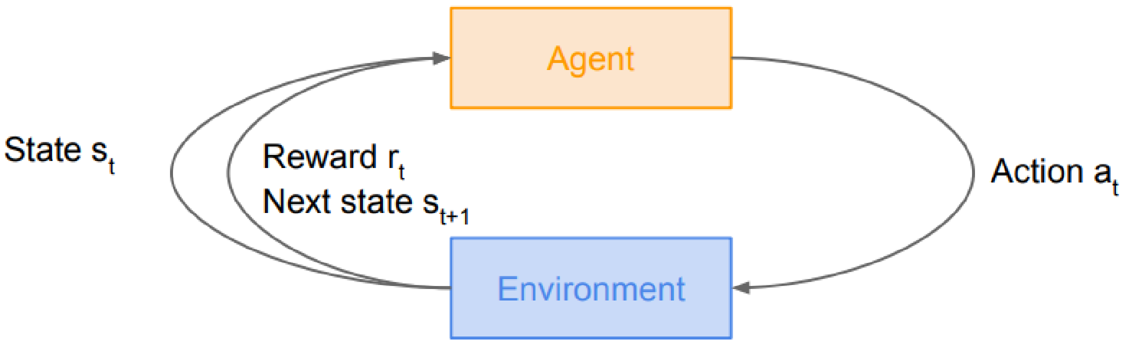


강화학습은 Env(Environment) 와 Agent 간의 상호작용에서 학습이 이루어짐
Agent 는 초기 상태에서 특정 Aciton 을 하면, reward 와 다음 state 를 얻게 되고, 다음 action 을 하게 됨.

Markov Decision Process

Markov Property: 이전 state 와 상관 없이, 과거와 미래 state 는 현재 state 와 완전히 independent 하고, 현재 state 에서 다음 state 로 갈 확률은 항상 같다 라는 성질.

- state
    - agent 가 관찰 가능한 상태의 집합.
- action
    - agent 가 특정 state 에서 행동할 수 있는 action 의 집합
    - 예) 2 차원 grid world 라면, 상 하 좌 우 이동
- reward
    - (state, action) pair 에 따라 env 가 agent 에게 주는 유일한 정보
- state transition probability
    - (state, action) pair 에 의해 agent 가 특정 state 로 변경 되야 했지만, env 에 의해 다른 state 로 변경될 확률
- discount factor
    - agent 가 받는 reward 중, 현재에 가까운 reward 를 더 비싸게, 현재에 먼 reward 를 더 싸게 해주는 factor.
    - 당장 현재에 있는 reward 가 더 비싸다.

Markov Decision Process

- At time step t=0, environment samples initial state $s_0 \sim p(s_0)$
- Then, for t=0 until done:
    - Agent selects action a_t
    - Environment samples reward $r_t \sim R(. | s_t, a_t)$
    - Environment samples next state $s_{t+1} \sim P(. | s_t, a_t)$
    - Agent receives reward r_t and next state s_{t+1}

- A policy **π** is a function from S to A that specifies what action to take in each state
- **Objective**: find policy **π*** that maximizes cumulative discounted reward: $\sum_{t>0} \gamma^t r_t$

MDP 는 다음과 같이 학습하게된다.

- t=0 인 first step, env 는 S_0 를 sampling 하여 initializing 한다.

- 그 후, 해당 game 이 끝날때 까지 반복한다
  - agent 가 Action_t 를 고른다
  - env 가 (S_t, A_t) pair 에 state transition probability 를 적용하여 reward 를 sampling 한다.
  - env 는 state_t+1 를 sampling 한다.
  - agent 는 env 로 부터 reward_t, state_t+1 를 받는다.

여기서 말하는 policy Pi 는, 특정 state 에서 agent 가 어떤 action 을 결정할지에 대한 일종의 판단 함수 이다.

Input State -> policy function -> Output action

결국, 우리가 학습해야 하는것은 이 policy pi 인것이고, 감가율을 적용하여 agent 가 얻을 전체 reward 를 더했을때 가장 큰 reward 의 합($\sum_{t>0} \gamma^t r_t$)을 주는 action 을 고르게 학습해야한다.

