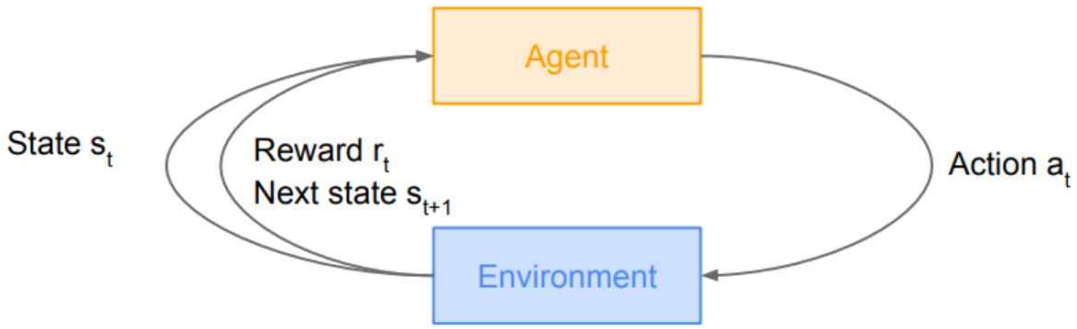
cs231n lecture 14 신지연

1. Reinforcement Learning




강화학습은 Env(Environment) 와 Agent 간의 상호작용에서 학습

Agent 는 초기 상태에서 특정 Aciton을 하면, reward와 다음 state 를 얻게 되고, 다음 action 을 하게 됨

2. Markov Decision Process

Markov Property: 이전 state와 상관없이, 과거와 미래 state는 현재 state와 완전히 independent하고, 현재 state에서 다음 state로 갈 확률은 항상 같다는 성질.

## Markov Decision Process

- Mathematical formulation of the RL problem
- **Markov property**: Current state completely characterises the state of the world

Defined by: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

$\mathcal{S}$ : set of possible states
$\mathcal{A}$ : set of possible actions
$\mathcal{R}$ : distribution of reward given (state, action) pair
$\mathbb{P}$ : transition probability i.e. distribution over next state given (state, action) pair
$\gamma$ : discount factor

- state
  : agent 가 관찰 가능한 상태의 집합.
  ; 예) 2차원 grid world 라면, 가능한 모든 (x, y) 좌표
- action
  : agent 가 특정 state 에서 행동할 수 있는 action의 집합
  : 예) 2차원 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 되어야 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 $r_t \sim R( . \mid s_t, a_t)$
  - Environment samples next state $s_{t+1} \sim P( . \mid s_t, a_t)$
  - 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.

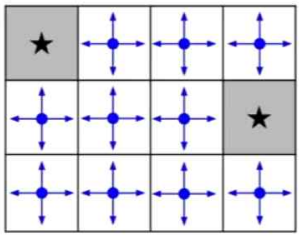
t=0 인 first step, env 는 S_0 를 sampling 하여 initializing 한다.
그 후, 해당 game 이 끝날때 까지 반복한다
agent 가 Action_t 를 고른다
env 가 (S_t, A_t) pair 에 state transition probability 를 적용하여 reward 를 sampling 한다.
env 는 state_t+1 를 sampling 한다.
agent 는 env 로 부터 reward_t, state_t+1 를 받는다.
여기서 말하는 policy Pi 는, 특정 state 에서 agent 가 어떤 aciton 을 결정할지에 대한 일

종의 판단 함수 이다.

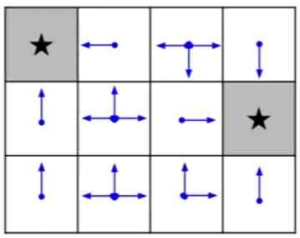
함수로 생각하면 다음과 같다. Input State -> policy function -> Output action

결국, 우리가 학습해야하는것은 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.

## A simple MDP: Grid World



Random Policy



Optimal Policy

위의 그림은 simple MDP로 2가지 Policy로 나눠볼 수 있습니다. 두 가지 방법 동일하게 어느 점에서 시작하든지 회색 별 위치에 도달하게 되면 에피소드가 종료가 됩니다. 이 경우 시간에 따라 보상은 negative로 주어지므로, 최단 거리로 끝날 수 있도록 설정하는 것이 가장 좋은 방법이 됩니다. 차이점으로 좌측 Random Policy의 경우 모든 확률이 동일하지만, Optimal Policy는 가까운 방향의 회색 지점에 도달할 수 있도록 방향을 선택하게 행동을 취합니다.

## The optimal policy $\pi^*$

We want to find optimal policy $\pi^*$ that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability…)?
Maximize the **expected sum of rewards!**

Formally: $\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi\right]$ with $s_0 \sim p(s_0), a_t \sim \pi(\cdot|s_t), s_{t+1} \sim p(\cdot|s_t, a_t)$

우리는 보상의 합을 최대화하는 optimal policy π∗를 찾고 싶어 합니다. 하지만 초기 상태, 전이 확률 등 랜덤성에 대한 제어는 어떻게 할 수 있을까요? 그것은 보상의 합의 기댓값을 최대화하는 것을 찾아주면 해결됩니다. 여기서 argmax는 뒤의 값을 최대로 만들어주는 π값입니다. 좀 더 쉽게 말하자면, y값을 최대로 만들어주는 x값을 구할 때, argmax를 사용하게 됩니다.

