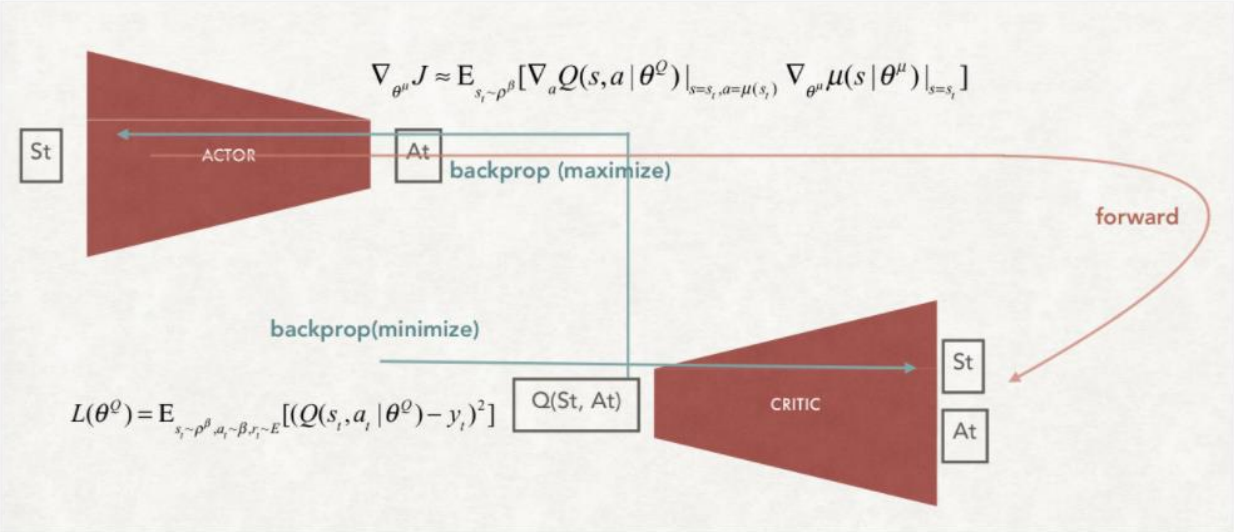
  DDPG 에서는 Exploration을 위해서 output으로 나온 행동에 노이즈를 추가해줍니다.

  ORNSTEIN UHLENBECK PROCESS(OU)

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t$$

  - OU Process는 평균으로 회귀하는 random process입니다.
  - θ는 얼마나 빨리 평균으로 회귀할 지를 나타내는 파라미터이며 μ는 평균을 의미합니다.
  - σ는 process의 변동성을 의미하며 W_t는 Wiener process를 의미합니다.
  - 따라서 이전의 noise들과 temporally correlated입니다.
  - 위와 같은 temporally correlated noise process를 사용하는 이유는 physical control과 같은 관성이 있는 환경에서 학습 시킬 때 보다 효과적이기 때문입니다.

- Diagram and pseudocode



Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

4. Conclusion

- Continuous action space를 robust하게 풀어냄
- Non-linear function approximator를 쓰는 것은 수렴성을 보장하진 않지만 안정적으로 수렴하는 것을 실험적으로 나타냄
- Atari Domain에서는 DQN보다 적은 step으로 수렴하는 것을 관찰

SAC(2018) Soft Actor-Critic : exploration을 더 잘하기 위해 reward에 policy에 대한 entropy measure 추가

1. 구성요소
    - Actor-critic 구조(policy function network + value function network)
    - Off-policy
    - Entropy maximization in order to be stability + exploration
2. Objective function

$$J(\theta) = \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_{\pi\theta}}[r(s_t,a_t) + \alpha\mathcal{H}(\pi_\theta(\cdot|s_t))]$$

결국 SAC는 세가지 함수를 학습하는데 초점을 맞추고 있다.

- Parameter θ에 대한 policy, π_θ
- Parameter w에 의해 결정되는 Soft Q-value function, Q_w
- Parameter ψ에 의해 결정되는 Soft state-value function, V_ψ
  : 이론적으로는 Q와 π를 통해서 V를 추론할 수 있지만, 실제 상황에서는 이렇게 하면 training에 안정성을 더할 수 있다.

