Lecture I2. Policy-based agent 2

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

서울과학기술대학교 데이터사이언스학과

I. Simulator and Benchmark

Optional: Using python in R Studio

The R library `reticulate`

- It enables to use python in R Studio.

```
library(reticulate)
```

Python version check

- Check which version of python in your computer is active in R Studio session.
- It is desirable that the python engine in R Studio session is miniconda. Verify the path includes `R-MINI~1`.

```
Sys.which("python")
```

```
##                                                                      python
## "C:\\Users\\MINKYU~1\\AppData\\Local\\R-MINI~1\\envs\\R-RETI~1\\python.exe"
```

Use python in console.

- In Console of R Studio, type in `repl_python()` to use the console with python prompt. Typing in `exit` returns the python console back to R.

Installing python package

- Python package can be installed by `py_install()` in R session, if available from anaconda channel.

```
py_install('pandas')
```

- If not available from anaconda channel, then `virtualenv_install()` or `conda_install()` can be used. Since we are under miniconda, let's use `conda_install()` as below.

```
conda_install(envname = 'r-reticulate', packages = 'torch', pip = TRUE)
```

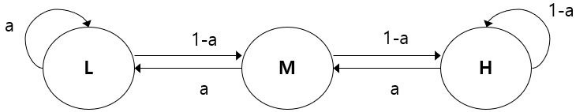
Motivation

- This section is to write the `simulator.py`, which contains information for environment, such as state, transition, and reward.
- This section also demonstrates how to evaluate benchmark policies using the environment.

`simulator.py` - design

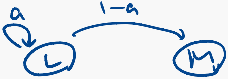
Class `simulator` must possess

1. Method to return reward for given state and action pair (i.e. $R(s,a)$)
2. Method to make transition from `state` to `next_state` depending on `action`. (i.e. $P_{ss'}$)



simulator.py - code

```
import random; import numpy as np
class Simulator():
  def __init__(self, init_state = 'M'):
    self.number_of_fish = {'L': 100, 'M': 1000, 'H': 5000}
  def reward_fn(self, state, action): # 4 `R(s,a)`
    return action * self.number_of_fish[state]
  def transition(self, state, action): # 5 `P(s,s')`
    u = random.random() # follows Unif(0,1)
    if(state == 'L'):
      if action > u: next_state = 'L'
      else: next_state = 'M'
    elif(state == 'M'):
      if action > u: next_state = 'L'
      else: next_state = 'H'
    else: # state == "H"
      if action > u: next_state = 'M'
      else: next_state = 'H'
    return next_state
```



`simulator.py` - test

Initialization

```
env = Simulator()
env.number_of_fish
```

```
## {'L': 100, 'M': 1000, 'H': 5000}
```

The reward if `action=0.7` is chosen at `state="M"`.

```
env.reward_fn(state="M", action = 0.7)
```

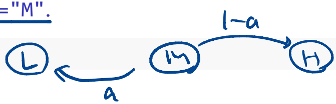
```
## 700.0
```

The next_state if `action=1.0` is chosen at `state="M"`.

```
env.transition(state = "M", action = 1.0)
```

```
## 'L'
```

next_state



A benchmark π^{50}

Evaluation strategy

- Consider a benchmark agent, whose action is always 0.5 regardless of the current state.
- We denote her policy π^{50}. In other words, $\pi^{50}(s) = 0.5, \ \forall s \in \{\text{L, M, H}\}$
- In the total 1000 episodes, let her run the business for 100 years for each episode. We shall collect the returns from each episode. (`num_ep=1000` and `num_yr=100`)

Setting

$r_0 + (0.95)^1 r_1 + \cdots + (0.95)^{99} r_{99}$

```
gamma = 0.95
pi_50 = {'L': 0.5, 'M': 0.5, 'H': 0.5}
num_ep = 1000
return_ep = np.zeros(num_ep)
num_year = 100
env = Simulator()
```

Simulating 1000 episodes of 100 years

```
for epi_i in range(num_ep):            # 1000
  state = "M"
  annual_reward = np.zeros(num_year)
  for year in range(num_year):         # 100
    action = pi_50[state]
    annual_reward[year] = env.reward_fn(state, action)
    state = env.transition(state, action)
  return_ep[epi_i] = sum(annual_reward * np.power(gamma, range(num_year)))
```

$\sum \gamma^i r_i$

Return in each episode

```
return_ep[-6:] # tail() equivalent
```

```
## array([22654.56900945,  8400.89638471, 27399.75476289, 15210.00559593,
##        24951.81960092, 18728.97083883])
```

```
np.mean(return_ep)
```

```
## 19627.35768662127
```

