Lecture E1. MDP with Model 1

Sim, Min Kyu, Ph.D., mksim@seoultech.ac.kr

SEOULTECH 서울과학기술대학교 데이터사이언스학과

I. Introduction and Preview

Taxonomy

Model	Component
MC	S, P
MRP	S, P, R, γ
DP	S, R, γ, A
MDP	S, P, R, γ, A

- See how we have incrementally introduced the notions of
  - Stochasticity: P
  - Reward: R and γ
  - Action: A

Goal

- MRP
  - The aim is to find $V_t(s)$ (for infinite problem) or $V_t(s)$ (for finite problem)

- DP/MDP
  1. Now, action is introduced and actions are governed by policy.
  2. *(policy evaluation)* We need to be able to evaluate $V^{\pi}(s)$ for a fixed π. This is called *policy evaluation*. This is also called as *prediction* in reinforcement learning.
  3. *(optimal value function)* We want to be able to evaluate $V^{\pi^*}(s)$, where π^* is the *optimal policy*. The quantity, $V^{\pi^*}(s)$, is optimal policy's value function, or called shortly as *optimal value function*.
  4. *(optimal policy)* We want to find the *optimal policy* π^*. This is also called as *control* in reinforcement learning

Preview of approaches

- Some natural thoughts.
  - *(policy search)* As long as *(2. policy evaluation)* is possible, then you may try out all possible policies, the scenarios of $|A|^{|S|}$. No matter how prohibitive, in theory, you will find *(4. optimal policy)* and then *(3. optimal value function)*. This method is stupid, but may work if problems are small we call this method *policy search*.
  - If you find *(3. optimal value function)*, then you can simply direct yourself to go to the next state where better optimal value function is promised. In other words, finding *(4. optimal policy)* is easy task as soon as you find *(3. optimal value function)*.
  - Reversely, if you have *(4. optimal policy)*, then you can find *(3. optimal value function)* by using *(2. policy evaluation)*.
- Summary
  - Policy evaluation is a nice tool to have.
  - Policy search is inefficient.
  - Widely applied approaches are either 1) find *optimal policy* first or 2) find *optimal value function* first.

Two possible approaches

1. Find *optimal policy* first.
   - Policy improvement (MDP)
   - Policy-based agent (RL) - policy gradient, REINFORCE, Actor-Critic.

2. Find *optimal value function* first.
   - Value iteration (MDP)
   - Value-based agent (RL) - Deep Q Learning

II. Setting

A climbing skier's problem.

*A skier is climbing a steep mountain covered in snow. If the skier attempts to climb slowly (**normal mode**) she can advance 10 meters per minute, and if she attempts to climb fast (**speed mode**) she can advance 20 meters per minute with success probability of 0.9. The remaining probability of 0.1 implies that she falls and slides back by 10 meters. If she falls at the 0 meter point, she comes back to 0 meter point again.*

*In the **normal mode**, 1 unit of her energy is consumed. In the **speed mode**, 1.5 unit of her energy is consumed. Her mission starts at the 0 meter point and ends as soon as she reaches to 70 meter point. She decides her climing mode on every minute. Her goal is to reach 70 meter point while consuming minimal amount of energy.*

*She know that there stands a helper at the 40 meter points. Every time she starts a minute at the 40 meter points, the helper pushes her so that she will only consume 0 unit of her energy in the **normal mode** and consume 0.5 unit of her energy in the **speed mode**.*

스키어가 눈으로 덮인 가파른 산을 오르고 있다. 천천히 오르려고하면 (***normal mode***) 분당 *10* 미터를 전진 할 수 있고, 빠르게 오르려고하면 (***speed mode***) 성공 확률 *0.9*로 분당 *20* 미터를 전진 할 수 있다. 남은 확률 *0.1*은 그녀가 넘어져서 *10* 미터 뒤로 미끌어지는 경우를 의미한다. *0* 미터 지점에서 넘어지면 *0*미터 지점으로 돌아온다.

normal mode에서는 *1* 단위의 에너지가 소비된다. ***speed mode***에서는 *1.5* 단위의 에너지가 소비된다. 그녀의 임무는 *0* 미터 지점에서 시작하여 *70* 미터 지점에 도달하는 즉시 종료된다. 그녀는 매분마다 클라이밍 모드를 결정하며, 그녀의 목표는 최소한의 에너지를 소비하면서 *70* 미터 지점에 도달하는 것이다.

*40*미터 지점에는 조력자가 있다. 그녀가 *40*미터 지점에서 출발한다면, 조력자가 그녀를 밀어주기에 그녀는 ***normal mode***에서는 *0* 단위, ***speed mode***에서는 *0.5* 단위의 에너지만을 소비한다.