Value function and Q-funciton

특정 Policy pi 는 흔적(path?) 를 남기게 된다(s, a ,r 의 조합으로)

Value function 은 이 흔적들을 기반으로 미래에 받을 reward 들의 기댓값을 구한다.

Following a policy produces sample trajectories (or paths) $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$

How good is a state?

The **value function** at state s, is the expected cumulative reward from following the policy from state s:

$$V^\pi(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

How good is a state-action pair?

The **Q-value function** at state s and action a, is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^\pi(s,a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

이때, 현재의 reward 가 미래의 reward 보다 중요하므로,

미래의 reward 에 대해서는 감가율이 적용된다.

이 식을 Bellman Expectation Equation, 벨만 기대 방정식이라고 한다.

벨만 기대 방정식은 현재 상태의 value function 과 다음 상태의 value function 간의 관계를 말해주는 방정식이다.

Q-value function 은 위의 벨만 기대 방정식에 action 이라는 factor 가 추가된 버전이다. state s 에서 action a 를 행동했을때, reward 의 기댓값을 나타낸다.

Bellman equation

Bellman equation

The optimal Q-value function Q* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Q* satisfies the following **Bellman equation**:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s, a\right]$$

Intuition: if the optimal state-action values for the next time-step Q*(s',a') are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s',a')$

The optimal policy **π*** corresponds to taking the best action in any state as specified by Q*

optimal Q-function 을 얻기 위해서는 reward 의 기댓값이 최대가 되야함.

Q'-funciton 을 분해해보면, 지금 당장 얻을 reward 와 미래에 얻을 reward 로 나눌 수 있음.

현재 상태의 value function 과 다음 상태의 value function 간의 관계를 나타낼 수 있음.

이 방정식을 Bellman expectation equation 이라함.

Solving for the optimal policy

Value iteration algorithm: Use Bellman equation as an iterative update

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

Q_i will converge to Q* as i -> infinity

What's the problem with this?
Not scalable. Must compute Q(s,a) for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solution: use a function approximator to estimate Q(s,a). E.g. a neural network!

i 를 무한대로 발산 시킨다면, 위 식이 Q' 에 수렴하게 된다.

하지만 너무 무거움.

각 state-action pair 마다 이 연산을 수행한다 생각하면, 거의 brute force 수준.

위와 같이 복잡한 function 들은 대부분 neural network 를 통해서 근사시킨다. 다음 장 참고.

Q-learning

Solving for the optimal policy: Q-learning

Q-learning: Use a function approximator to estimate the action-value function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

function parameters (weights)

If the function approximator is a deep neural network => **deep q-learning**!

Q-function 을 근사시키기 위해, deep neural network 를 사용해보자.
우리의 목표는, bellman equation 을 만족하는 Q-funciton 을 찾는 것.

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s',a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)} \left[(y_i - Q(s,a;\theta_i))^2 \right]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) | s, a \right]$

Backward Pass

Gradient update (with respect to Q-function parameters θ):

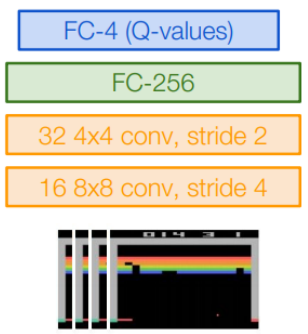
$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i) \right) \nabla_{\theta_i} Q(s,a;\theta_i) \right]$$

loss function 에서 target function 과 현재 구한 q-function 의 L2 loss 를 통해 학습하게 됨..
근데 여기서 target function 이 Bellman equation.

Q-network Architecture

$Q(s, a; \theta)$:
neural network with weights θ

A single feedforward pass to compute Q-values for all actions from the current state => efficient!



