* Value Function Approximation (VFA)

state, action을 parameter로 받음

: state나 action을 table로 저장하지 않고 w라는 새로운 벡터를 이용해 value function을 추정한다
↳ Tabular representation 사용

- SARSA, Q-Learning, MC, TD 등의 알고리즘은 경험을 통해 좋은 policy를 얻음

⇒ state, action의 차원이 크면 어려워짐, table이 커지게 되어 memory 수 (real world에서 적용 어려움)

∴ VFA를 사용해 state가 함수 input으로 입력되면 w라는 parameter를 거쳐 action value function을 output으로 도출

- $\hat{v}(s,w) \approx V_\pi(s)$, $\hat{q}(s,a,w) \approx q_\pi(s,a)$ ↳ approximate action-value function

$s \to \boxed{w} \to \hat{v}(s,w)$

$s, a \to \boxed{w} \to \hat{q}(s,a,w)$

- 장점 : ① reduce memory need to store $(P,R), V, Q, \pi$: transition, reward, value …)
  ② Reduce computation to look up values
  ③ Reduce experience (# of data for good V, Q, π) needed to find optimal value, policy : sample efficient
- 단점 : ① 필요한 data 수가 줄어들지만 너무 작은 수인 경우 VFA 사용시 훨씬 많은 approximation이 될 수 있음
  ② memory, computation, data 수 등은 부족하지만 표현의 용량 (representational capacity)도 함께 줄어듦
- 종류 : ① linear ② Neural Network ③ Decision tree ④ Nearest Neighbor ⑤ Fourier / Wavelet based

†) SGD (Stochastic Gradient Descent) - basis of VFA

: find local minima of error function by Gradient Descent (기울기가 최소인 곳을 찾아 적당한 weight 찾음)

$w \leftarrow w - \alpha \nabla_w \mathrm{Error}^2$ (α : step-size)

⇒ VFA에 적용하면 $-\frac{1}{2}\nabla_w J(w) = (q_\pi(S,A) - \hat{q}(S,A,w))\nabla_w \hat{q}(S,A,w)$, $\Delta w = \alpha(q_\pi(S,A) - \hat{q}(S,A,w))\nabla_w \hat{q}(S,A,w)$

- Linear function Approximation

: Value function을 보정해주기 위한 weight (w)에 의해 value function이 결정되는 linear function (parameterized value function)을 만들어 가는 과정

$\boxed{\hat{v}(s,w) = \sum w_i x_i(s)\ ,\ x_i(s) : \text{해당 state } s \text{에서의 feature - feature vector}}$

→ 각 action은 weight 벡터와 곱해져 Q value를 만드는데 이때 weight는 feature의 importance를 나타냄

- Neural Network based function Approximation

: Linear function으로 정의할 수 없는 경우 유의한 feature가 아닌 새로운 feature를 찾아 function approximation 함

ex. DQN

→ Value function approximation에서 Neural Network를 사용하면 결국 Deep RL

- function approximation은 on-policy / off-policy 상관 없이 사용 가능

ex. ① MC (Monte Carlo Method) use target as one sample return G_t

$\boxed{\Delta w = \alpha(G_t - \hat{q}(S_t, A_t, w))\nabla_w \hat{q}(S_t, A_t, w)}$

② SARSA (TD(0)) use TD target

$\boxed{\Delta w = \alpha(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, w) - \hat{q}(S_t, A_t, w))\nabla_w \hat{q}(S_t, A_t, w)}$

③ TD(λ) use λ-return q_t^λ (forward view)

$\boxed{\Delta w = \alpha(q_t^\lambda - \hat{q}(S_t, A_t, w))\nabla_w \hat{q}(S_t, A_t, w)}$

Incremental Control

- Minimize mean-squared error between $\hat{q}(S,A,w)$, $q_\pi(S,A)$

: $\boxed{J(w) = E_\pi[(q_\pi(S,A) - \hat{q}(S,A,w))^2]}$

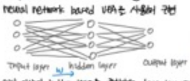
- VFA 평가 방법
  - 1) Generalization : value function이 모든 state에 대해 정확히 되는가
    → value function이 정의 되었을 때 한 state의 value가 update 될 경우 이 정보가 다른 state의 value에도 영향을 끼치는가
    ∴ 잘 되어있다면 기존 방문 비율에도 정확히 일반화한 value function이 됨 : 높을 수록 좋음
  - 2) Discrimination : state가 여러개 있을 때 state 들이 대한 value를 잘 구분하여 차이를 주는가
    ∴ 잘 되어있다면 주변 state에 대해 확실 수렴하는 policy 만들 수 있음