3. Value function and Q-value function

## Definitions: Value function and Q-value function

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:

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

모든 에피소드들은 하나의 경로를 가지게 됩니다. 그렇다면 현재의 상태는 얼마나 좋은 상태일까? 현재 상태가 좋은지 알려주는 것이 value function이다. 임의 상태의 s에 대한 가치 함수는 상태 s와 정책 π가 주어졌을 때 누적 보상의 기댓값으로 표현할 수 있습니다. 그러면 state와 action 쌍이 얼마나 좋은지는 어떻게 알 수 있을까? 우리는 이를 Q-가치 함수로 정의할 수 있습니다. Q-가치 함수는 정책 π, 행동 a, 상태 s가 주어졌을 때 받을 수 있는 누적 보상의 기댓값으로 표현됩니다.

# 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 $\pi^*$ corresponds to taking the best action in any state as specified by Q*

위의 슬라이드에서 Q∗는 Q-value function을 최대화시킨 것입니다. 여기서 Q∗는 에이전트가 최적으로 행동할 것이기에 Bellman equation을 만족할 것입니다. 이것이 의미하는 바는 어떤 s, a가 주어지던지 현재 (s, a)에서 받을 수 있는 r과 에피소드가 종료될 s'까지의 보상을 더한 값이 된다는 것입니다. 여기서 우리는 이미 최적의 정책을 알고 있기 때문에 s'에서 우리가 취할 수 있는 최상의 행동을 취할 수 있습니다. s'에서의 Q∗의 값은 우리가 현재 상태에서 취할 수 있는 모든 행동들 중에 max(Q∗(s',a'))을 만드는 값이 됩니다. 이를 통해서 최적의 Q값을 얻을 수 있습니다.

# 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!

Optimal policy를 위한 해결 방법은 Value iteration algorithm입니다. 이 알고리즘은 반복적인 업데이트를 위해서 벨만 방정식을 사용합니다. Qi는 i가 inf로 갈 때, Q∗에 수렴하게 됩니다. 이것은 잘 동작하지만 문제가 발생합니다. 이것은 확장 가능(scalable) 하지 않다는 것입니다. 이 문제를 해결하기 위해서는 Q(s, a)를 신경망을 이용해서 근사 시키면 가능합니다.

# 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-learning은 전이 확률과 보상을 초기에 알지 못한 상황에서 Q-value iteration 알고리즘을 적용한 것입니다. Q-learning은 에이전트가 플레이하는 것을 보고 점진적으로 Q-가치 추정을 향상하는 방식으로 작동합니다. 행동 가치 함수를 추정하기 위해서 함수 근사를 이용합니다. 이때, deep q-learing을 사용하는 데, 여기에 있는 θ는 neural network 내부의 가중치입니다.

# Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

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

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

Iteratively try to make the Q-value close to the target value ($y_i$) it should have, if Q-function corresponds to optimal Q* (and optimal policy $\pi^*$)

Backward Pass

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

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

Forward pass에서 Q-function은 학습시켜서 벨만 방정식의 에러가 최소가 되도록 하면 됩니다. 여기서 손실 함수는 Q(s, a)가 목적함수와 얼마나 멀리 떨어져 있는지 측정하는 것입니다. Backward pass에서는 계산한 손실을 기반으로 파라미터 θ를 업데이트합니다. 그래서 Forward pass와 Backward pass를 반복적으로 업데이트해서 우리가 가진 Q-function이 타깃과 가까워지도록 학습시킵니다.

# Case Study: Playing Atari Games

**Objective**: Complete the game with the highest score

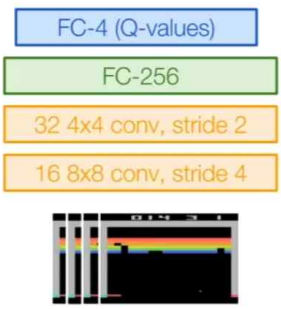
**State:** Raw pixel inputs of the game state
**Action:** Game controls e.g. Left, Right, Up, Down
**Reward:** Score increase/decrease at each time step

Atari game은 Q-learning의 대표적인 방법입니다. 이 게임은 높은 점수를 획득하는 것이 목표

# Q-network Architecture

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

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)

Q-function에서 사용한 네트워크는 위의 그림과 같이 생겼습니다. Q-network는 θ를 가중치로 가지고 있습니다. input으로는 최근 4개 프레임을 누적해서 사용합니다. 이것의 목적은 상태와 행동에 대한 쌍을 구하기 위해서 Q-value를 근사 시키는 것입니다. 이 neural network에서는 softmax를 사용하지 않습니다. 직접 확률을 예측하는 것이 목표이기 때문에 regression을 활용합니다. 이 구조는 한 번의 forward pass만으로 현재 상태에 해당하는 모든 함수에 대한 Q-value를 계산할 수 있습니다.

## Training the Q-network: Loss function (from before)

Remember: want to find a Q-function that satisfies the Bellman Equation:

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

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

Iteratively try to make the Q-value close to the target value ($y_i$) it should have, if Q-function corresponds to optimal Q* (and optimal policy $\pi^*$)

Backward Pass

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

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

네트워크로 근사 시킨 함수도 벨만 방정식을 만족해야 합니다. 따라서, 네트워크의 출력인 Q-value가 타깃 값과 가까워지도록 반복적으로 학습해야 합니다. backward pass에서는 손실 함수를 기반으로 그레디언트를 계산하고 이를 바탕으로 네트워크를 학습시킵니다.

## Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand size) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions ($s_t$, $a_t$, $r_t$, $s_{t+1}$) as game (experience) episodes are played
- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

Each transition can also contribute to multiple weight updates
=> greater data efficiency

Q-network를 학습시킬 때 몇 가지 문제점이 존재합니다. 첫 번째는 하나의 배치에서 시간적으로 연속적인 샘플들로 학습하면 좋지 않습니다. 왜냐하면 연속적인 샘플들을 학습시키면 모든 샘플들이 상관관계를 가지기 때문입니다. 이는 학습에 아주 비효율적입니다. 두 번째로 현재 Q-network 파라미터를 생각해보면 우리가 어떤 행동을 해야 할 지에 대한 정책을 결정한다는 것은 우리가 다음 샘플들도 결정하게 된다는 의미와 같습니다. 만약, 현재 상태에서 "왼쪽"으로 이동하는 것이 보상을 최대화하는 행동이라면 결국 다음 샘플들도 전부 "왼쪽"에서 발생할 수 있는 것들로만 편향될 것입니다.

이 두 가지 문제점을 해결하는 방법이 experience replay라는 방법입니다. 이 방법은

replay memory를 이용합니다. replay memory에는 (상태, 행동, 보상, 다음 상태)로 구성된 전이 테이블이 있습니다. 게임 에피소드를 플레이하면서 더 많은 경험을 얻음에 따라 전이 테이블을 지속적으로 업데이트시킵니다. 여기에서는 replay memory에서의 임의의 미니 배치를 이용해서 Q-network를 학습시킵니다. 연속적인 샘플을 사용하는 대신 전이 테이블에서 임의로 샘플링된 샘플을 사용하는 것이죠. 이 방식을 통해서 앞서 상관관계로 발생하는 문제를 해결할 수 있을 것입니다. 추가적으로 각각의 전이가 가중치 업데이트에 여러 차례(딱 한 번이 아니라) 기여할 수 있습니다. 전이 테이블로부터 샘플링을 하면 하나의 샘플도 여러 번 뽑힐 수 있기 때문이죠. 이를 통해 데이터 효율이 훨씬 더 증가하게 됩니다.

## Putting it together: Deep Q-Learning with Experience Replay

**Algorithm 1** Deep Q-learning with Experience Replay

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

Experience Replay를 적용한 Deep Q-learning은 다음과 같은 방식으로 실행됩니다.

1. memory capacity인 N을 정해주고 Q-network를 임의의 가중치로 초기화시킵니다.
2. 앞으로 M번의 에피소드를 진행합니다.
3. 각 에피소드마다 상태를 초기화시켜줍니다.
4. 상태는 게임 시작하면 픽셀이 들어가게 됩니다.
5. 게임이 진행 중인 매 다음 스탭마다 적은 확률로 임의의 행동을 취하게 합니다(정책을 따르지 않고).
6. 낮은 확률로 임의의 행동을 취하거나 현재 정책에 따라 greedy action을 취합니다.
7. 이렇게 행동 at를 취하면 보상(rt)과 다음 상태(St+1)를 얻을 수 있습니다.
8. transition을 replay memory에 저장합니다.
9. replay memory에서 임의의 미니 배치 transitions을 샘플링한 다음에 이를 이용해서 업데이트합니다.

4. Policy Gradients

## Policy Gradients

What is a problem with Q-learning?
The Q-function can be very complicated!

Example: a robot grasping an object has a very high-dimensional state => hard to learn exact value of every (state, action) pair

But the policy can be much simpler: just close your hand
Can we learn a policy directly, e.g. finding the best policy from a collection of policies?

Q-learning에는 문제가 있습니다. 그것은 Q-function이 너무나도 복잡하다는 것입니다. Q-function은 모든 state와 action 쌍들을 학습해야만 합니다. 그런데 가령, 로봇이 어떤 물체를 손에 쥐는 문제를 풀어야 한다고 생각해봅시다. 이 경우 아주 고차원의 상태 공간이 존재합니다. 가령 로봇의 모든 관절의 위치와 각도가 이룰 수 있는 모든 경우의 수를 생각해볼 수 있겠죠. 이 모든 state와 action을 학습시키는 것은 아주 어려운 문제입니다. 하지만, policy를 활용하면 더 간단할 수 있습니다. 정책 자체는 어떻게 학습시킬 수 있을까요? 만약 가능하다면 여러 정책들 가운데 최고의 정책을 찾아낼 수 있을 것입니다. 정책을 결정하기에 앞서서 Q-value를 추정하지 않고도 가능합니다. 이러한 접근 방법이 policy gradients입니다.

## 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!

일반적으로, 정책들은 가중치 θ에 의해서 매개변수화 됩니다. J(θ)는 미래에 받을 보상들의 누적 합의 기댓값으로 나타낼 수 있습니다. 이는 우리가 지금까지 사용했던 보상과 동일합니다.

최적의 정책인 θ∗를 찾는 것인데 이것은 argmaxJ(θ)로 나타낼 수 있습니다. 보상의 기댓값을 최대로 하는 정책 파라미터를 찾으면 됩니다. 그러면 이 문제는 어떻게 풀면 될까요? policy parameter에 대해서 gradient ascent를 수행하면 됩니다.