Soft Q-value (아니면 state-action value) 와 soft state value는 다음과 같이 정의할 수 있다

$$\begin{aligned} Q(s_t,a_t) &= r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim\rho_\pi(s)}[V(s_{t+1})] && \text{; according to Bellman equation} \\ \text{where } V(s_t) &= \mathbb{E}_{a_t\sim\pi}[Q(s_t,a_t) - \alpha\log\pi(a_t|s_t)] && \text{; soft state value function} \end{aligned}$$

여기서 Q-value내의 $V(s_{t+1})$을 state value function으로 치환시키면 아래 식과 같이 유도할 수 있다.

$$Q(s_t,a_t) = r(s_t,a_t) + \gamma\mathbb{E}_{(s_{t+1},a_{t+1})\sim\rho_\pi}[Q(s_{t+1},a_{t+1}) - \alpha\log\pi(a_{t+1}|s_{t+1})]$$

여기서 $\rho_\pi(s)$와 $\rho_\pi(s,a)$는 policy $\pi(a|s)$에 의해서 도출되는 state distribution의 state marginal과 state-action marginal 인데, 이 부분은 DPG에서도 비슷한 방식으로 유도했었다.

soft state value function은 mean squared error를 최소화하는 방향으로 학습된다.

$$J_V(\psi) = \mathbb{E}_{s_t \sim D}[\frac{1}{2}\big(V_\psi(s_t) - \mathbb{E}[Q_w(s_t, a_t) - \log \pi_\theta(a_t|s_t)]\big)^2]$$
$$\text{with gradient: } \nabla_\psi J_V(\psi) = \nabla_\psi V_\psi(s_t)\big(V_\psi(s_t) - Q_w(s_t, a_t) + \log \pi_\theta(a_t|s_t)\big)$$

위의 식에서 D은 replay buffer를 말한다.

soft Q function은 soft Bellman residual을 최소화하는 방향으로 학습된다.

$$J_Q(w) = \mathbb{E}_{(s_t,a_t)\sim D}[\frac{1}{2}\big(Q_w(s_t, a_t) - (r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1}\sim \rho_\pi(s)}[V_{\bar{\psi}}(s_{t+1})]\big)^2]$$
$$\text{with gradient: } \nabla_w J_Q(w) = \nabla_w Q_w(s_t, a_t)\big(Q_w(s_t, a_t) - r(s_t, a_t) - \gamma V_{\bar{\psi}}(s_{t+1})\big)$$

잘 보일지는 모르겠지만 $\bar{\psi}$는 exponential moving average를 가지는 (혹은 "어려운 방법으로" 주기적으로만 update되는) target value function을 말하는데, 마치 DQN에서 training을 안정화시키기 위해서 target Q network의 parameter를 다루는 방법과 동일하다.

SAC는 KL-divergence를 최소화하는 방향으로 policy를 update한다.

$$\begin{aligned}
\pi_{\text{new}} &= \arg\min_{\pi' \in \Pi} D_{\text{KL}}\Big(\pi'(.|s_t) \| \frac{\exp(Q^{\pi_{\text{old}}}(s_t, .))}{Z^{\pi_{\text{old}}}(s_t)}\Big) \\
&= \arg\min_{\pi' \in \Pi} D_{\text{KL}}\big(\pi'(.|s_t) \| \exp(Q^{\pi_{\text{old}}}(s_t, .) - \log Z^{\pi_{\text{old}}}(s_t))\big) \\
\text{objective for update: } J_\pi(\theta) &= \nabla_\theta D_{\text{KL}}\big(\pi_\theta(.|s_t) \| \exp(Q_w(s_t, .) - \log Z_w(s_t))\big) \\
&= \mathbb{E}_{a_t \sim \pi}\Big[- \log \big(\frac{\exp(Q_w(s_t, a_t) - \log Z_w(s_t))}{\pi_\theta(a_t|s_t)}\big)\Big] \\
&= \mathbb{E}_{a_t \sim \pi}[\log \pi_\theta(a_t|s_t) - Q_w(s_t, a_t) + \log Z_w(s_t)]
\end{aligned}$$