⇒ 둘다 잘 되어야 하지만 둘 사이에 trade-off 존재

- mini-batch SGD 사용
- 일반 SGD 외에 다양한 optimizer 사용
  ex. Momentum, Nag, RMS Prop, Adam
- Experience Replay : replay memory를 만들어 놓고 agent의 경험을 time-step 마다 통째 저장해둠
  → action value function의 parameter를 update 하는 것은 memory에 저장해둔 모든 transition
  → DQN에서 사용

* Deep Reinforcement Learning

- neural network based VFA를 사용해 구현



2층이 쌓인 Shallow learning

- 2개 이상의 hidden layer를 구성해야 deep learning
- 학습 전에 train / test data set 나누어 overfitting 할게 방지
- 입력 data가 다음 node로 전달될 때 w 사용
- loss function : feed forward 시 예측값과 실제값 비교
  ⇒ RL에서는 (target - prediction) using to update
- 1 epoch : 전체 training data를 한 번 학습하는 것
  ↳ 너무 작으면 underfitting, 너무 크면 overfitting
- 1 iteration : 1 epoch를 끝내기 위한 mini-batch 수
- batch size : mini-batch를 구성하는 데이터 크기

* DQN (Deep Q-Network) : mini batch / experience Replay / Fixed Q-target 사용

: station-action value Q를 Deep learning을 통해 approximate 함

- 2 networks for Q
  - Prediction Network : parameter (θ_i) for Q-network at iteration i
  - Target Network : Parameter (θ_i^-) for computing target network at iteration i
    ↳ only updated with θ_i every C steps , held between individual updates
- deep learning의 문제점 : ① label (정답)이 있는 data로 학습시키는데, RL은 label이 없고 reward로 학습시키는데 제한적인 학습이 됨
  ② Data sample이 i.i.d (서로 독립적)이라고 가정하는데, RL은 다음 state와 현재 state사이 correlation이 커 가정이 성립하지 않음

⇒ Experience Replay 사용 : Episode 마다 DL의 weight를 업데이트하지 않고, Time-step 마다 $[S, A, R, S']$ Dataset을 모아 학습함

- Q-learning에 VFA를 적용하는 것은 weight update 과정에서 sample들 사이에 correlation , Non-stationary Target으로 인해 발산 가능성 높음

⇒ Experience Replay, Fixed Q-targets로 해결

① Experience Replay

- experience Buffer에 DQN의 학습 과정에서 매 step마다 주어지는 experience $(S_t, A_t, R_{t+1}, S_{t+1})$ 저장함.
  ↳ 이 experience를 통해 Network 학습
- 기존 Neural Network의 weight 와 Policy를 한 번 update 하고 끝낼 것이 아니라 Buffer에 남아있다가 Network의 weight update에 사용됨
- mini-batch 만큼 random sampling을 거친 experience data를 추출해 weight update
  ⇒ correlation이 떨어져 Independence 확보 + 데이터 재활용 가능
- weight update 과정에서 Q-learning 방식을 사용해 Target value가 변경되었고, 이 변경된 값에 대해 새로운 독립적 data로 weight update 가능
- $\boxed{\text{loss Function} : L_i(w_i) = E_{s,a,r,s' \sim D_i}[(r + \gamma \max_{a'} Q(s',a';w_i^-) - Q(s,a;w_i))^2]}$
- using variant of SGD

② Fixed Q-targets

- Target weight를 고정함
  ⇒ Q-learning에서 Target의 변화 문제를 위해 고정하는 방식으로 대응
- Target update 과정에서는 Update할 때 사용하는 weight와 실제 Update할 weight를 나누어 사용

⇒ Q-learning에서 potential divergence 줄여 줌

- equation : $\boxed{\Delta w = \alpha[\underbrace{(R + \gamma \max_{a'} \hat{Q}(s',a,w^-)) - \hat{Q}(s,a,w)}_{\text{TD error}}]\underbrace{\nabla_w \hat{Q}(s,a,w)}_{\text{Gradient of current predicted Q-value}}}$

- 실행 과정 ⇒ Neural Network의 Hyper parameter 3정, Replay Buffer 크기 3정, Target network update 주기 변경으로 성능 개선 가능

: 용량 N을 갖는 Replay Memory 할당 → Q-function을 random weight로 초기화 → episode 진행

> Episode 내부 진행 과정
> - ε의 확률 random Action 지정, Q를 최대화하는 Action 선택
>   ↳ $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
> - Next State (S_{t+1})은 현재 state와 action, action으로 변화된 상태로 설정
> - Next policy에 대한 정보의 진행
> - Replay memory D에 t step에서의 state, action, reward, (t+1)step의 state 저장
> - D에서 random sampling으로 mini-batch size 만큼의 history를 가져와 episode가 종료되는 경우와 아닌 경우로 나누어 ground truth로 사용할 값을 지정하고, SGD를 사용해 학습

- Bellman's equation : $\boxed{\underbrace{Q(s,a)}_{\text{Q-target}} = r + \gamma \max_{a'} Q(s',a';\theta)}$