## REINFORCE algorithm

Mathematically, we can write:

$$J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)} [r(\tau)]$$
$$= \int_{\tau} r(\tau) p(\tau;\theta) \mathrm{d}\tau$$

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

REINFORCE algorithm은 수학적으로 보면, J(θ)는 경로에 대한 미래 보상의 기댓값으로 표현할 수 있습니다.

## REINFORCE algorithm

Expected reward: $J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)} [r(\tau)]$
$= \int_{\tau} r(\tau) p(\tau;\theta) \mathrm{d}\tau$

Now let's differentiate this: $\nabla_\theta J(\theta) = \int_{\tau} r(\tau) \nabla_\theta p(\tau;\theta) \mathrm{d}\tau$ 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:

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

Can estimate with Monte Carlo sampling

이제 gradient ascent를 구해야 합니다. gradient ascent를 구하려면 미분을 해야 하는 데, p가 θ에 종속되어 있는 상황에서 기댓값 안에 그레디언트가 있으면 문제가 발생할 수 있습니다(Intractable). τ에 대한 적분을 수행해야 되기 때문에 어렵습니다. 하지만, 여기서 트릭이 하나 존재합니다. p(τ ; θ)의 그레디언트를 구하는 것이 어렵기 때문에 log(p(τ ; θ))를 미분하는 것을 활용했습니다(ln(f(x))는 미분하면 f′(x)/f(x) 형태가 됩니다). 이를 활용하면, 기댓값으로 표현할 수 있게 되고 Monte Carlo 샘플링(난수를 이용하여 함수의 값을 확률적으로 계산하는 알고리즘)을 사용할 수 있습니다. 이것이 바로 REINFORCE의 주요 아이디어입니다.

## REINFORCE algorithm

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

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 $\tau$, we can estimate J($\theta$) with

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

우리는 p(τ ; θ)를 전이 확률을 모르는 채로 계산할 수 있을까요? p(τ): 어떤 경로에 대한 확률을 나타내는데 이는 현재 (상태, 행동)이 주어졌을 때, 다음에 얻게 될 모든 상태에 대해서 "전이 확률"과 "정책 π로부터 얻은 행동에 대한 확률"의 곱의 형태로 이뤄집니다. 따라서, 이를 모두 곱하면 경로에 대한 확률을 얻어낼 수 있을 것입니다. 위에서도 볼 수 있듯이 전이 확률은 θ와 무관한 항이기 때문에 그레디언트를 계산할 때 신경 쓰지 않아도 됩니다. 해당 항은 미분하면 상수로 취급해서 0이 되기 때문입니다.

## 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($\tau$) is high, push up the probabilities of the actions seen
- If r($\tau$) 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?

지금까지는 하나의 경로만 고려했었지만 기댓값을 계산하기 위해서는 여러 개의 경로를 샘플링할 수도 있을 것입니다. 그레디언트 계산을 하기 위해서 지금까지 유도해본 것들을 한번 해석해 봅시다. 어떤 경로로부터 얻은 보상이 크다면, 즉 어떤 일련의 행동들이 잘 수행한 것이라면 그 일련들 행동들을 할 확률을 높여줍니다. 그 행동들이 좋았다는 것을 말해주는 것입니다. 반면 어떤 경로에 대한 보상이 낮다면 해당 확률을 낮춥니다. 그러한 행동들은 좋지 못한 행동이었으며 따라서 그 경로가 샘플링되지 못하도록 해야 합니다.

여기 수식을 보시면 π(a|s)는 우리가 취한 행동들에 대한 우도(likelihood)입니다. 파라미터를 조정하기 위해서 그레디언트를 사용할 것이고, 그레디언트는 우도를 높이려면 어떻게 해야 하는지를 알려줍니다. 즉 우리가 받게 될 보상, 즉 행동들이 얼마나 좋았는지에 대한 그레디언트를 통해 파라미터를 조정하는 것입니다. 단순하게 말해서, 어떤 경로가 좋았으면 경로에 포함되었던 모든 행동이 좋았다는 것을 의미합니다. 하지만 기댓값에 의해서 이 모든 것들이 averages out 됩니다. average out을 통해 unbiased estimator를 얻을 수 있고 충분히 많은 샘플링을 한다면 그레디언트를 잘 이용해서 정확하고 좋은 estimator를 얻을 수 있을 것입니다. 이 방법이 좋은 이유는 그레디언트만 잘 계산한다면 손실 함수를 작게 만들 수 있고 정책 파라미터 θ에 대한 (적어도) local optimum을 구할 수 있기 때문입니다.

하지만 여기에도 문제가 존재합니다. 문제의 원인은 높은 분산에 있습니다. 왜냐하면 신뢰할당 문제(credit assignment)가 아주 어렵기 때문입니다. 일단 보상을 받았으면 해당 경로의 모든 행동들이 좋았다는 정보만 알려줄 것입니다. 하지만 우리는 구체적으로 어떤 행동이 최선이었는지를 알고 싶을 수도 있지만, 이 정보는 average out 됩니다. 하지만 구체적으로 어떤 행동이 좋았는지 알 길이 없으며, 좋은 estimator를 위해서는 샘플링을 충분히 하는 수밖에 없습니다. 이 문제는 결국 분산을 줄이고 estimator의 성능을 높이기 위해서는 어떻게 해야 하는지에 관한 질문으로 귀결됩니다.

# 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 $\gamma$ to ignore delayed effects

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

그래서 위에서 말한 분산의 문제를 감소시키는 것은 policy gradient에서 중요한 분야입니다. 이 방법은 샘플링을 적게 하면서도 estimator의 성능을 높일 수 있는 방법이기도 합니다.