MDP Formulation

- The tuple of (S, P, R, γ, A) needs to be defined.
- State space
  - State S_t is defined as the meter point where she stands at the beginning of time t.
  - $\mathcal{S} = \{0, 10, 20, 30, 40, 50, 60, 70\}$
- Action space
  - Action A_t is defined as the climbing mode at time t.
  - Let a_1 implies the *normal* mode and a_2 implies *speed* mode.
  - Then, $\mathcal{A} = \{a_1, a_2\}$

- Reward
  - Reward r_t depends on the current state S_t as well as her action A_t.
  - In other words, reward r_t is a function of S_t and A_t.
  - In light of this MDP defines the *reward function* $R(\cdot)$ as a bivariate function, $R(s, a) = \mathbb{E}[r_t | S_t = s, A_t = a]$.
  - Specifically,
    - $R(s, a_1) = 1$, where $s \neq 40$ and $R(40, a_1) = 0$.
    - $R(s, a_2) = 1.5$, where $s \neq 40$ and $R(40, a_2) = 0.5$.
  - The *return function* is $G_t = \sum_{i=t}^{\infty} \gamma^{t-i} r_i$ as before. Since $\gamma = 0$, we simply have $G_t = \sum_{i=t}^{\infty} r_i$.

- Transition probability
  - As discussed, transition function can be denoted as $S_{t+1} = f(S_t, A_t, \text{some randomness})$.
  - Remind that MC's transition probability matrix $\mathbf{P}$ was formed in the way that i) S_t forms rows, ii) S_{t+1} forms columns, and iii) *some randomness* fills the element. Now with a newly introduced dimension A_t, one can't possibly represent the transition as a nice 2-dimensional array.
  - However, as long as we *fix actions for all states* (or, equivalently, fix a *policy*), we can represent the transition in a matrix and a diagram.
  - The next page presents transition diagram when 1) a policy is fixed with normal mode action for all states and 2) policy is fixed with speed mode action for all states.

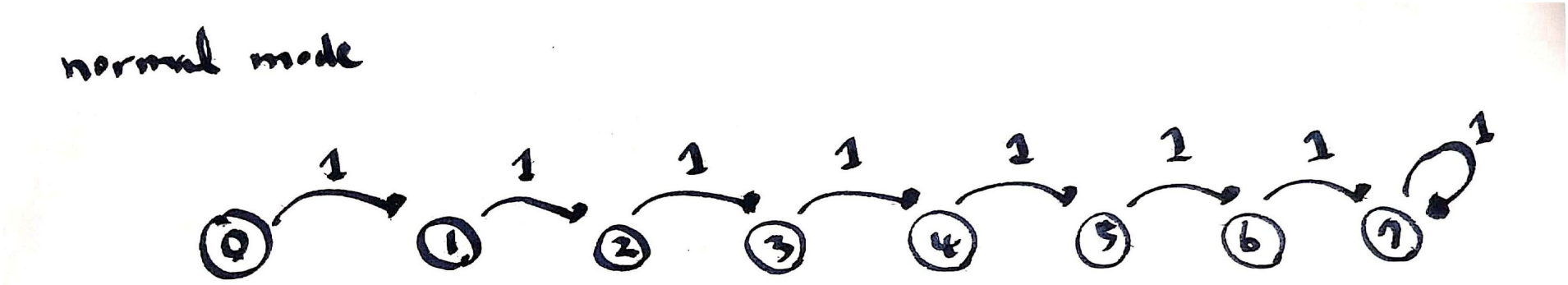


그림 1: Transition probability if normal mode is chosen for every state

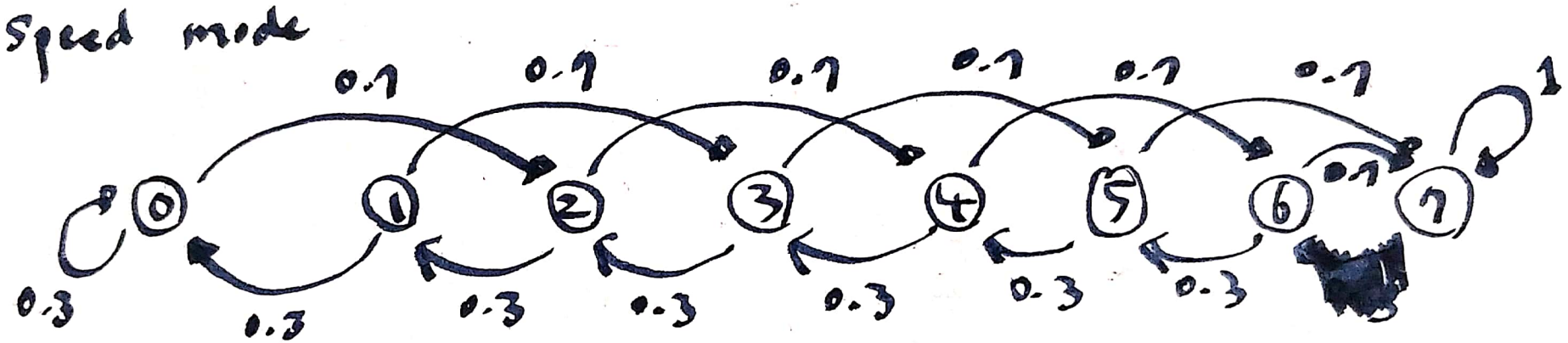


그림 2: Transition probability if normal mode is chosen for every state

- We shall denote
  - a policy of normal mode in all states as π^{normal}
  - a policy of speed mode in all states as π^{speed}