- Double DQN : DQN에서 최대 이상으로 Q-value가 계산된 경우 Q-network의 성능 저하 방지
  ⇒ 2개의 Q-network 사용 (action을 선택하는 network, Q-value를 계산하는 network)
  → Bellman's equation : $\boxed{Q(s,a;\theta) = r + \gamma Q(s', \operatorname{argmax}_{a'} Q(s',a';\theta); \theta)}$
- Dueling DQN : Q-value 계산 전에 Value function과 advantage를 계산하는 계층을 두고 둘을 나중에 합침
  ⇒ Action, value에 대한 정보를 나누어 얻으며 성능 향상
- Prioritized experience replay
  : Replay memory를 통해 ① 어떤 experience를 저장할지 ② 어떤 experience를 replay할지 선택해 우선순위 설정
  → 각 transition의 중요도를 TD error의 크기로 측정해 우선순위 선택 : TD error의 비율에 따라 가중치 update / stochastic Sampling을 통해 우선순위 설정
- Rainbow DQN : 7가지 DQN 알고리즘을 결합
  ↳ DQN, Double DQN, multi-step learning, Noisy DQN, Dueling DQN, Prioritized DQN, C51

* Policy Gradient : Value에서 Policy를 찾지 않고 θ로 weight 값을 찾아 직접 Policy 결정 (on-policy)

<Action-value method (previous)>	<Policy Gradient>
- learn the value of each action - usually pick the max value - $\hat{q}_t(s,a) = E[\sum \gamma^{k-1} R_k \mid S_t = s, A_t = a]$ - $A_t = \operatorname{argmax}_a \hat{Q}_t(S_t, a)$ / random (ε-greedy) ex. DQN	- learn the parameters of stochastic policy ⇒ no value function - update by gradient ascent in performance ex. Actor-Critic algorithms (A2C / A3C), TRPO, PPO ↳ learn both value & policy parameters

- 또한 Continuous한 action에 적용 (DQN도 DQN 적용시 Continuous한 상황에 적용 가능)

- Policy based RL의 장점 : ① Efficient in high-dimensional or continuous action spaces
  ② Better convergence properties
  ③ Can learn stochastic policy (optimal stochastic policy)
  ↳ Value-based learning은 거의 greedy한 방법이고, deterministic policy인 경우 stochastic한 환경에서 최적 정책을 찾지 못함
- 단점 : ① Typically converge to a local rather than global optimum
  ② calculating policy is sample efficient (need lots of samples), high variance
- Theorem : $\boxed{\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)]\ ,\ \nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_\theta \pi(a \mid s;\theta)}$
  ↳ Policy π를 가진 상태의 Markov chain에 대한 stationary distribution (가중치 합은 on policy state distribution)
- given policy parameterization : $\pi(a \mid s, \theta)$ / objective : $J(\theta) = V_{\pi_\theta}(S_0)$
- stochastic gradient ascent : $\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta_t)$
- Policy search : value function의 값을 계산하지 않고 action이 크게 변하는 것 방지
  → 최적의 policy π*를 reward로부터 직접 찾음
  ① gradient based optimization : policy gradient
  ② gradient free optimization
- expected return을 찾는 법
  - ① deterministic approximation : MDP의 dynamics 사용
  - ② monte carlo approximation : dynamics 사용 없이 얻은 sample들로 expected return 계산

- ↳ MC PG
- REINFORCE : $\boxed{\nabla_\theta E_{p(x;\theta)}[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(X_i)\nabla_\theta \log p(X_i;\theta)}$ ↳ gradient를 estimate 하려는 (REINFORCE 기법 이용)
  ⇒ 높은 variance를 가진다는 단점 존재 + on-policy라 샘플 효율성 낮음 ↳ Actor-Critic에 활용
- Policy gradient는 stochastic policy를 가지도록 parameter를 가진 function approximator를 이용해 근사함.
  ↳ gradient의 update 방식
- Policy의 성능을 평가할 수 있는 목표함수를 설정하고, network의 weight에 대한 gradient 계산해 Gradient Ascent로 정책 최적화 (maximize $J(\theta)$)

<REINFORCE>	<Actor-Critic>
- gradient를 episode마다 계산 - $\theta_{t+1} = \theta_t + \alpha(G_t - b(S_t))\frac{\nabla\pi(A_t \mid S_t)}{\pi(A_t \mid S_t)}$ ↳ baseline 적용 - analytical하게 gradient 계산	- gradient를 time-step마다 계산 - $\theta_{t+1} = \theta_t + \alpha\delta_t \frac{\nabla\pi(A_t \mid S_t)}{\pi(A_t \mid S_t)}$ ↳ TD error $\delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1}) - \hat{v}(S_t) \approx A_\pi(S_t,A_t)$ (TD target, value, Advantage)