Last FC layer has 4-d output (if 4 actions), corresponding to $Q(s_t, a_1)$, $Q(s_t, a_2)$, $Q(s_t, a_3)$, $Q(s_t, a_4)$

Number of actions between 4-18 depending on Atari game

Current state s_t: 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

게임 내 pixel 정보를 전처리해서 가져와서, 4 픽셀을 묶어 (-1, -1, 4) shape 의 input 을 forward 시킨다.
마지막 FC layer 의 output 은 action 의 수 만큼 나온다.
한 번의 Forward 를 통해서 현재 state 의 모든 action 에 대한 Q-value 가 계산되게 network 를 학습시킨다.
이를 위해 위에서 나왔던 target function 인 bellman equation 과 비슷해지도록 학습시켜야한다.

Experience Replay

학습이 진행되면서 얻은 새로운 experience 들을 곧바로 다시 학습에 사용하지 않고, 특정 memory 에 저장한 후, mini batch 형태로 random sampling 하여 학습한다.

network 가 그저 특정 action 이후에 어떤 action 을 할지 예측하는 model 이 될 수 있기 때문에 데이터간의 상관관계를 깨는 작업이 필요했고,

그 작업을 위해 새로운 experience 들을 mini batch 형태에서 random sampling 하여 사용함.

Policy Gradient

지금까지 학습한 Q-learning 은 DNN 을 통해 Q-function 을 target 에 근사 시켜 나온 Q-function 을 통해 Policy 를 얻은 것.

하지만 high-demension state 이라면, 각 state-action pair 에 대한 최적의 Policy 를 찾기는 매우 어려울 것.

그럼 DNN 을 통해서 최적의 Policy 를 찾으면 어떨까? 에서 시작한것이 바로 Policy Gradient.

Policy Gradients

Formally, let's define a class of parametrized policies: $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$

For each policy, define its value:

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi_\theta\right]$$

We want to find the optimal policy $\theta^* = \arg\max_\theta J(\theta)$

How can we do this?

Gradient ascent on policy parameters!

각 정책에 대해서, 최적의 reward 를 얻는 policy 를 얻으면 될 것.

여기서 세타가 바로 neural network 의 weight. 이 weight 를 잘 업데이트 시켜서 목적함수인 J 가 최대값이 되면 될 것. gradient descent(ascent) 를 사용.

REINFORCE algorithm

Mathematically, we can write:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau\sim p(\tau;\theta)}\left[r(\tau)\right] \\ &= \int_\tau r(\tau)p(\tau;\theta)\mathrm{d}\tau \end{aligned}$$

Where r(τ) is the reward of a trajectory $\tau = (s_0, a_0, r_0, s_1, \dots)$

$$\nabla_\theta J(\theta) = \int_\tau r(\tau)\nabla_\theta p(\tau;\theta)\mathrm{d}\tau$$

Intractable! Gradient of an expectation is problematic when p depends on θ

However, we can use a nice trick: $\nabla_\theta p(\tau;\theta) = p(\tau;\theta)\frac{\nabla_\theta p(\tau;\theta)}{p(\tau;\theta)} = p(\tau;\theta)\nabla_\theta \log p(\tau;\theta)$
If we inject this back:

$$\begin{aligned}\nabla_\theta J(\theta) &= \int_\tau \left(r(\tau)\nabla_\theta \log p(\tau;\theta)\right) p(\tau;\theta)\mathrm{d}\tau \\ &= \mathbb{E}_{\tau\sim p(\tau;\theta)}\left[r(\tau)\nabla_\theta \log p(\tau;\theta)\right]\end{aligned}$$

Can estimate with
Monte Carlo sampling

REINFORCE algorithm

$$\begin{aligned}\nabla_\theta J(\theta) &= \int_\tau \left(r(\tau)\nabla_\theta \log p(\tau;\theta)\right) p(\tau;\theta)\mathrm{d}\tau \\ &= \mathbb{E}_{\tau\sim p(\tau;\theta)}\left[r(\tau)\nabla_\theta \log p(\tau;\theta)\right]\end{aligned}$$

Can we compute those quantities without knowing the transition probabilities?