Analytic verification

$$\begin{array}{c|ccc} & L & M & H \\ \hline L & & & \\ M & & \begin{bmatrix} a & 1-a & \\ a & & 1-a \\ & a & 1-a \end{bmatrix} & \\ H & & & \end{array}$$

π^{50} has

$$\begin{bmatrix} .5 & .5 & \\ .5 & & .5 \\ & .5 & .5 \end{bmatrix}$$

Exercise 1

In the above policy evaluation, analytically prove that the expected return of π^{50} *is equal to (Hint: soda problem, stationary distribution, doubly stochastic, and* $0.95^{100} \approx 0.006$)

Suggested Answer

$1017 + 1017 \times 0.95 + 1017 \times 0.95^2 + \cdots + 1017 \times 0.95^{99}$

- Since the MC with policy π^{50} is doubly stochastic, the stationary distribution is $(1/3, 1/3, 1/3)$.
- In each state, the reward is 50, 500, and 2500, respectively. The average reward is therefore $(50 + 500 + 2500)/3 \approx 1017$.
- cf) $a + ar + ar^2 + \cdots + ar^{n-1} = a \cdot \frac{1-r^n}{1-r}$
- The answer is the following:

$$\begin{aligned} (Ans) &= 1017 + 1017 \cdot 0.95 + 1017 \cdot 0.95^2 + \cdots + 1017 \cdot 0.95^{99} \checkmark \\ &= 1017 \cdot \frac{1 - 0.95^{100}}{1 - 0.95} = 1017 \cdot (1 - 0.006) \cdot 20 \approx 20218 \checkmark \end{aligned}$$

Exercise 2

Consider another benchmark $\pi^{momentum}$*, whose motive is* "물들어 올때 노젓는다"*.* *Namely,* $\pi^{momentum}(L) = 0.3$, $\pi^{momentum}(M) = 0.5$, *and* $\pi^{momentum}(H) = 0.7$. *In other words, the object should be defined as:*

```
pi_momentum = {'L': 0.3, 'M': 0.5, 'H': 0.7}
```

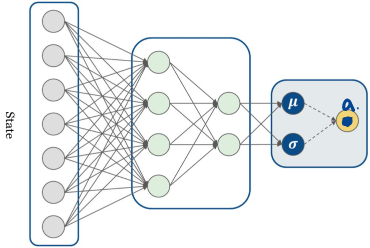
1. *Repeat the numerical experiment in this section.*
2. *Calculate the expected return using Markov chain approach.*

II. Policy Network

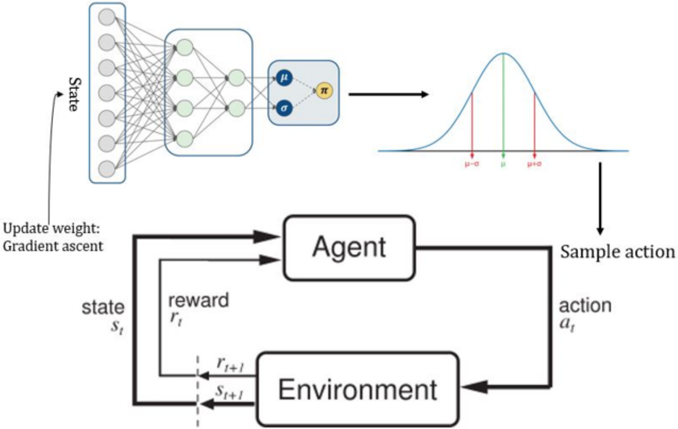
- ✓ $\pi_\theta(s, a) \rightarrow [0, 1]$ (s에서 a를 행할 확률)
- ✓ $\pi_\theta(s) \rightarrow a$
- ✓ $\boxed{\pi_\theta(s) \rightarrow N(\mu_a, \sigma_a^2)}$

Gaussian policy

- In a policy-based approach, a policy is parameterized with a parameter θ.
- Gaussian policy implies a policy to be modelled with normal distribution of parameter mean and stadard deviation (μ and σ).
- It works not only for *continuous action* but also for *randomized action*. If random action is not desired, then we expect $\sigma \searrow 0$ as the agent becomes more intelligent.
- The following figure illustrates a Gaussian policy network.