위의 식에서 policy Π는 현재의 policy를 가지고 쉽게 바꿀수 있는 잠재적인 policy들의 집합이다. 예를 들어 Π는 modeling하기는 어렵지만, 직관적이고, 쉽게 다룰 수 있는 Gaussian mixture distribution의 집합일 수도 있다. $Z^{\pi_{\text{old}}}(s_t)$는 distribution을 normalize시키는 partition function이다. 이건 쉽게 다룰수 있는건 아니지만 그렇다고 gradient에 영향을 미치는 것은 아니다. 결국 objective function인 $J_\pi(\theta)$를 최소화하는 것은 policy들의 집합인 Π를 어떻게 선택하냐에 달린 것이다.

Algorithm 1 Soft Actor-Critic

Inputs: The learning rates, λ_π, λ_Q, and λ_V for functions π_θ, Q_w, and V_ψ respectively; the weighting factor τ for exponential moving average.

Initialize parameters θ, w, ψ, and $\bar{\psi}$.
for each iteration **do**
    (In practice, a combination of a single environment step and multiple gradient steps is found to work best.)
    for each environment setup **do**
        $a_t \sim \pi_\theta(a_t|s_t)$
        $s_{t+1} \sim \rho_\pi(s_{t+1}|s_t, a_t)$
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1}\}$
    for each gradient update step **do**
        $\psi \leftarrow \psi - \lambda_V \nabla_\psi J_V(\psi)$.
        $w \leftarrow w - \lambda_Q \nabla_w J_Q(w)$.
        $\theta \leftarrow \theta - \lambda_\pi \nabla_\theta J_\pi(\theta)$.
        $\bar{\psi} \leftarrow \tau\psi + (1-\tau)\bar{\psi})$.

TD3(2018) : Twin delayed DDPG -> 3가지 기법을 활용해 Q-learning 문제(overestimation) 해결

1. Clipped Double Q-learning

$$y_1 = r + \gamma \min_{i=1,2} Q_{w_i}(s', \mu_{\theta_1}(s'))$$
$$y_2 = r + \gamma \min_{i=1,2} Q_{w_i}(s', \mu_{\theta_2}(s'))$$

Double Q learning : action selection, Q-value 측정을 다른 network에서 실시 -> policy가 느리게 변화해서 independent decision하기에는 두 network가 유사

Clipped Double Q-learning : 두 network 중 min estimation 선택하게해서 underestimation bias를 선호하게끔

2. Delayed update of Target and Policy Network

Actor-critic : policy network, value network 연관이 깊기 때문에 하나의 network가 나쁘게 동작하면 나머지 network에도 악영향미침(var 증가)

Target network개념과 비슷하게 policy network를 value network보다 적은 빈도로 update.

3. Target Policy Smoothing

$$y = r + \gamma Q_w(s', \mu_\theta(s') + \epsilon)$$
$$\epsilon \sim \text{clip}(\mathcal{N}(0, \sigma), -c, +c) \quad ; \text{clipped random noises.}$$

TD3은 value function에 대한 smoothing regularization strategy 적용(이유는 value function에서 발생하는 peak에 의해 deterministic policy가 overfit될 수 있음)

Algorithm 1 TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_ϕ
with random parameters θ_1, θ_2, ϕ
Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1$ **to** T **do**
  Select action with exploration noise $a \sim \pi(s) + \epsilon$,
  $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'
  Store transition tuple (s, a, r, s') in $\mathcal{B}$

  Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$
  $\tilde{a} \leftarrow \pi_{\phi'}(s) + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$ Target policy smoothing
  $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$ Clipped Double Q-learning
  Update critics $\theta_i \leftarrow \min_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  if t mod d **then** Delayed update of target and policy networks
    Update ϕ by the deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
    Update target networks:
    $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
    $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
  end if
end for

TRPO(2015) Trust Region Policy Optimization

- Original Goal : Expected discounted reward 최대화

$$\max_{\pi} \eta(\pi)$$

- Conservative Policy iteration : Directly update하는 것은 힘들다 -> Kakade & Langford 기법으로 policy update가 최소 성능 악화는 되지 않음.