* Actor - Critic

: Policy의 parameter θ를 update 하는 'Actor' + Value function을 update 하는 'Critic'

- Actor : Action을 하고 θ를 update 함 (policy update) → Critic에서 update된 policy 를 ($\pi_\theta(a \mid s)$)
- Critic : Actor로부터 받은 updated policy 평가 → Value function의 parameter w update → updated value function을 Actor에게 줌 ($Q_w(s,a) / V(s)$)
  ↳ 1-step TD error 사용
- Actor에서 State S를 input으로 받아 policy, Action을 output으로 내놓고 Critic에서 받은 Updated Value-function에 policy gradient를 취해 parameter θ update → Critic은 이를 이용해 Value function update
- MC, TD, TD(λ) 를 활용해 Policy evaluation을 통해 Q 업데이트하고, Policy gradient를 통해 조정함. 이 때 활용하는 Joint loss function : gradient 존 값을 또는 policy 업데이트
- 목적함수 $J = (R_{t+1} + Q_w(S_{t+1}, A_{t+1}) - Q_w(S_t, A_t))^2$ → REINFORCE
- Actor Update : $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a \mid s) Q_w(s,a)$
- Critic Update : $w \leftarrow w - \beta \nabla_w (R_t + \gamma Q_w(S_{t+1}, A_{t+1}) - Q_w(S_t, A_t))^2$
  ⇒ Goal 도달하기 위해 episode를 끝까지 진행하지 않고, next step에 도달을 통해 update
  ∴ low variance than REINFORCE
  Gradient ascending
- loss function : $E[Q_w \log \pi_\theta(a \mid s) Q(s,a)]$

* A2C (Advantage Actor-Critic) : Synchronous Advantage Actor-Critic

- Advantage : $\boxed{A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)} = Q_\pi(s,a) - E_{a \sim \pi(a \mid s)}[Q_\pi(s,a)]$
  → Q_π 상태에서 a 행동을 했을 때 Value function 보다 Expectation보다 얼마나 좋은지 나타냄
  ↳ 높이 작아져서 variance 감소
- → Advantage 에서 $V_\pi(s)$, $Q_\pi(s,a)$를 뽑기 위해서 advantage를 계산할 수 있도록 해서 진행
  ⇒ 두 가지 함수 추정 필요
- $\boxed{A_\pi(S_t, A_t) \approx r(S_t, A_t) + \gamma V_\pi(S_{t+1}) - V_\pi(S_t)}$: state value function (V)만 만 추정하면 됨 (neural network 이용)
- Bellman's equation : $V_\pi(S_t) = E_{A_t \sim \pi(a \mid S_t)}[r(S_t, A_t) + E_{S_{t+1} \sim p(S_{t+1} \mid S_t, A_t)}[\gamma V_\pi(S_{t+1})]]$
  → $V_\pi(S_t) \approx r(S_t, A_t) + \gamma V_\pi(S_{t+1})$
- Critic의 loss function : $\mathcal{L}_{critic}(\phi) = \frac{1}{2}\sum_i \| r(S_i, A_i) + \gamma V_\phi(S_{i+1}) - V_\phi(S_i) \|_2^2$
- Advantage : $A_\phi(S_t, A_t) \approx r(S_t, A_t) + \gamma V_\phi(S_{t+1}) - V_\phi(S_t)$
- actor의 loss function : $\mathcal{L}_{actor}(\theta) = -\sum_i (\log \pi_\theta(a_i \mid s_i) A_\phi(s_i, a_i))$

Off-Policy
Policy Gradient
④ DPG, DDPG, TRPO, PPO, SAC

- Parallel training
  → 여러 workers 가 각 environment와 interact해 reward와 다음 state 받아옴
  → 이 reward 와 state를 Master node의 Actor-Critic에 적용 Update 후 Updated value, Updated policy를 Worker에 보내줌
  → 이 policy 와 Value를 기반으로 Worker는 다음 Action 진행
- Alphago 가 활용한 방법

* A3C (Asynchronous Advantage Actor-Critic)

- Actor-Critic approach
  - Actor : optimize θ parameter of policy
  - Critic : optimize w parameters of value function
- Network : use DNN to model policy and value function (DQN과 유사)
- do not use experience replay
  → have many agents working in different state-action space
  ↳ training에 diversity 제공
- use fixed period of time from training to compute advantage function
- A3C agent 여러개는 독립적인 환경에서 진행하면서 global network와 학습 ↳ 동시에 여러 agent가 진행되므로 학습 느림