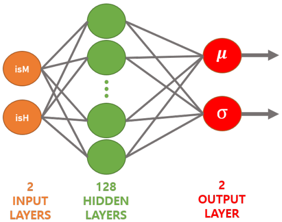


Schematic view of RL where agent is under Gaussian policy

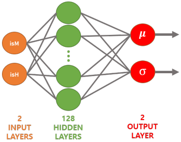


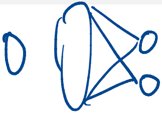
Network design

- Input: state vector of `[isM, isH]`
  - If state is at 'L' , then state vector is [0,0]
  - If state is at 'M' , then state vector is [1,0]
  - If state is at 'H' , then state vector is [0,1]
- Output: action, parameterized by μ and σ.



policy.py





```
import torch; import torch.nn as nn; import torch.nn.functional as F
class Policy(nn.Module):
  def __init__(self, num_inputs = 2, hidden_size = 128, num_outputs = 1):
    super(Policy, self).__init__()
    self.data = []
    self.fc1 = nn.Linear(num_inputs, hidden_size)
    self.fc_mu = nn.Linear(hidden_size, num_outputs) # for mu
    self.fc_sd = nn.Linear(hidden_size, num_outputs) # for sigma
  def forward(self, inputs):
    x = F.relu(self.fc1(inputs))
    mu = self.fc_mu(x)
    sd = self.fc_sd(x)
    sd = torch.clamp(sd, min=0.01, max=0.3)
    return mu, sd
```

$0.01 \le sd \le 0.3$

III. REINFORCE - vanilla algorithm for policy gradient.

Recap: Policy gradient (I1, p21)

- Policy gradient is to maximize $J(\theta)$ by finding optimal policy parameterized by θ.

$$\begin{aligned} J(\theta) &:= \sum_{s\in\mathcal{S}} p(s) V_{\pi_\theta}(s) \\ &= \sum_{s\in\mathcal{S}} p(s) \left(\sum_{a\in\mathcal{A}} \pi_\theta(s,a) Q_{\pi_\theta}(s,a) \right) \end{aligned}$$

- Policy improvement occurs by *gradient ascent* algorithm, where α is the learning rate.

$$\theta \leftarrow \theta + \alpha \cdot \nabla_\theta J(\theta)$$

- Of which, the first order derivative can be evaluated by *policy gradient theorem*.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(s,a) \cdot Q_{\pi_\theta}(s,a)]$$

- The remaining issue was to estimate $Q_{\pi_\theta}(s,a)$, resulting in various algorithms.

Recap: Policy gradient (I1, p21)

Strategy

1. Let our agent run on π_θ.
2. Collect $\nabla_\theta log\pi_\theta(s,a)$, which is the gradient of the neural net. (pytorch)
3. Estimate $Q_{\pi_\theta}(s,a)$ ✓
4. Use Step 2 and Step 3 to evaluate $\nabla_\theta J(\theta)$ ✓
5. Then, improve θ by gradient ascent. ✓
6. Go back to Step 1 until it converges.

REINFORCE (Monte-Carlo Policy Gradient)

- From the gradient of obj fn $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s,a) \cdot Q_{\pi_\theta}(s,a)]$,
- REINFORCE replaces $Q_{\pi_\theta}(s,a)$ with the return G_t. Namely,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s,a) \cdot G_t]$$

- In other words, $Q_{\pi_\theta}(s_t,a_t) \approx \mathbb{E}[G_t|s_t,a_t] = \mathbb{E}[r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots |s_t,a_t]$.
- It is also called Monte-Carlo policy gradient because it replaces $Q_{\pi_\theta}(s,a)$ with return G_t, which is an unbiased estimator for $Q_{\pi_\theta}(s,a)$.
- How do we find the expected value of $\nabla_\theta log\pi_\theta(s,a) \cdot G_t$? Just take sample mean of observed data as below:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s,a) \cdot G_t] \approx \frac{1}{N}\sum_{n=1}^{N} \nabla_\theta log\pi_\theta(s_t,a_t) \cdot G_t$$

- In the plain REINFORCE algorithm, we use $N = 1$ for approximating expectation. That is, the parameter θ is updated in every episode.

Advantage of REINFORCE

- Effective in high-dimensional or continuous action space. ✓

Disadvantage of REINFORCE

- Full stochastic path for an episode is required.
  - Can be time consuming.
  - Tricky to deal with infinite horizon problem.
- Since $N = 1$,
  - Variance is high even though it use unbiased sample for approximation.
  - Typically converge to a local optimum rather than global optimum

Pseudo-code of REINFORCE

```
REINFORCE
1: Initialize \theta
2: For each of episode under current \pi, {s_1, a_1, r_1,...,s_{T-1}, a_{T-1}, r_{T-1}}
3:   For t=1 to T-1
4:     Improve \theta
5: Return \theta
```