- Given a fixed policy,
  - S_t determines A_t.
  - *(If you know where you are, then you know what to do.)*
  - S_t, along with determined A_t, determines reward function $R(s, a)$.
  - *(If you know where you are, then you know what to do, and you have a clear expectation how much you will be rewarded.)*
  - Again, S_t determines not only action but also reward. It means that evaluating $V(s)$ is not any different from evaluation process of MRP.
  - *In other words, under the fixed policy, evaluating value function in MDP is reduced to that in MRP.*

- *Policy evaluation* is to evaluate $V(s)$ for all states given a policy. Based on the above discussion, this process is not any different from that of MRP.

III. Policy evaluation 1

Recap

- This section is to evaluate state-value function $V^{\pi^{speed}}(s)$ for all states.
- As a recap, remind that we covered four methods to evaluate value function in MRP.
  - Finite horizon MRP
    1. Monte-Carlo simulation
    2. Iterative solution (backward induction)
  - Infinite horizon MRP
    3. Analytic solution
    4. Iterative solution (by fixed point theorem)
- Since the current problem is in infinite horizon and we do not want to rely on the capability of inverting a matrix, we choose *4. Iterative solution*.

Development

- From MRP, we had following Bellman's equations.

$$V(s) = R(s) + \gamma \sum_{\forall s'} \mathbf{P}_{ss'} V(s') \quad \texttt{(D2, p15)}$$

$$v = R + \gamma \mathbf{P} v \quad \text{(in a vector form)}$$

- We need come up with Bellman's equation for MDP as follows.

$$V^{\pi}(s) = R^{\pi}(s) + \gamma \sum_{\forall s'} \mathbf{P}^{\pi}_{ss'} V^{\pi}(s')$$

- A few changes are made as follows.
  - $V^{\pi}(s)$ replaced $V(s)$. Why?
  - $R^{\pi}(s)$ replaced $R(s)$. Why?
  - $\mathbf{P}^{\pi}_{ss'}$ replaced $\mathbf{P}_{ss'}$. Why?
  - $V^{\pi}(s')$ replaced $V(s')$. Why?

From $V^\pi(s) = R^\pi(s) + \gamma \sum_{\forall s'} \mathbf{P}^\pi_{ss'} V^\pi(s')$, we further need two remarks.

1. Reward function $R^\pi(s)$
   - $R^\pi(s)$ is equal to $R(s, \pi(s))$ only if $\pi(s)$ is an single action. However, she may randomly choose among multiple actions on a state.
   - Including the random policy, it should be the sum of $R(s, a)$ weighted by the action distribution $\pi(a|s)$ as follows.

$$R^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) R(s, a) \tag{1}$$

   - $\pi(s)$ returns a single action on the state s if policy π is adopted.
   - $\pi(a|s)$ returns the probability of choosing an action a on the state s if policy π is adopted.
2. Transition probability $\mathbf{P}^\pi_{ss'}$
   - Likewise, if a policy allows randomized action, then this notation is not straight-forward. Again, the probability should be weighted by the action distribution $\pi(a|s)$ as follows.

$$\mathbf{P}^\pi_{ss'} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathbf{P}(s'|s, a), \tag{2}$$

   where $\mathbf{P}(s'|s, a) = \mathbb{P}(S_{t+1} = s' | S_t = s, A_t = a)$.

Iterative estimation of state-value function for a given policy.

- The pseudo code for MRP. (D2, p15)

```
1: Let epsilon <- 10^{-8}
2: Let v_0 <- zero vector
3: Let v_1 <- R + \gamma*P*v_0
4: i <- 1
5: While ||v_i-v_{i-1}|| > epsilon # any norm
6:   v_{i+1} <- R + \gamma*P*v_{i}
7:   i <- i+1
8: Return v_{i+1}
```

- The pseudo code for MDP.

```
0: For a fixed \pi,
0: For all states s,
     define R^{\pi}(s) using Eq.(1)
0: For all states s,s',
     define P^{\pi}_{ss'} using Eq.(2)
1: Let epsilon <- 10^{-8}
2: Let v_0 <- zero vector
3: Let v_1 <- R + \gamma*P*v_0
4: i <- 1
5: While ||v_i-v_{i-1}|| > epsilon # any norm
6:   v_{i+1} <- R + \gamma*P*v_{i}
7:   i <- i+1
8: Return v_{i+1}
```