첫 번째 아이디어는 해당 상태로부터 받을 미래의 보상만을 고려하여 어떤 행동을 취할 확률을 키워주는 방법입니다. 이 방법은 해당 경로에서 얻을 수 있는 전체 보상을 고려하는 것 대신에 맨 처음부터가 아닌, 현재의 time step에서부터 종료 시점까지 얻을 수 있는 보상의 합

을 고려하는 것입니다. 이 방법이 의도하는 바는 어떤 행동이 발생시키는 미래의 보상이 얼마나 큰지를 고려하겠다는 것입니다.

두 번째 아이디어는 지연된 보상에 대해서 할인율을 적용하는 것입니다. 수식을 보면, 할인율이 추가되었습니다. 할인율이 의미하는 것은 당장 받을 수 있는 보상과 조금은 더 늦게 받은 보상 간의 차이를 구별하는 것입니다. 이 방법은 어떤 행동이 좋은 행동인지 아닌지를 해당 행동에 가까운 곳에서 찾습니다. 나중에 수행하는 행동에 대해서는 가중치를 조금 낮추는 것입니다.

이 두 가지 방법은 실제로도 일반적으로 많이 사용하기도 하는 아주 쉬운 아이디어입니다.

## 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 아이디어는 baseline이라는 방법입니다. 경로에서부터 계산한 값을 그대로 사용하는 것은 문제가 있습니다. 그런 값들 자체가 반드시 의미 있는 값은 아닐 수도 있기 때문입니다. 가령 보상이 모두 양수이기만 해도 행동들에 대한 확률이 계속 커지기만 할 것입니다. 물론, 그것이 그것대로 의미가 있을 순 있지만 가장 중요한 것은 얻은 보상이 우리가 얻을 것이라고 예상했던 것보다 좋은 것인지 아닌지를 판단하는 것입니다. 이를 다루기 위해서 baseline function을 사용할 수 있습니다. 이 방법은 상태를 이용하는 방법입니다.

baseline 함수가 말하고자 하는 것은 해당 상태에서 우리가 얼마만큼의 보상을 원하는지입니다(기준). 그렇게 되면 확률을 키우거나 줄이는 보상 수식이 조금 바뀌게 됩니다. 이를 수식에 적용하면 미래에 얻을 보상들의 합을 특정 기준이 되는 값(baseline)에서 값을 빼주는 형태가 되며 이를 통해 우리가 기대했던 것에 비해 보상이 좋은지 아닌지에 대한 상대적인 값을 얻을 수 있습니다.

## 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”

그렇다면 baseline을 어떻게 선택하면 좋을까요? 가장 단순한 baseline은 지금까지 경험했던 보상들에 대해서 이동 평균을 취하는 것입니다. 에피소드를 수행하는 학습 과정에서 지금까지 봤던 모든 경로들에 대해서 보상이 어땠는지에 대한 평균을 내는 것입니다. 이 아이디어를 이용해서 현재 보상이 상대적으로 좋은지 나쁜지를 알 수 있습니다. 우리가 지금까지 배운 variance reduction방법을 통상적으로 "vanilla REINFORCE"라고 부릅니다. 할인율을 적용하여 미래에 받을 보상을 누적시키고 여기에 단순한 baseline을 추가하는 방식입니다.

## How to choose the baseline?

A better baseline: Want to push up the probability of an action from a state, if this action was better than the **expected value of what we should get from that state**.

Q: What does this remind you of?

A: Q-function and value function!

Intuitively, we are happy with an action $a_t$ in a state $s_t$ if $Q^\pi(s_t, a_t) - V^\pi(s_t)$ is large. On the contrary, we are unhappy with an action if it’s small.

Using this, we get the estimator: $\nabla_\theta J(\theta) \approx \sum_{t \geq 0} (Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t)$

이제는 baseline의 주요 아이디어와 더 좋은 baseline을 선택하는 방법에 대해서 살펴보도록 하겠습니다. 우선 더 좋은 baseline이란 무엇인지 생각해봅시다. 우선 우리는 어떤 행동이 그 상태에서의 기댓 값 보다 좋은 경우에는 해당 상태에서 그 행동을 수행할 확률이 크길 원할 것입니다. //자 그럼 그 상태로부터 기대할 수 있는 값이라면 여러분은 어떤 것이 생각나십니까? 바로 가치함수입니다. Q-learning에서 다뤘던 가치함수와 Q-function을 이용해 볼 것입니다. 여기서 직관은 우리가 어떤 상태 s에서 어떤 행동을 취했을 때, 어떤 조건을 만족하면 그 행동이 좋았다고 판단하는 것입니다. 그 조건은 어떤 상태에서 특정 행동을 했을 때 얻을 수 있는 Q-value가 그 상태에서 얻을 수 있는 미래에 받을 수 있는 누적 보상들의 기댓 값이

라는 가치 함수보다 더 큰 경우를 의미합니다. 이는 그 행동이 우리가 선택하지 않은 다른 행동들보다 더 좋았다는 것을 의미합니다. 반대로 그 차이 값이 음수거나 작은 경우에는 안 좋은 행동을 취했다는 것을 의미하는 것이겠죠. 이를 다시 estimator 수식에 적용해 보겠습니다. 수식 자체는 기존과 동일하지만 앞선 수식에서는 variation reduction 기법과 baseline만 추가된 형태였다면, 지금은 현재 행동이 얼마나 좋은 행동이었는지를 해당 상태에서의 Q-function과 value function의 차이를 통해 나타냅니다.