Class `Agent` must possess

1. A Record bin for $\boxed{log\pi_\theta(s,a)}$ and $\boxed{r}$ (`self.data`)
2. Gamma. (`self.gamma`)
3. Her policy network. (`self.model = Policy()`)
4. Optimizer for her policy network. (`optim.Adam(self.model.parameters(),...)`)
5. Method to select an action. (`select_action()`)
6. Method to train her policy network by REINFORCE algorithm. (`REINFORCE_train()`)

Agent.py - components (1/3)

1. A Record bin for $log\pi_\theta(s, a)$ and r. (self.data)
2. Gamma. (self.gamma)
3. Her policy network. (self.model = Policy())
4. Optimizer for her policy network. (optim.Adam(self.model.parameters(),...))

```
class Agent:
  def __init__(self, hidden_size, lr, gamma, num_inputs=2, num_outputs=1):
    self.data = []; # 1
    self.gamma = gamma # 2
    self.model = Policy(num_inputs, hidden_size, num_outputs) # 3
    self.optimizer = torch.optim.Adam(self.model.parameters(), lr = lr) # 4
```

Agent.py - components (2/3)

5. Method to select an action. (select_action())
   - **a.** Get value of mu (μ) and sigma (σ) from model.
   - **b.** Determine random action using a normal distribution with mean = μ and sd = σ.

```
class Agent:
  # (continued from the above page)
  def select_action(self, state):
    state_input = {'L':[0,0], 'M':[1,0], 'H':[0,1]}
    state = torch.FloatTensor(state_input[state])
    mean, sd = self.model(state) # a.
    normal = torch.distributions.Normal(mean, sd)
    action = torch.clamp(min=0, max=1, input=normal.sample()) # b.
    log_prob = normal.log_prob(action)
    return action, log_prob
```

`Agent.py` - components (3/3)

G_{100}

$G_{99} = r_{99} + \gamma G_{100}$

$\vdots$

$G_0 = r_0 + \gamma G_1$

6. Method to train her policy network. (`REINFORCE_train()`)
   - **a)** Conduct $G_{t-1} = r_{t-1} + \gamma G_t$ recursively backward to arrive G_0.
   - **b)** `loss` is an estimator for $-J(\theta)$, the negative value for objective function. Since PyTorch's default way to optimize is to minimize through gradient descent, we assign `loss` as the negative objective, which is later to be minimized. (So that $J(\theta)$ is maximized.)
   - **c)** `loss.backward()` takes the derivative of `loss`, which results in $-\nabla_{\theta} J(\theta)$.
   - **d)** `self.optimizer.step()` improves the θ by $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$.

```python
class Agent:
  # (continued from the above page)
  def REINFORCE_train(self): # 5
    self.model.train()
    self.optimizer.zero_grad() # reset optimizer
    G = 0
    for r, log_prob in self.data[::-1]: # reads from backward
      G = r + self.gamma*G # a) accumulates reward r to calculate return G
      loss = -(log_prob) * G # b) estimator for -J(\theta)
      loss.backward() # c) estimator for \nabla (-J(\theta))
    self.optimizer.step() # d) improves \theta
    self.data = []
```

Agent.py - the whole code

```
class Agent:
  def __init__(self, hidden_size, lr, gamma, num_inputs=2, num_outputs=1):
    self.data = []; self.gamma = gamma # 1 & # 2
    self.model = Policy(num_inputs, hidden_size, num_outputs) # 3
    self.optimizer = torch.optim.Adam(self.model.parameters(), lr = lr) # 4
  def select_action(self, state):
    state_input = {'L':[0,0], 'M':[1,0], 'H':[0,1]}
    state = torch.FloatTensor(state_input[state])
    mean, sd = self.model(state)
    normal = torch.distributions.Normal(mean, sd)
    normal_sample = normal.sample()
    action = torch.clamp(min=0, max=1, input=normal_sample)
    log_prob = normal.log_prob(action)
    return action, log_prob
  def REINFORCE_train(self): # 5
    self.optimizer.zero_grad() # reset optimizer
    G = 0
    for r, log_prob in self.data[::-1]: # reads from backward
      G = r + self.gamma*G # a) accumulates reward r to calculate return G
      loss = -(log_prob) * G # b) estimator for -J(\theta)
      loss.backward() # c) estimator for \nabla(-J(\theta))
    self.optimizer.step() # d) improves \theta
    self.data = []
```

Main code

```
env = Simulator()
agent = Agent(hidden_size = 16, lr=0.001, gamma = 0.95)
num_ep = 3000
return_ep = np.zeros(num_ep)
num_year = 100
for epi_i in range(num_ep):
  state = "M"
  annual_reward = np.zeros(num_year)
  for year in range(num_year):
    action, log_prob = agent.select_action(state)
    r = env.reward_fn(state, action.item())
    agent.data.append((r,log_prob))
    state = env.transition(state, action.item())
    annual_reward[year] = r
  agent.REINFORCE_train()
  return_ep[epi_i] = sum(annual_reward * np.power(gamma, range(num_year)))
```