<DQN>	<Actor-Critic>	<Policy Gradient>
- Replay Buffer 사용 - Network 1개 - Q(s,a)를 학습	- Replay Buffer 사용 X (매 step마다 얻은 S_t, A_t, r, S_{t+1} 이용해 학습) - Actor Network , Critic Network 의 2개 사용 - π(s,a) , V(s) 학습 (이 때 π(s,a)는 softmax 사용해서 얻음) ↳ 안정성 보장 (PG에 비해) ex. A2C, A3C	- π(s,a)를 직접 학습함 (non stable) ex. REINFORCE

* Trust Region Policy Optimization (TRPO)

- policy gradient 에 문제점 step size를 너무 크게 하면 reward에 악영향 미침, 너무 작으면 training 느림
  ↳ step-size에 따라 불안정 또는 slowly 함
⇒ New policy를 Improved performance로 update 한다고 믿을 수 있는 지역 (trust-region) 에서만 update

* Proximal Policy Optimization (PPO)

- TRPO에서 Approximation으로 인한 Constraint를 더 간결하게 정리함
⇒ clipping 사용

* Values vs Policy

- Q(s,a) 에서는 $\pi_w(s,a)$ 추출 가능 ⇒ greedy policy $\operatorname{argmax}_a Q(s,a)$
  ↳ Value function
- $\pi_w(s,a)$에서는 Q(s,a) 추출 불가

* Types of Action

	Reccurent	Box	Discrete
DQN	X	X	O
A2C	O	O	O
PPO1	X	O	O
PPO2	O	O	O
TRPO	X	O	O

* Regularization

: model에 penalty를 주어 perfect fit을 포기하여 (training accuracy ↓) testing accuracy 를 증가시킴

- 너무 높은 complexity를 피하기 위해 사용

ex. drop out, early stopping, weight decay
→ 전부 training 시에 사용 ↳ 훈련데이터 감소

⇒ Loss function에 항을 추가하는 것을 목표로 Overfit을 줄이지만, 특정 parameter를 일정 크기 만드는 방식 넣어 작용

* Hyper-parameter

: Model의 training 과 연관된 parameter로 problem과는 연관 X ⟷ parameter : training에 사용 problem과 연관

- # of hidden layer : 모델의 accuracy 와 연관되어 Underfitting, Overfitting에 영향
- Drop out : overfitting을 피하기 위한 방법, hidden layer의 일정 node 비활. 값이 너무 크면 underfitting 가능
- learning rate (α)
- Momentum : SGD에 대한 optimization 기법
- epoch : 전체 dataset을 한번 학습
- Batch size : Data 여러 시스템의 부담을 줄이기 위해 Batch size로 나눔

* High level RL

- Multi-agent RL : 2개 이상의 agent가 주어진 환경에서 협업 또는 경쟁을 통해 높은 보상을 얻을 수 있는 policy 학습
- Hierarchical RL : 에이전트 행동을 계층 나눠서 해결함 (sparse reward / delayed reward 문제 해결)
- Supervised : label이 있는 data 학습 / Unsupervised : label 없이 data의 범주 만드는 학습 방법 / Self-Supervised : label 없이 스스로 분류하여 데이터 범주 만듦 ↳ Unsupervised의 범주
- Curiosity based learning : Policy network 와 Curiosity driven 통 Intrinsic reward를 설정해 두면서 network 구성

* Pseudo code

① REINFORCE with Baseline (episodic)

Input: a differentiable policy parameterization $\pi(a|s;\theta)$
Input: a differentiable state-value parameterization $\hat{v}(s,w)$
Parameter: step size $\alpha > 0$, $\beta > 0$
Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^{d}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \cdots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\theta)$
  For each step of the episode $t = 0, \cdots, T-1$:
    $G_t \leftarrow$ return from step t
    $\delta \leftarrow G_t - \hat{v}(S_t, w)$
    $w \leftarrow w + \beta\gamma^t \delta \nabla_w \hat{v}(S_t, w)$
    $\theta \leftarrow \theta + \alpha\gamma^t \delta \nabla_\theta \ln \pi(A_t|S_t;\theta)$
    $= \frac{\nabla_\theta \pi(a|s,\theta)}{\pi(a|s,\theta)}$

Policy Gradient

② REINFORCE, MCPG (episodic)

Input: a differentiable policy parameterization $\pi(a|s;\theta), \forall a \in A, s \in S, \theta \in \mathbb{R}^n$
Initialize policy weights θ
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \cdots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\theta)$
  For each step of the episode $t = 0, \cdots, T-1$:
    $G_t \leftarrow$ return from step t
    $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \ln \pi(A_t|S_t,\theta)$

③ DQN (Deep Q-learning) with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ do
  Initialize 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 D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from 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 $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

value based

④ One-Step Actor-Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\theta), \forall a \in A, s \in S, \theta \in \mathbb{R}^n$
Input: a differentiable state-value parameterization $\hat{v}(s,w), \forall s \in S, w \in \mathbb{R}^m$
Parameters: step sizes $\alpha > 0$, $\beta > 0$
Initialize policy weights θ and state-value weights w
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\theta)$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S', w) - \hat{v}(S, w)$ (if S' is terminal, then $\hat{v}(S,w) \doteq 0$)
    $w \leftarrow w + \beta\delta\nabla_w \hat{v}(S, w)$
    $\theta \leftarrow \theta + \alpha I \delta \nabla_\theta \ln \pi(A|S,\theta)$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