## Actor-Critic Algorithm

**Problem:** we don't know Q and V. Can we learn them?

**Yes,** using Q-learning! We can combine Policy Gradients and Q-learning by training both an **actor** (the policy) and a **critic** (the Q-function).

- The actor decides which action to take, and the critic tells the actor how good its action was and how it should adjust
- Also alleviates the task of the critic as it only has to learn the values of (state, action) pairs generated by the policy
- Can also incorporate Q-learning tricks e.g. experience replay
- **Remark:** we can define by the **advantage function** how much an action was better than expected

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

우리가 지금까지 살펴본 REINFORCE 알고리즘에서는 Q-function과 Value-function을 구하지 않았습니다. 그렇다면 이를 학습시킬 수 있을까요? 정답은 가능합니다. 바로 Q-learning을 통해서 가능합니다. 우리는 policy gradient와 Q-learning을 조합해서 이를 학습시킬 수 있습니다. 여기서 actor가 policy이고, critic이 Q-function입니다. 이들은 어떤 상태가 얼마나 좋은지 그리고 상태에서의 어떤 행동이 얼마나 좋았는지를 말해줍니다.

Actor-Critic 알고리즘에서 actor는 어떤 행동을 취할지를 결정합니다. 그리고 critic은 그 행동이 얼마나 좋았으며 어떤 식으로 조절해 나가야 하는지를 알려줍니다. 그리고 Q-learning은 기존의 방법보다는 조금 완화된 임무를 수행합니다. 기존의 Q-learning에서는 모든 (상태, 행동) 쌍에 대한 Q-value를 학습해야만 했습니다. 하지만, 여기서는 policy가 만들어낸 (상태, 행동) 쌍에 대해서만 학습을 시키면 됩니다. 그리고 여기에서도 Q-function을 학습시킬 때, experience replay와 같은 다양한 학습 전략을 추가할 수 있습니다. 여기서 Q(s, a) - V(s)를 보상 함수(advantage function)로 나타낼 것입니다. 즉 보상 함수는 어떤 행동을 했을 때 얼마나 많은 보상이 주어지는 지를 나타냅니다. 행동이 예상했던 것보다 얼마나 더 좋은지를 나타내는 것입니다.

# Actor-Critic Algorithm

Initialize policy parameters $\theta$, critic parameters $\phi$
**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**

Actor-Critic 알고리즘은 다음과 같은 방식으로 진행됩니다.

1. policy parameter인 θ와 critic parameter인 π를 초기화시켜줍니다.
2. 매 학습 iteration마다 현재의 정책을 기반으로 M개의 경로를 샘플링합니다.
3. 그레디언트를 계산합니다. 각 경로마다 보상 함수를 계산할 것입니다.
4. 보상함수를 이용해서 gradient estimator를 계산하고 이를 전부 누적시킵니다.
5. 우리는 critic parameter인 π를 학습시키기 위해서 가치 함수를 학습시켜야 합니다. 이는 보상 함수를 최소화시키는 것과 동일합니다. 이를 통해 가치 함수가 벨만 방정식에 근사하도록 학습시킬 수 있습니다.

이런 식으로 policy function과 critic function을 반복적으로 학습시키고 그레디언트를 업데이트하면서 학습을 진행합니다.

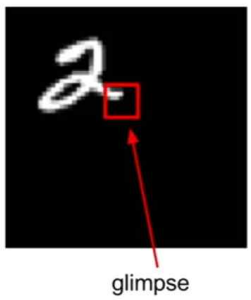
## REINFORCE in action: Recurrent Attention Model (RAM)

**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

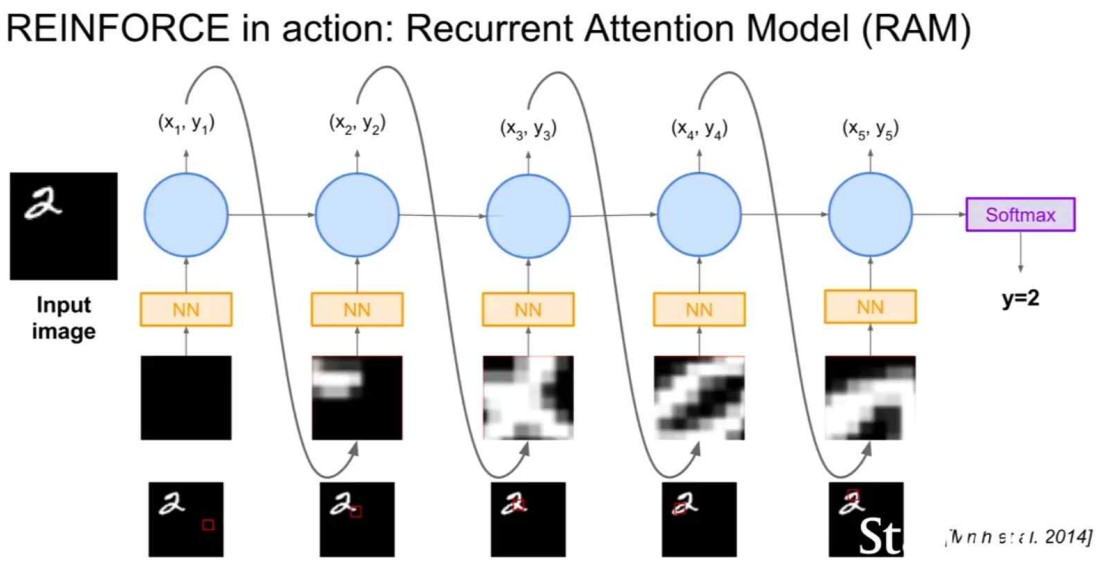



Glimpsing is a non-differentiable operation => learn policy for how to take glimpse actions using REINFORCE
Given state of glimpses seen so far, use RNN to model the state and output next action