We have: $p(\tau;\theta) = \prod_{t\geq 0} p(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$

Thus: $\log p(\tau;\theta) = \sum_{t\geq 0} \log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t)$

And when differentiating: $\nabla_\theta \log p(\tau;\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t)$

Doesn't depend on
transition probabilities!

Therefore when sampling a trajectory τ, we can estimate J(θ) with

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} r(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

Intuition

Gradient estimator: $\nabla_\theta J(\theta) \approx \sum_{t\geq 0} r(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)$

Interpretation:

- If r(τ) is high, push up the probabilities of the actions seen
- If r(τ) is low, push down the probabilities of the actions seen

Might seem simplistic to say that if a trajectory is good then all its actions were good. But in expectation, it averages out!

However, this also suffers from high variance because credit assignment is really hard. Can we help the estimator?

문제점 : 분산이 매우 높음.
한 경로가 좋은 보상을 준다 하더라도, 내부 action 들이 하나 하나가 다 좋은 aciton 인 것은 아닐것이다. 평균에 매몰된 결과가 나올 수 있다.
구체적으로 어떤 행동이 좋았는지 알 수 있는 길이 없고, 이 단계에서는 결국 수많은 sampling 을 통해 해결.

Variance reduction

Policy gradient 에서 gradient 를 구하기 위해 sampling 하는 과정에서 큰 분산이 큰 문제가 되었다.
조금 더 적은 샘플링으로 낮은 분산을 얻기 위한 방법들을 배워보자.

Variance reduction

Gradient estimator: $$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

First idea: Push up probabilities of an action seen, only by the cumulative future reward from that state

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} r_{t'} \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Second idea: Use discount factor γ to ignore delayed effects

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

첫 번째 아이디어는, 미래의 reward 만 사용하는 것. 경로 내 에서 모든 reward 를 사용하는것이 아닌, 현재의 time step 에서 부터 종료 시점까지 얻을 수 있는 보상의 합을 사용하는 것.
이 방법은 특정 행동이 발생했을 때 미래의 보상이 얼마나 큰지를 고려하겠다는 의도.

두 번째는 감가율을 적용 시키는 것.

Variance reduction: Baseline

Problem: The raw value of a trajectory isn't necessarily meaningful. For example, if rewards are all positive, you keep pushing up probabilities of actions.

What is important then? Whether a reward is better or worse than what you expect to get

Idea: Introduce a baseline function dependent on the state.
Concretely, estimator is now:

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

세 번째 방식은 Baseline , reward 의 기준을 0 으로 두는게 아닌, 우리가 설정하는 임의의 baseline 을 중심으로 reward 가 baseline 보다 크면 양수, 아니면 음수로 처리하는 것.

How to choose the baseline?

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} \left(\sum_{t'\geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

A simple baseline: constant moving average of rewards experienced so far from all trajectories

Variance reduction techniques seen so far are typically used in "Vanilla REINFORCE"

Actor-Critic Algorithm

Initialize policy parameters θ, critic parameters ϕ
For iteration=1, 2 … **do**
  Sample m trajectories under the current policy
  $\Delta\theta \leftarrow 0$
  For i=1, …, m **do**
    For t=1, ... , T **do**

$$A_t = \sum_{t'\geq t} \gamma^{t'-t} r_t^i - V_\phi(s_t^i)$$

$$\Delta\theta \leftarrow \Delta\theta + A_t \nabla_\theta \log(a_t^i|s_t^i)$$

$$\Delta\phi \leftarrow \sum_i \sum_t \nabla_\phi ||A_t^i||^2$$

$$\theta \leftarrow \alpha\Delta\theta$$

$$\phi \leftarrow \beta\Delta\phi$$

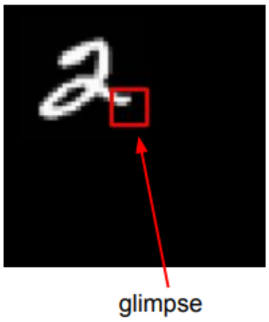
End for

REINFORCE in action: Recurrent Attention Model (RAM)