⑤ A2C (Advantage Actor-Critic)

Assume parameter vectors θ and θ_v
Initialize step counter $t \leftarrow 1$
Initialize episode counter $E \leftarrow 1$
repeat:
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
  $t_{start} = t$
  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}
    $t \leftarrow t+1$
  until terminal s_t or $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta_v) & \text{for non-terminal } s_t \text{ // bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \cdots, t_{start}\}$ do
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ: $d\theta \leftarrow d\theta + \nabla_\theta \log \pi(a_i|s_i;\theta)(R - V(s_i;\theta_v)) + \beta_e \partial H(\pi(a_i|s_i;\theta))/\partial\theta$
    Accumulate gradients wrt θ: $d\theta_v \leftarrow d\theta_v + \beta_v (R - V(s_i;\theta_v))(\partial V(s_i;\theta_v)/\partial\theta_v)$
  end for
  perform update of θ using $d\theta$ and of θ_v using $d\theta_v$
  $E \leftarrow E+1$
until $E > E_{max}$

⑥ A3C (Asynchronous Advantage Actor-Critic)

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 $d\theta_v \leftarrow 0$
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  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 $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{ // bootstrap last state} \end{cases}$
  for $i \in \{t-1, \cdots, t_{start}\}$ do
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i;\theta'_v))^2 / \partial\theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$
until $T > T_{max}$

off-policy
Policy Gradient

⑦ SAC (Soft Actor-Critic)

Input: the learning rates, λ_π, λ_Q and λ_V for functions π_θ, Q_w and V_ψ respectively;
Input: The weighting factor τ for exponential moving average
Initialize parameters θ, w, ψ and $\bar{\psi}$.
for each iteration do
  for each environment setup do
    $a_t \sim \pi_\theta(a_t|s_t)$
    $s_{t+1} \sim p_\pi(s_{t+1}|s_t, a_t)$
    $D \leftarrow D \cup \{(s_t, a_t, r(s_t,a_t), s_{t+1})\}$
  for each gradient update step do
    $\psi \leftarrow \psi - \lambda_V \nabla_\psi J_V(\psi)$
    $w \leftarrow w - \lambda_Q \nabla_w J_Q(w)$
    $\theta \leftarrow \theta - \lambda_\pi \nabla_\theta J_\pi(\theta)$
    $\bar{\psi} \leftarrow \tau\psi + (1-\tau)\bar{\psi}$

⑧ TRPO (Trust Region Policy Optimization)

Initialize π_θ to anything
Repeat forever:
  Sample s_0 and set $n \leftarrow 0$
  Repeat N times:
    Sample $a_n \sim \pi_\theta(a|s_n)$
    Execute a_n, observe s_{n+1}, r_n
    $\delta \leftarrow r_n + \gamma \max_{a_{n+1}} Q_w(s_{n+1}, a_{n+1}) - Q_w(s_n, a_n)$
    $A(s_n, a_n) \leftarrow r_n + \gamma \max_{a_{n+1}} Q_w(s_{n+1}, a_{n+1}) - \sum_a \pi_\theta(a|s_n) Q_w(s_n, a)$
    Update Q: $w \leftarrow w + \alpha_w \gamma^n \delta \nabla_w Q_w(s_n, a_n)$
    $n \leftarrow n+1$
  Update π: $\theta \leftarrow \arg\max_{\theta} \frac{1}{N}\sum_{n=0}^{N-1} \frac{\pi_\theta(a_n|s_n)}{\pi_{\theta}(a_n|s_n)} A_\theta(s_n|a_n)$
    Subject to $\frac{1}{N}\sum_{n=0}^{N-1} D_{KL}(\pi_\theta(\cdot|s_n), \pi_\theta(\cdot|s_n)) \leq \delta$

⑨ PPO (Proximal Policy Optimization)

Input: initial policy parameters θ_0, initial function parameters ϕ_0
for $k = 0, 1, 2 \cdots$ do
  Collect set of trajectories $D_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment
  Compute rewards to go $\hat{R}_t$
  Compute Advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on current value function V_{ϕ_k}
  Update the policy by maximizing PPO objective:
  $\theta_{k+1} = \arg\max_\theta \frac{1}{|D_k| T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t))\right)$
  Fit value function by MSE: $\phi_{k+1} = \arg\min_\phi \frac{1}{|D_k| T} \sum_{\tau \in D_k} \sum_{t=0}^{T} (V_\phi(s_t) - \hat{R}_t)^2$
end for