Recurrent Attention Model(RAM)은 hard attention과 관련이 있는 방법인데 요즘 컴퓨터 비전 분야에서 매우 중요합니다. hard attention은 image classification과 관련이 있는데, 여기에서도 이미지의 클래스를 분류하는 문제이지만 이미지의 일련의 glimpses를 가지고만 예측해야 합니다. 이미지 전체가 아닌 지역적인 부분만을 보는 것이며 어떤 부분을 볼지를 선택적으로 집중할 수 있으며 그렇게 살펴보면서 정보를 얻어나가는 것입니다.

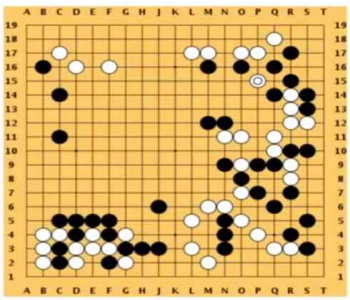
이런 방식으로 접근하는 이유는 우선 인간의 안구 운동에 따른 지각 능력으로부터의 영감을 받았기 때문입니다. 가령 우리가 복잡한 이미지를 보고 있고 이 이미지가 어떤 이미지인가를 알아내야 할 때 처음에는 저해상도로 살펴본 다음에 점점 더 세부적인 것들을 들여다보게 되겠죠. 두 번째로, 이런 식으로 이미지의 지역적인 부분만 살펴보는 방법은 계산 자원(computational resources)을 절약할 수 있습니다. 이미지 전체를 전부 처리할 필요가 없습니다. 우선 저해상도로 살펴보면서 어디서부터 시작할지 정하고 고해상도로 세부적인 것들을 찾아내는 방식이기 때문입니다. 이로 인해 계산 자원을 절약할 수 있으며, 크기가 큰 이미지들을 더 효과적으로 처리할 수 있는 scalability를 가질 수 있습니다. 그리고 마지막으로 이 방법은 실제로 classification 성능을 높여주기도 합니다. 이 방법을 사용하면 필요 없는 부분은 무시할 수 있기 때문입니다. ConvNet에 이미지 전체를 집어넣는 것이 아니라, ConvNet으로 실제로 내가 처리하고 싶은 부분만을 잘 선택할 수 있을 것입니다.

상태는 지금까지 관찰한 glimpses라고 할 수 있으며, 우리가 지금까지 얻어낸 정보라고 할 수도 있습니다. 행동은 다음에 이미지 내에 어떤 부분을 볼지를 선택하는 것입니다. 실제로는 다음 스텝에서 보고 싶은 고정된 사이즈의 glimpse의 중간 x-y좌표가 될 수 있습니다. 강화 학습에서 분류 문제를 풀 때 보상은 최종 스텝에서 1인데, 이미지가 올바르게 분류되면 1, 그렇지 않으면 0입니다. 이 문제에서 강화 학습이 필요한 이유는 이미지에서 glimpses를 뽑아내는 것은 미분이 불가능한 연산이기 때문입니다. 어떻게 glimpse를 얻어낼 것인지를 정책을 통해 학습하는 것입니다. REINFORCE를 통해서 학습시킬 수 있습니다. 누적된 glimpses가 주어지고 여기에서는 상태를 모델링하기 위해서 RNN을 이용합니다. 그리고 policy parameters를 이용해서 다음 액션을 선택하게 됩니다.




자세히 어떤 방식으로 진행되는지 살펴봅시다. 먼저 입력 이미지가 들어옵니다. 그리고 이미지에서 glimpse를 추출합니다. 여기에서는 빨간색 박스가 glimpse인데, 처음 사진에서는 비어있습니다. 그리고 추출된 glimpse는 neural network를 통과합니다. 모델은 테스크에 따라 어떤 모델이든 가능합니다. 논문의 실험에서는 MNIST를 사용했는데, MNIST는 단순한 데이터셋이라서 몇 개의 작은 FC-layers로 충분할 것입니다. 만일 조금 더 복잡한 이미지 데이터셋이 있다면 fancier ConvNets를 고려하는 것도 좋은 방법입니다. glimpse를 neural network에 통과시키고 나면 앞서 말씀드렸듯이 RNN을 이용해서 지금까지 있었던 glimpses를 전부 결합시켜 줘야 합니다. neural network의 출력은 x-y 좌표입니다. 실제로는 출력값이 행동에 대한 분포의 형태인데 이것은 가우시안 분포를 따르며 결국 출력 값은 분포의 평균이 될 것입니다. 평균과 분산을 모두 출력하는 경우도 있고 분산은 고정된 값으로 설정하기도 합니다. 그럼 이 행동 분포로부터 특정 x, y 위치를 샘플링한 다음에 이 x-y 좌표를 이용해서 다음 glimpse를 얻어냅니다. 새로운 glimpse를 보시면 숫자 2의 꼬리 부분으로 이동한 것을 볼 수 있습니다. 이제야 모델이 우리가 원하는 부분을 보기 시작한 것 같습니다. 우리가 원하던 것은 모델이 이미지에서 classification에 유용한 부분을 보기를 원했으니까요. 그리고 다시 새롭게 추출한 glimpse를 neural network에 통과시킵니다. 그리고 현재 입력과 이전 상태를 입력으로 받은 RNN을 이용해서 policy를 모델링할 것입니다. 이 RNN 모델은 다음 위치의 glimpse에 대한 분포를 출력합니다. 마지막 타임 스텝에서는 결국 우리는 분류하고 싶기 때문에 softmax를 통해서 각 클래스 확률분포를 출력하도록 합니다. 그레디언트를 구하는 법은 기존과 동일합니다. 이 모델로부터 경로를 샘플링하고 그레디언트를 계산해서 backpropagation을 진행합니다. REINFORCE 알고리즘을 통해서 모델과 정책 파라미터를 학습시킬 수 있을 것입니다. 이 방법은 fine-grained image recognition 같은 문제에도 사용할 수 있습니다. 이 방법의 이점으로는 계산 효율이 좋다는 점과 이미지 내의 불필요한 정보들을 무시할 수 있다는 점이 있습니다.