Performance analysis

```
return_ep[-6:] # tail() equivalent
```

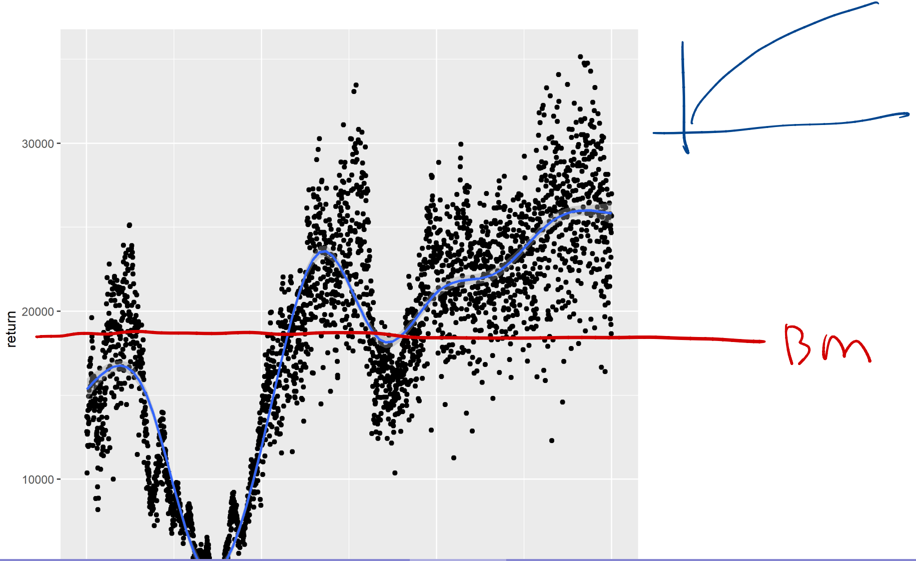
```
## array([20986.85135094, 22025.77727391, 20210.45722412, 20898.41847885,
##        22909.50528845, 17890.3328771 ])
```

```
np.mean(return_ep[-100:])
```

```
## 19096.019414224182
```

- The last 100 episode results has average return of 1.9096×10^4, which is better quantity than the benchmark π^{50}.

- RL agent must demonstrate improvement as episode matures, is the REINFORCE agent improving as episode goes?



- A plot similar to the previous page may be generated by the following code.
- (But `matplotlib` does not work well with this note format. So the plot was not generated by the following code.)

```
import matplotlib.pyplot as plt
plt.plot(x=range(len(return_ep)), y = return_ep)
plt.show()
```

- The plot in the previous page was instead generated by the following R code.
- (`py$return_ep` in R session accesses the python object `return_ep` in `reticulate` R package).

```
library(tidyverse)
return_ep_df <- data.frame(epi = 1:length(py$return_ep), return = py$return_ep)
ggplot(return_ep_df, aes(x = epi, y = return)) +
  geom_point() + geom_smooth()
```

Exercise 3

Try changing the following hyperparameters of the previous run to gain better performance.

- *hidden_size = 16*
- *lr = 0.001*
- *num_ep = 3000*

Brain analysis

- Another interest in RL other than the performance is how the intelligent agent behaves in response to the given state.

```
state_input = {'L':[0,0], 'M':[1,0], 'H':[0,1]}
# For state "L"
state = torch.FloatTensor(state_input['L'])
mu, sd = agent.model(state)
print("State L:", mu, sd)
```

```
## State L: tensor([0.3073], grad_fn=<AddBackward0>) tensor([0.1101], grad_fn=<ClampBackward1>)
```

```
# For state "M"
state = torch.FloatTensor(state_input['M'])
mu, sd = agent.model(state)
print("State M:", mu, sd)
```

```
## State M: tensor([0.0208], grad_fn=<AddBackward0>) tensor([0.0100], grad_fn=<ClampBackward1>)
```

```
# For state "H"
state = torch.FloatTensor(state_input['H'])
mu, sd = agent.model(state)
print("State H:", mu, sd)
```

```
## State H: tensor([0.2868], grad_fn=<AddBackward0>) tensor([0.0100], grad_fn=<ClampBackward1>)
```

```
"Man is gifted with reason; he is life being aware of itself; he has awareness of himself, of his
fellow man, of his past, and of the possibilities of his future. - Erich Fromm"
```