$$\max L_{\pi}(\tilde{\pi}) = \eta(\pi) + \sum_{s} \rho_{\pi}(s) \sum_{a} \tilde{\pi}(a|s) A_{\pi}(s,a)$$

Policy가 바뀌어도 이전 state distribution 계속 이용..
-> 얼마나 많은 변화 무시를 허용할 것인가? => Trust Region

- Theorem 1 of TRPO : 기존 iteration은 new, old policy가 함께 있어서 실용적이지 않다 -> 온전히 새로운 policy만 update!

$$\max L_{\pi_{\text{old}}}(\pi_{\text{new}}) - \frac{4\epsilon\gamma}{(1-\gamma)^2}\alpha^2, \quad \left(\alpha = D_{\text{TV}}^{\max}(\pi_{\text{old}}, \pi_{\text{new}})\right)$$

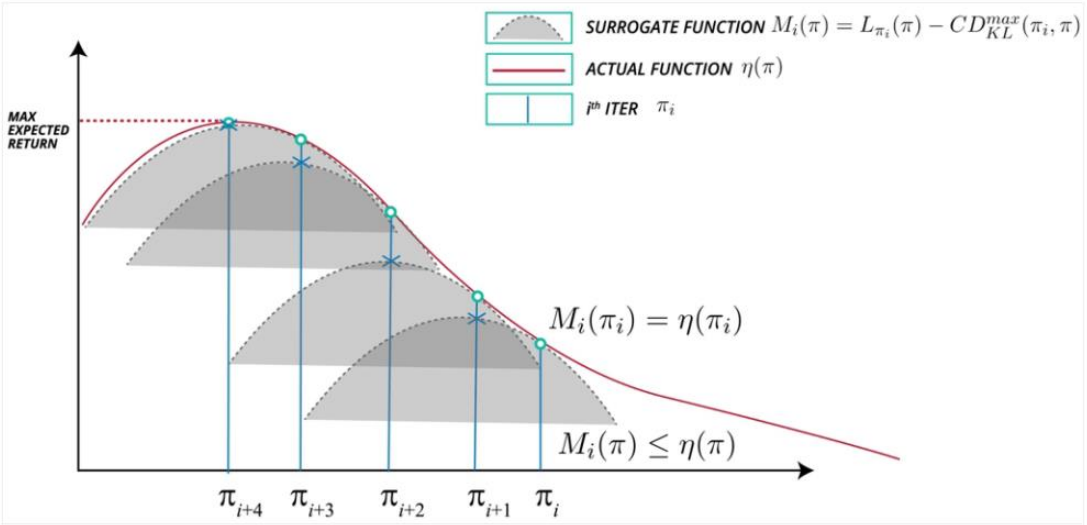
: eta phi의 lower bound

- KL Divergence version of Theorem 1

$D_{\text{TV}}(p \parallel q)^2 \le D_{\text{KL}}(p \parallel q)$ 에 의해 eta phi의 lower bound가 아래와 같이 바뀜

$$\max L_{\pi}(\tilde{\pi}) - C \cdot D_{\text{KL}}^{\max}(\pi, \tilde{\pi}), \quad \left(C = \frac{4\epsilon\gamma}{(1-\gamma)^2}\right)$$

$\eta(\pi) \geq M_i$: surrogate funtion



M_i는 π_i일 때 equality가 되는 η에 대한 surrogate function입니다. TRPO는 이 surrogate function을 최대화하고 KL divergence를 penalty가 아닌 constraint로 두는 알고리듬입니다.