# 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]*

마지막으로 2016년에 핫했던 알파고에 대해서 알아보도록 하겠습니다. 알파고는 지도 학습과 강화 학습이 섞여있습니다. 또한 Monte Carlo Tree Search와 같은 오래된 방식과 아주 최신의 딥러닝 방법도 섞여있습니다. 알파고는 어떻게 세계 챔피언을 이길 수 있었을까요?

먼저 입력 벡터를 만들어야 합니다. 바둑판과 바둑돌의 위치와 같은 요소들을 특징화 시켜서 알파고의 입력이 될 수 있도록 합니다. 알파고는 성능 향상을 위해 몇 가지 채널을 추가했습니다. 바둑돌의 색깔에 따라 채널을 따로 만들기도 했습니다. 이는 바둑의 형세를 표현하는 역할을 합니다. 그리고 이외에도 몇 가지 채널이 더 추가되었는데, 가령 move ligality, bias 등 다양한 채널이 추가되었습니다. 이렇게 상태를 정의하고 나면 네트워크를 학습시킬 차례입니다. 네트워크는 프로 바둑기사의 기보를 지도 학습으로 학습시켜서 초기화합니다. 바둑판의 현재 상태가 주어지면 다음 행동을 어떻게 취할지를 결정합니다. 프로 바둑기사의 기보가 주어지면 프로 바둑기사가 두는 수들을 supervised mapping으로 학습시킵니다. 이것이 DeepMind가 AlphaGo를 초기화시켰던 방법이고 상당히 좋은 방법입니다. 다음 단계는 policy network를 초기화할 차례입니다. policy network는 바둑판의 상태를 입력으로 받아서 어떤 수를 둬야 하는 지를 반환해 줍니다. 지금까지 살펴본 policy gradients가 거의 다 이런 식으로 동작했었습니다. 이 또한 policy gradients를 이용해서 지속적으로 학습시킵니다. 그리고 알파고는 임의의 이전 interaction에서의 자기 자신과 대국을 두며 학습을 진행합니다. 스스로 대국을 둬서 이기면 보상을 받고 지면 -1을 받습니다. 그리고 value network도 학습을 해야 합니다. 최종적인 알파고는 policy/value network와 Monte Carlo Tree Search 알고리즘의 결합된 형태입니다. 알파고가 다음 수를 어디에 놓을지는 value function과 MCTS로 계산된 값의 조합으로 결정합니다.

5. 요약

## Summary

- **Policy gradients**: very general but suffer from high variance so requires a lot of samples. **Challenge**: sample-efficiency
- **Q-learning**: does not always work but when it works, usually more sample-efficient. **Challenge**: exploration

- Guarantees:
  - **Policy Gradients**: Converges to a local minima of $J(\theta)$, often good enough!
  - **Q-learning**: Zero guarantees since you are approximating Bellman equation with a complicated function approximator

policy gradients는 아주 범용적인 방법입니다. gradient descent/ascent를 이용해서 policy parameters를 직접 업데이트했습니다. 많은 경우에 잘 동작했으나 high-variance 문제가 있었습니다. 이로 인해 아주 많은 샘플이 필요했고 sample-efficiency에 관한 문제가 대두되었습니다.

Q-learning은 항상 잘 동작하지는 않습니다. 왜냐하면 매우 큰 고차원 공간에서 (상태, 행동) 쌍을 전부 계산해야 하기 때문입니다. 하지만, Atari의 예처럼 잘 동작하기만 한다면, policy gradients보다는 sample-efficient 합니다. Q-learning에서 가장 중요한 것은 충분한 exploration을 해야 한다는 것입니다.

Policy gradients는 항상 J(θ)의 local minima로 수렴할 수 있고 이는 아주 좋은 특성입니다. 반면, Q-learning은 어떠한 것도 보장할 수 없습니다. 함수 근사를 통해 벨만 방정식을 근사 시키기 때문입니다. 이로 인해, 다양한 문제에 응용 가능성을 고려해보면 Q-learning은 학습시키기 매우 까다롭다고 할 수 있습니다.