- Iterative estimation of state-value function for a given policy π^{speed}

```
# 0: For a fixed \pi,
# 0: For all states s,
#      define R^{\pi}(s) using Eq.(1)
R <- c(rep(-1.5,4),-0.5,rep(-1.5,2),0)
# 0: For all states s,s',
#      define P^{\pi}_{ss'} using Eq.(2)
states <- seq(0, 70, 10)
P <- matrix(
  c(.1, 0,.9, 0, 0, 0, 0, 0,
    .1, 0, 0,.9, 0, 0, 0, 0,
     0,.1, 0, 0,.9, 0, 0, 0,
     0, 0,.1, 0, 0,.9, 0, 0,
     0, 0, 0,.1, 0, 0,.9, 0,
     0, 0, 0, 0,.1, 0, 0,.9,
     0, 0, 0, 0, 0,.1, 0,.9,
     0, 0, 0, 0, 0, 0, 0, 1),
  nrow = 8, ncol = 8, byrow = TRUE,
  dimnames = list(states, states))
```

```
R
```

```
## [1] -1.5 -1.5 -1.5 -1.5 -0.5 -1.5 -1.5  0.0
```

```
P
```

```
##      0  10  20  30  40  50  60  70
## 0  0.1 0.0 0.9 0.0 0.0 0.0 0.0 0.0
## 10 0.1 0.0 0.0 0.9 0.0 0.0 0.0 0.0
## 20 0.0 0.1 0.0 0.0 0.9 0.0 0.0 0.0
## 30 0.0 0.0 0.1 0.0 0.0 0.9 0.0 0.0
## 40 0.0 0.0 0.0 0.1 0.0 0.0 0.9 0.0
## 50 0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.9
## 60 0.0 0.0 0.0 0.0 0.0 0.1 0.0 0.9
## 70 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
```

- *(Cont'd)*

```
# 1: Let epsilon <- 10^{-8}
gamma <- 1.0
epsilon <- 10^(-8)
# 2: Let v_0 <- zero vector
v_old <- array(rep(0,8), dim=c(8,1))
# 3: Let v_1 <- R + \gamma*P*v_0
v_new <- R + gamma*P%*%v_old
# 4: i <- 1
# 5: While ||v_i-v_{i-1}|| > epsilon
# 6:   v_{i+1} <- R + \gamma*P*v_{i}
# 7:   i <- i+1
# 8: Return v_{i+1}
while (max(abs(v_new-v_old)) > epsilon) {
  v_old <- v_new
  v_new <- R + gamma*P%*%v_old
}
```

```
print(t(v_new))
```

```
##               0        10        20        30       40        50        60 70
## [1,] -5.805929 -5.208781 -4.139262 -3.475765 -2.35376 -1.735376 -1.673538  0
```

- Rewritten with intermediate saving

```
# R, P are already assigned
gamma <- 1.0
epsilon <- 10^(-8)
v_old <- array(rep(0,8), dim=c(8,1))
v_new <- R + gamma*P%*%v_old
results <- t(v_old)                  # to save
results <- rbind(results, t(v_new)) # to save
while (max(abs(v_new-v_old)) > epsilon) {
  v_old <- v_new
  v_new <- R + gamma*P%*%v_old
  results <- rbind(results, t(v_new)) # to save
}
```

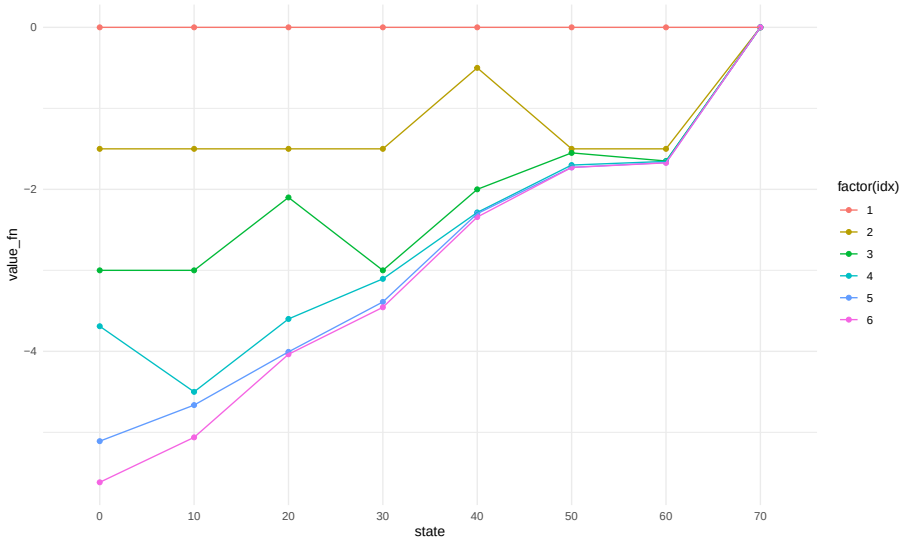
```
results <- data.frame(results)
colnames(results) <- as.character(states)
head(results)
```

```
##          0      10       20      30     40      50       60 70
## 1  0.00000  0.0000  0.00000  0.0000  0.000  0.0000  0.00000  0
## 2 -1.50000 -1.5000 -1.50000 -1.5000 -0.500 -1.5000 -1.50000  0
## 3 -3.00000 -3.0000 -2.10000 -3.0000 -2.000 -1.5500 -1.65000  0
## 4 -3.69000 -4.5000 -3.60000 -3.1050 -2.285 -1.7000 -1.65500  0
## 5 -5.10900 -4.6635 -4.00650 -3.3900 -2.300 -1.7285 -1.67000  0
## 6 -5.61675 -5.0619 -4.03635 -3.4563 -2.342 -1.7300 -1.67285  0
```