* MCTS (Monte Carlo Tree Search)
: Alphago의 게임 탐색 알고리즘
- 모든 tree node 대상으로 하는 대신 게임 시뮬레이션을 통해 가장 가능성이 높아 보이는 방향으로 행동 결정
- Min-Max 알고리즘의 성능을 개선한 것으로, 모든 경로를 탐색하는 것이 불가능할 때 효과적

* AlphaGo
: MCTS를 deep learning pipeline을 통해 훨씬 성능을 개선한 것
- 3개의 network 가정 (SL, RL의 policy network, value network)

* AlphaZero
: Alphago는 바둑에 한정이 되고, Alphazero는 체스에 한정이 없음

* Q&A of DQN
① A DQN algorithm has 2 networks, a target network and a Q network (T)
② A Double DQN algorithm has 2 networks, a target network and a Q network (F)
→ multiple copies of each network is needed
③ A DQN algorithm utilizes replay buffer (T)
④ The only difference between DQN and Q learning is that DQN utilizes value function approximation with no additional enhancements (ex. replay buffer) (F)
→ DQN has 2 networks
⑤ A Double DQN algorithm utilizes replay buffer (T)

*Midterm

#1 Multi-armed Bandits are simply more efficient method for Reinforcement Learning that can be used for small domains. T/F (F)
→ MABs have no state so are inherently more limited than full RL. The domains for MABs are simpler but not just because they are small in state or action space.

#2 Any time an agent takes an action $b \neq \max_a Q(s,a)$, it could be considered exploration. T/F (T)

#3 Exploration actions only result from the use of an ϵ-greedy policy. T/F (F)
→ one example is ϵ-greedy but it could be any other function of the value function.

#4 To get the value function from a Q function, we need to get the average Q value over all actions at a given state. T/F (F)
→ We need to get the max of Q value for all states.

#5 An example of an optimistic action selection strategy would be to take an action which has never been tried for a given state s. T/F (T)
→ Because it assumes unvisited pair (s,a) are good.

#6 The "Target" in a TD method is provided as input to the agent, it is the pre-defined goal you seeking to find. T/F (F)
→ The target is the new immediate reward combined with the estimate of return going forward with the current policy.

#7 The expected SARSA algorithm attempts to use information from all actions by taking the mean over all actions of the value function for the current state. T/F (F)
→ The algorithm takes expected value of all actions, this would only be equal to the mean if all actions had the same probability. The value of each (s,a) pair is weighted by the probability that the action a will be selected by the current policy while s.

#8 Which of the following best describe the meaning of the values stored in a state-value function or state-action value function?
(a) an estimate of expected reward (b) an estimate of discounted rewards (c) a function of rewards resulting from following the current policy into the future ✓(d) all of them

#9 A Monte Carlo RL method is one that
(a) Samples randomly from trajectory of (s,a,r) tuples and uses the average weighted return from those samples.
✓(b) uses the complete return from an entire trajectory which is sampled randomly from the current policy
(c) uses any kind of random sampling policy in its policy.

#10 Which one of the following parameters was introduced to help the Credit Assignment Problem?
✓(a) λ in TD(λ) (b) γ in DP/MC/TD methods (c) ϵ in greedy policy definition (d) α in TD methods

#11 Which one of the following parameters was introduced to encourage more exploration during RL?
(a) λ in TD(λ) (b) γ in DP/MC/TD methods ✓(c) ϵ in greedy policy definition (d) α in TD methods

#12 Which one of the following parameters was introduced to control the influence new information has on the value function update?
(a) λ in TD(λ) (b) γ in DP/MC/TD methods (c) ϵ in greedy policy definition ✓(d) α in TD methods

#13 Which one of the following parameters was introduced to control the importance of receiving reward sooner in episode, rather than later?
(a) λ in TD(λ) ✓(b) γ in DP/MC/TD methods (c) ϵ in greedy policy definition (d) α in TD methods

#14 The main difference between solving an MDP with Dynamic Programming and using Reinforcement Learning is:
(a) In RL we don't need to find an exact solution
✓(b) In RL, the dynamics are not available
(c) when we solve an MDP, we don't discount future rewards
(d) In RL, we use the policy improvement theorem to greedily select actions based on our value estimate.