- Using parameterized policy : 낮은 dimension으로 parameterized된 policy 사용

$$\max_{\theta} L_{\theta_{\text{old}}}(\theta) - C \cdot D_{\text{KL}}^{\max}(\theta_{\text{old}}, \theta)$$

- Trust region constraint : policy update시 지나치게 많이 변하는 것을 방지하기 위해 trust region을 constraint(delta)로 설정

$$\begin{aligned} \max_{\theta} \quad & L_{\theta_{\text{old}}}(\theta) \\ \text{s.t.} \quad & D_{\text{KL}}^{\max}(\theta_{\text{old}}, \theta) \leq \delta \end{aligned}$$

- Heuristic approximation : 하지만 위의 delta는 모든 state에 대해서 성립해야하므로 조건이 매우 까다롭다. -> state distribution에 대한 평균으로 나타냄

$$\begin{aligned} \max_{\theta} \quad & L_{\theta_{\text{old}}}(\theta) \\ \text{s.t.} \quad & \overline{D}_{\text{KL}}^{\rho_{\text{old}}}(\theta_{\text{old}}, \theta) \leq \delta \end{aligned}$$

- MC simulation : sampling을 통한 계산이 가능하도록 식 변형

$$\max_{\theta} \quad E_{s\sim\rho_{\theta_{\text{old}}},a\sim q}\left[\frac{\pi_\theta(a|s)}{q(a|s)}Q_{\theta_{\text{old}}}(s,a)\right]$$
$$\text{s.t.} \quad E_{s\sim\rho_{\theta_{\text{old}}}}\left[D_{\text{KL}}\left(\pi_{\theta_{\text{old}}}(\cdot|s)\parallel\pi_\theta(\cdot|s)\right)\right]\leq\delta$$

- Efficiently solving TRPO : 문제를 효율적으로 풀기 위해 approximation 적용

$$\max_{\theta} \quad \nabla_\theta L_{\theta_{\text{old}}}(\theta)|_{\theta=\theta_{\text{old}}}\left(\theta-\theta_{\text{old}}\right)$$ 1차 근사

$$\text{s.t.} \quad \frac{1}{2}\left(\theta_{\text{old}}-\theta\right)^T A\left(\theta_{\text{old}}\right)\left(\theta_{\text{old}}-\theta\right)\leq\delta$$ 2차 근사

이후 natural gradient or conjugate gradient

$$A_{ij}=\frac{\partial}{\partial\theta_i}\frac{\partial}{\partial\theta_j}\overline{D}_{\text{KL}}^{\rho_{\text{old}}}\left(\theta_{\text{old}},\theta\right)$$

PPO(2017) : Proximal Policy Optimization Algorithm : clipped surrogate function을 사용해서 TRPO와 유사한 성능을 내면서 더 간단하게 구현 가능

1. Clipped probability ratio를 포함하는 objective function 제안 : TRPO의 data efficiency와 robustness 유지하면서 1차근사만 사용 + policy 성능에 대한 lower bound 제공

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \quad J^{\text{TRPO}}(\theta) = \mathbb{E}[r(\theta)\hat{A}_{\theta_{\text{old}}}(s,a)]$$

: theta old와 theta 거리 제한이 없다면, TRPO의 J를 maximize하는 작업자체는 안정성이 없다.

$$J^{\text{CLIP}}(\theta) = \mathbb{E}[\min(r(\theta)\hat{A}_{\theta_{\text{old}}}(s,a), \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_{\theta_{\text{old}}}(s,a))]$$

: PPO에서는 r(theta)를 1이내에 유지되게끔 constraint 설정(clip개념)

PPO는 objective function의 원래 값과 clip된 값 중 작은 값을 취하는 형태이므로 극도로 policy를 update하는 동작이 생략됨

2.

policy fuction (actor)와 value function (critic)간에 parameter를 공유하는 형태의 network에다가 PPO를 적용하는 경우, 앞에서 소개한 clipped reward를 쓰는 것에 덧붙여서, objective function에다가 value estimation에 대한 error term (아래 식 중 빨간 부분) 와 entropy term (아래 식 중 파란 부분)을 덧붙여서 사용하는데, 이를 통해 exploration을 충분히 수행할 수 있도록 해준다.

$$J^{\text{CLIP}'}(\theta) = \mathbb{E}[J^{\text{CLIP}}(\theta) - c_1(V_\theta(s) - V_{\text{target}})^2 + c_2 H(s, \pi_\theta(\cdot))]$$

여기서도 역시 c_1, c_2는 hypterparameter constant이다.