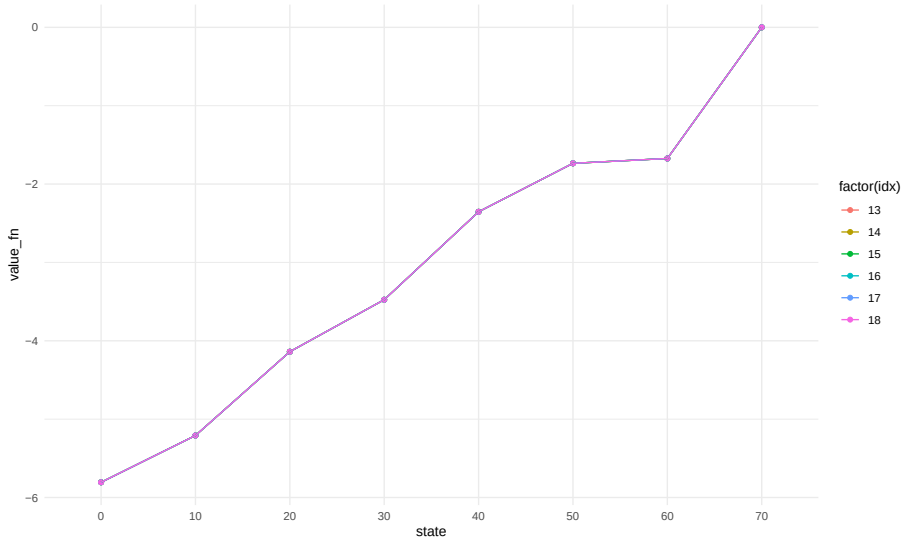
```
tail(results)
```

```
##             0        10        20        30       40        50        60 70
## 18 -5.805928 -5.208781 -4.139262 -3.475765 -2.35376 -1.735376 -1.673538  0
## 19 -5.805929 -5.208781 -4.139262 -3.475765 -2.35376 -1.735376 -1.673538  0
## 20 -5.805929 -5.208781 -4.139262 -3.475765 -2.35376 -1.735376 -1.673538  0
## 21 -5.805929 -5.208781 -4.139262 -3.475765 -2.35376 -1.735376 -1.673538  0
## 22 -5.805929 -5.208781 -4.139262 -3.475765 -2.35376 -1.735376 -1.673538  0
## 23 -5.805929 -5.208781 -4.139262 -3.475765 -2.35376 -1.735376 -1.673538  0
```

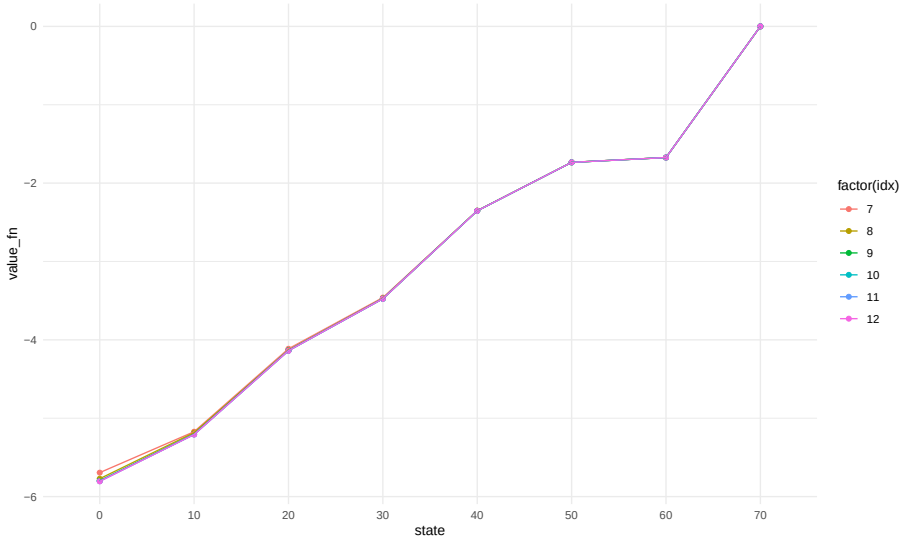
- Iteration from 1 to 6.



- Iteration from 13 to 18.



- Iteration from 7 to 12.