#15 Which of the following calculations could give different answers depending on the starting conditions of the value function:
(a) Exact solution using the value iteration Algorithm
(b) Exact solution using the policy iteration Algorithm
(c) Converged solution of SARSA (tabular representation)
✓(d) Converged solution of SARSA (using linear value function approximation)
✓(e) Converged solution of Q-Learning (tabular representation)

#16 Let the general form of the return for an MDP from timestep t going forward until the terminal state is reached at t=T be defined recursively for all non-terminal steps as:
$G_t = R_{t+1} + \gamma G_{t+1}$
Suppose we are given a sequence of rewards actually received from a policy on an MDP as follows: $R_1=2, R_2=-1, R_3=5, R_4=1, R_5=-1$
The MDP has horizon of 5 and we use a discount factor of 0.9. Once the terminal state is reached, there are no rewards received and the episode ends.
① Write one-step, recursive calculation for each return below in symbols and show your calculation.
$G_0 = R_1 + \gamma G_1 = 2 + 0.9 \times 3.581 = 5.2229$
$G_1 = R_2 + \gamma G_2 = -1 + 0.9 \times 5.09 = 3.581$
$G_2 = R_3 + \gamma G_3 = 5 + 0.9 \times 0.1 = 5.09$
$G_3 = R_4 + \gamma G_4 = 1 + 0.9 \times (-1) = 0.1$
$G_4 = R_5 + \gamma G_5 = -1 + 0.9 \times 0 = -1$ ($\because G_5$ = terminal state $\therefore 0$)
② Write out the full, non-recursive calculation of G_0 in symbols and numbers for the whole domain from the start state.
$G_0 = R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \gamma^4 R_5$
$= 2 + (0.9 \times -1) + (0.9^2 \times 5) + (0.9^3 \times 1) + (0.9^4 \times -1) = 5.2229$

#17 SARSA Value function Update
Consider a system with 3 states and 2 actions. You will perform update calculations to a tabular state-action value function on an MDP using a learning rate $\alpha = 0.15$, and a discount factor $\gamma = 0.9$ for each step. For each step, the current state, action, reward, and next state and action are given as (s_1, a_1, r, s_2, a_2). In this MDP, rewards are obtained when the agent enters the state after taking the action.
The pre-existing state-action value function, $Q(s,a)$ is defined by the following table:

Q(S,A)	a_1	a_2
s_1	5	-1
s_2	-2	2
s_3	-1	3

Show the calculations for the following updates using the provided trajectory of experiences using the SARSA algorithm:
① General Formula for updating $Q(s,a)$ using s, a, r, s', a':
$Q(s,a) = Q(s,a) + \alpha[r_t + \gamma Q(s',a') - Q(s,a)]$
② $(s_1, a_1, 2, s_2, a_2)$, $Q(s_1,a_1)$
$Q(s_1,a_1) = Q(s_1,a_1) + \alpha[r + \gamma Q(s_2,a_2) - Q(s_1,a_1)] = 5 + 0.15 \times (2 + 0.9 \times 2 - 5) = 4.82$
③ $(s_2, a_2, -1, s_1, a_1)$, $Q(s_2,a_2)$
$Q(s_2,a_2) = Q(s_2,a_2) + \alpha[r + \gamma Q(s_1,a_1) - Q(s_2,a_2)] = 2 + 0.15 \times (-1 + 0.9 \times 4.82 - 2) = 2.2$
④ $(s_1, a_1, 2, s_2, a_1)$, $Q(s_1,a_1)$
$Q(s_1,a_1) = Q(s_1,a_1) + \alpha[r + \gamma Q(s_2,a_1) - Q(s_1,a_1)] = 4.82 + 0.15 \times (2 + 0.9 \times -2 - 4.82) = 4.13$
⑤ $(s_2, a_1, 1, s_3, a_1)$, $Q(s_2,a_2)$
$Q(s_2,a_2) = Q(s_2,a_2) + \alpha[r + \gamma Q(s_3,a_1) - Q(s_2,a_2)] = -2 + 0.15 \times (1 + 0.9 \times -1 + 2) = -1.69$

#18 Expected SARSA

Now, in #17, we were instead using Expected SARSA and we knew the policy is ε-greedy but biased towards a_1, or more precisely:

- For any state s, if $a_1 = \underset{a}{\operatorname{argmax}} Q(s,a)$ then $\pi(a_1|s) = 0.9$
- For any state s, if $a_2 = \underset{a}{\operatorname{argmax}} Q(s,a)$ then $\pi(a_2|s) = 0.6$

Using the same MDP before, show the formula and just the first value update calculation for Expected SARSA:

① General formula for updating $Q(s,a)$ using s, a, r, s', a':

$$Q(s,a) = Q(s,a) + \alpha[r_k + \gamma[\sum_{a'} \pi(a'|s') Q(s',a')] - Q(s,a)]$$

② $(s_1, a_1, 2, s_2, a_2)$, $Q(s_1, a_1)$

$$Q(s_1,a_1) = Q(s_1,a_1) + \alpha[r + \gamma[(1-\pi(a_2|s_2))Q(s_2,a_1) + \pi(a_2|s_2)Q(s_2,a_2)] - Q(s_1,a_1)]$$

$$= 5 + 0.15(2 + 0.9 \times (0.4 \times -2 + 0.6 \times 2) - 5) = 4.6$$