Objective: Image Classification

Take a sequence of "glimpses" selectively focusing on regions of the image, to predict class
- Inspiration from human perception and eye movements
- Saves computational resources => scalability
- Able to ignore clutter / irrelevant parts of image

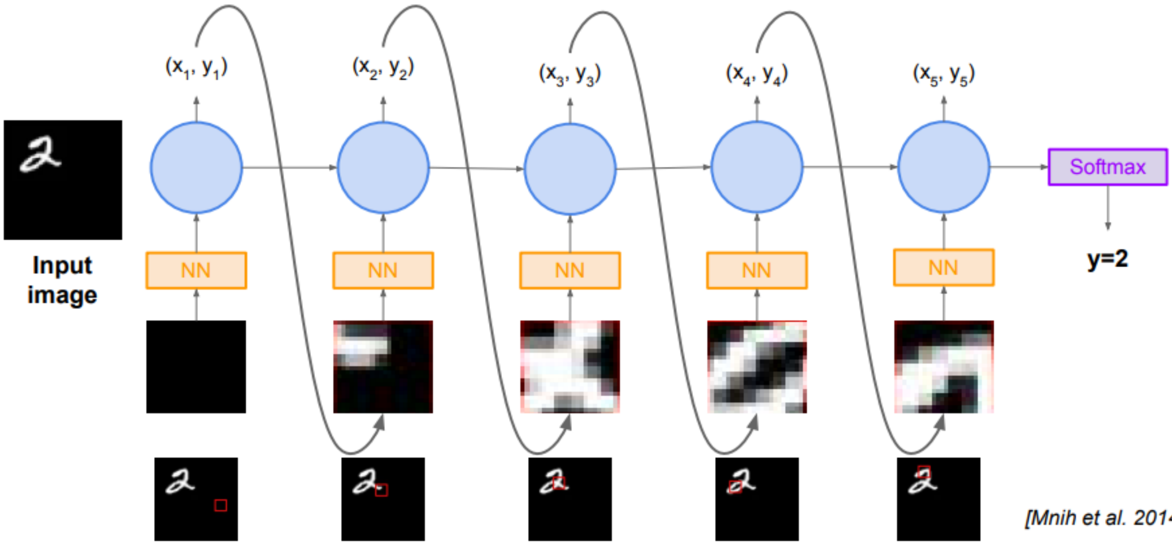
State: Glimpses seen so far
Action: (x,y) coordinates (center of glimpse) of where to look next in image
Reward: 1 at the final timestep if image correctly classified, 0 otherwise



Glimpsing is a non-differentiable operation => learn policy for how to take glimpse actions using REINFORCE
Given state of glimpses seen so far, use RNN to model the state and output next action

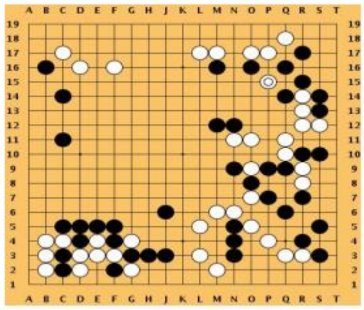
[Mnih et al. 2014]

REINFORCE in action: Recurrent Attention Model (RAM)



[Mnih et al. 2014]

More policy gradients: AlphaGo



Overview:

- Mix of supervised learning and reinforcement learning
- Mix of old methods (Monte Carlo Tree Search) and recent ones (deep RL)

How to beat the Go world champion:

- Featurize the board (stone color, move legality, bias, …)
- Initialize policy network with supervised training from professional go games, then continue training using policy gradient (play against itself from random previous iterations, +1 / -1 reward for winning / losing)
- Also learn value network (critic)
- Finally, combine combine policy and value networks in a Monte Carlo Tree Search algorithm to select actions by lookahead search

[Silver et al., Nature 2016]