- The previous plot was generated by the following code.

```
library(tidyverse)
results$idx <- as.numeric(row.names(results))
results <- results %>%
  gather(as.character(states), key="state", value="value_fn")
# the first figure
results %>% filter(idx <= 6) %>%
ggplot(aes(x=state, y=value_fn, group = factor(idx), color = factor(idx))) +
  geom_point() + geom_line() +
  theme_minimal()
```

Iteration process visualized

Animation Link

- Notice it moves very quick at first, but getting slower as iteration goes.

- The previous animation was generated by the following code.
- Check my `L21` note in Data Visualization if interested.

```
library(gganimate)
library(gifski)
results$idx <- as.factor(results$idx)
fig_static <-
  ggplot(results, aes(x=state, y=value_fn, color=state)) +
  geom_point(size = 5) +
  theme_minimal()
fig_dynamic <- fig_static +
  transition_states(idx) +
  labs(title = 'Now showing Iteration Number {closest_state}') +
  enter_fade() + exit_shrink()
anim_save(
  filename = "anim_policy_eval.gif",
  animation = fig_dynamic,
  renderer = gifski_renderer())
```

IV. Policy evaluation 2

Recap

- In the previous section, we estimated state-value function for a given policy π^{speed}.
  - The pseudo code for MDP. (p.20)

```
0: For a fixed \pi,
0: For all states s,
     define R^{\pi}(s) using Eq.(1)
0: For all states s,s',
     define P^{\pi}_{ss'} using Eq.(2)
1: Let epsilon <- 10^{-8}
2: Let v_0 <- zero vector
3: Let v_1 <- R + \gamma*P*v_0
4: i <- 1
5: While ||v_i-v_{i-1}|| > epsilon # any norm
6:   v_{i+1} <- R + \gamma*P*v_{i}
7:   i <- i+1
8: Return v_{i+1}
```

  - For a given π^{speed}, we had to hard-code as below (p.21).

```
# 0: For a fixed \pi,
# 0: For all states s,
#   define R^{\pi}(s) using Eq.(1)
R <- c(rep(-1.5,4),-0.5,rep(-1.5,2),0)
# 0: For all states s,s',
#   define P^{\pi}_{ss'} using Eq.(2)
states <- seq(0, 70, 10)
P <- matrix(c(.1, 0,.9, 0, 0, 0, 0, 0,
              .1, 0, 0,.9, 0, 0, 0, 0,
               0,.1, 0, 0,.9, 0, 0, 0,
               0, 0,.1, 0, 0,.9, 0, 0,
               0, 0, 0,.1, 0, 0,.9, 0,
               0, 0, 0, 0,.1, 0, 0,.9,
               0, 0, 0, 0, 0,.1, 0,.9,
               0, 0, 0, 0, 0, 0, 0, 1),
  nrow = 8, ncol = 8, byrow = TRUE,
  dimnames = list(states, states))
```

Discussion

- With a different π, the current approach have to hard-code $R^{\pi}(s)$ and $P^{\pi}_{ss'}$ again.
- We need a more *automated policy evaluation function* such as `policy_eval()` that takes *an arbitrary policy as an input* and produces an estimate for *state-value function* as an output.
- Namely,

```
function policy_eval(\pi) {
  # 1. input
  #   \pi                  # dimension?
  # 2. preparation
  #   R <- reward_fn(\pi)  # dimension?
  #   P <- transition(\pi) # dimension?
  # 3. iterate until converge
  #   v_{i+1} <- R + \gamma*P*v_{i}
  # 4. output
  #   V^{\pi}
}
```

1. $\pi : S \to A$
2. $R^{\pi} : S \to \mathbb{R}$
3. $P^{\pi} : S \times A \to S$

Preparation for 1-3

1. $\pi : S \to A$
   - The input π should be an array whose size is $|S| \times |A|$.
   - For example, π^{speed} is as follows:

```
states <- as.character(seq(0, 70, 10))
pi_speed <- cbind(
  rep(0,length(states)), rep(1,length(states)))
rownames(pi_speed) <- states
colnames(pi_speed) <- c("normal", "speed")
pi_speed
```

```
##    normal speed
## 0       0     1
## 10      0     1
## 20      0     1
## 30      0     1
## 40      0     1
## 50      0     1
## 60      0     1
## 70      0     1
```

- It is noteworthy that the policy can be described in such a tabular form, as long as $|S|$ and $|A|$ are small enough.
- Modern reinforcement learning tackles large-scaled problems by functional approximation notably using deep neural network.

② $R^\pi : S \to \mathbb{R}$

```
# Create R(s,a) first
R_s_a <- matrix(
  c(  -1,  -1,  -1,  -1, 0.0,  -1,  -1, 0,
    -1.5,-1.5,-1.5,-1.5,-0.5,-1.5,-1.5, 0),
  nrow = length(states), ncol = 2, byrow = FALSE,
  dimnames = list(states, c("normal", "speed")))
R_s_a
```

```
##    normal speed
## 0      -1  -1.5
## 10     -1  -1.5
## 20     -1  -1.5
## 30     -1  -1.5
## 40      0  -0.5
## 50     -1  -1.5
## 60     -1  -1.5
## 70      0   0.0
```

```
reward_fn <- function(given_pi) {
  R_s_a <- matrix(
    c(  -1,  -1,  -1,  -1, 0.0,  -1,  -1, 0,
      -1.5,-1.5,-1.5,-1.5,-0.5,-1.5,-1.5, 0),
    nrow = length(states), ncol = 2, byrow = FALSE,
    dimnames = list(states, c("normal", "speed")))
  R_pi <- rowSums(given_pi*R_s_a)
  return(R_pi)
}
reward_fn(pi_speed)
```

```
##    0   10   20   30   40   50   60   70
## -1.5 -1.5 -1.5 -1.5 -0.5 -1.5 -1.5  0.0
```

3. $P^{\pi} : S \times A \to S$

```
P_normal <- matrix(
  c(0,1,0,0,0,0,0,0,
    0,0,1,0,0,0,0,0,
    0,0,0,1,0,0,0,0,
    0,0,0,0,1,0,0,0,
    0,0,0,0,0,1,0,0,
    0,0,0,0,0,0,1,0,
    0,0,0,0,0,0,0,1,
    0,0,0,0,0,0,0,1),
  nrow = 8, ncol = 8, byrow = TRUE,
  dimnames = list(states, states))
P_speed <- matrix(
  c(.1, 0,.9, 0, 0, 0, 0, 0,
    .1, 0, 0,.9, 0, 0, 0, 0,
     0,.1, 0, 0,.9, 0, 0, 0,
     0, 0,.1, 0, 0,.9, 0, 0,
     0, 0, 0,.1, 0, 0,.9, 0,
     0, 0, 0, 0,.1, 0, 0,.9,
     0, 0, 0, 0, 0,.1, 0,.9,
     0, 0, 0, 0, 0, 0, 0, 1),
  nrow = 8, ncol = 8, byrow = TRUE,
  dimnames = list(states, states))
```

```
transition <- function(given_pi,
  states, P_normal, P_speed) {

  P_out <- array(0,
    dim = c(length(states), length(states)),
    dimnames = list(states, states))

  for (s in states) {
    action_dist <- given_pi[s,]
    P <- action_dist["normal"]*P_normal +
      action_dist["speed"]*P_speed
    P_out[s,] <- P[s,]
  }

  return(P_out)
}
```

- *(test 1)*

```
pi_speed
```

```
##    normal speed
## 0       0     1
## 10      0     1
## 20      0     1
## 30      0     1
## 40      0     1
## 50      0     1
## 60      0     1
## 70      0     1
```

```
transition(pi_speed,
           states = states,
           P_normal = P_normal,
           P_speed = P_speed)
```

```
##      0  10  20  30  40  50  60  70
## 0  0.1 0.0 0.9 0.0 0.0 0.0 0.0 0.0
## 10 0.1 0.0 0.0 0.9 0.0 0.0 0.0 0.0
## 20 0.0 0.1 0.0 0.0 0.9 0.0 0.0 0.0
## 30 0.0 0.0 0.1 0.0 0.0 0.9 0.0 0.0
## 40 0.0 0.0 0.0 0.1 0.0 0.0 0.9 0.0
## 50 0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.9
## 60 0.0 0.0 0.0 0.0 0.0 0.1 0.0 0.9
## 70 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
```

- *(test 2)*

```
pi_50 <- cbind(
  rep(0.5,length(states)), rep(0.5,length(states)))
rownames(pi_50) <- states
colnames(pi_50) <- c("normal", "speed")
pi_50
```

```
##    normal speed
## 0     0.5   0.5
## 10    0.5   0.5
## 20    0.5   0.5
## 30    0.5   0.5
## 40    0.5   0.5
## 50    0.5   0.5
## 60    0.5   0.5
## 70    0.5   0.5
```

```
transition(pi_50,
           states = states,
           P_normal = P_normal,
           P_speed = P_speed)
```

```
##       0   10   20   30   40   50   60   70
## 0  0.05 0.50 0.45 0.00 0.00 0.00 0.00 0.00
## 10 0.05 0.00 0.50 0.45 0.00 0.00 0.00 0.00
## 20 0.00 0.05 0.00 0.50 0.45 0.00 0.00 0.00
## 30 0.00 0.00 0.05 0.00 0.50 0.45 0.00 0.00
## 40 0.00 0.00 0.00 0.05 0.00 0.50 0.45 0.00
## 50 0.00 0.00 0.00 0.00 0.05 0.00 0.50 0.45
## 60 0.00 0.00 0.00 0.00 0.00 0.05 0.00 0.95
## 70 0.00 0.00 0.00 0.00 0.00 0.00 0.00 1.00
```

Summary

1. $\pi : S \to A$

```
pi_speed
```

```
##    normal speed
## 0       0     1
## 10      0     1
## 20      0     1
## 30      0     1
## 40      0     1
## 50      0     1
## 60      0     1
## 70      0     1
```

2. $R^\pi : S \to \mathbb{R}$

```
reward_fn(pi_speed)
```

```
##    0   10   20   30   40   50   60   70
## -1.5 -1.5 -1.5 -1.5 -0.5 -1.5 -1.5  0.0
```

3. $P^\pi : S \times A \to S$

```
transition(pi_speed,
           states = states,
           P_normal = P_normal,
           P_speed = P_speed)
```

```
##      0  10  20  30  40  50  60  70
## 0  0.1 0.0 0.9 0.0 0.0 0.0 0.0 0.0
## 10 0.1 0.0 0.0 0.9 0.0 0.0 0.0 0.0
## 20 0.0 0.1 0.0 0.0 0.9 0.0 0.0 0.0
## 30 0.0 0.0 0.1 0.0 0.0 0.9 0.0 0.0
## 40 0.0 0.0 0.0 0.1 0.0 0.0 0.9 0.0
## 50 0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.9
## 60 0.0 0.0 0.0 0.0 0.0 0.1 0.0 0.9
## 70 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
```

Summary

1. $\pi : S \to A$

```
pi_50
```

```
##    normal speed
## 0     0.5   0.5
## 10    0.5   0.5
## 20    0.5   0.5
## 30    0.5   0.5
## 40    0.5   0.5
## 50    0.5   0.5
## 60    0.5   0.5
## 70    0.5   0.5
```

2. $R^\pi : S \to \mathbb{R}$

```
reward_fn(pi_50)
```

```
##    0   10   20   30   40   50   60   70
## -1.25 -1.25 -1.25 -1.25 -0.25 -1.25 -1.25  0.00
```

3. $P^\pi : S \times A \to S$

```
transition(pi_50,
           states = states,
           P_normal = P_normal,
           P_speed = P_speed)
```

```
##       0   10   20   30   40   50   60   70
## 0  0.05 0.50 0.45 0.00 0.00 0.00 0.00 0.00
## 10 0.05 0.00 0.50 0.45 0.00 0.00 0.00 0.00
## 20 0.00 0.05 0.00 0.50 0.45 0.00 0.00 0.00
## 30 0.00 0.00 0.05 0.00 0.50 0.45 0.00 0.00
## 40 0.00 0.00 0.00 0.05 0.00 0.50 0.45 0.00
## 50 0.00 0.00 0.00 0.00 0.05 0.00 0.50 0.45
## 60 0.00 0.00 0.00 0.00 0.00 0.05 0.00 0.95
## 70 0.00 0.00 0.00 0.00 0.00 0.00 0.00 1.00
```

Recap

- Previous pseudo code

```
function policy_eval(pi) {
  # 1. input
  #   \pi
  # 2. preparation
  #   R <- reward_fn(pi)
  #   P <- transition(\pi)
  # 3. iterate until converge
  #   v_{i+1} <- R + \gamma*P*v_{i}
  # 4. output
  #   V^{\pi}
}
```

- Implementation in previous section (p.22)

```
# 1: Let epsilon <- 10^{-8}
gamma <- 1.0
epsilon <- 10^(-8)
# 2: Let v_0 <- zero vector
v_old <- array(rep(0,8), dim=c(8,1))
# 3: Let v_1 <- R + \gamma*P*v_0
v_new <- R + gamma*P%*%v_old
# 4: i <- 1
# 5: While ||v_i-v_{i-1}|| > epsilon
# 6:   v_{i+1} <- R + \gamma*P*v_{i}
# 7:   i <- i+1
# 8: Return v_{i+1}
while (max(abs(v_new-v_old)) > epsilon) {
  v_old <- v_new
  v_new <- R + gamma*P%*%v_old
}
print(t(v_new))
```

Implementation, finally.

```
policy_eval <- function(given_pi) {
  R <- reward_fn(given_pi)
  P <- transition(given_pi,
                  states = states, P_normal = P_normal, P_speed = P_speed)
  gamma <- 1.0
  epsilon <- 10^(-8)
  v_old <- array(rep(0,8), dim=c(8,1))
  v_new <- R + gamma*P%*%v_old
  while (max(abs(v_new-v_old)) > epsilon) {
    v_old <- v_new
    v_new <- R + gamma*P%*%v_old
  }
  return(t(v_new))
}
```

```
policy_eval(pi_speed)
```

```
##              0        10        20        30       40        50        60 70
## [1,] -5.805929 -5.208781 -4.139262 -3.475765 -2.35376 -1.735376 -1.673538  0
```

```
policy_eval(pi_50)
```

```
##              0        10        20        30       40        50        60 70
## [1,] -5.969238 -5.133592 -4.119955 -3.389228 -2.04147 -2.027768 -1.351388  0
```

```
"Success isn't permarnent, and failure isn't fatal. - Mike Ditka"
```