Let’s do Inverse RL!

이동민

Reinforcement Learning KR

Feb 23, 2019

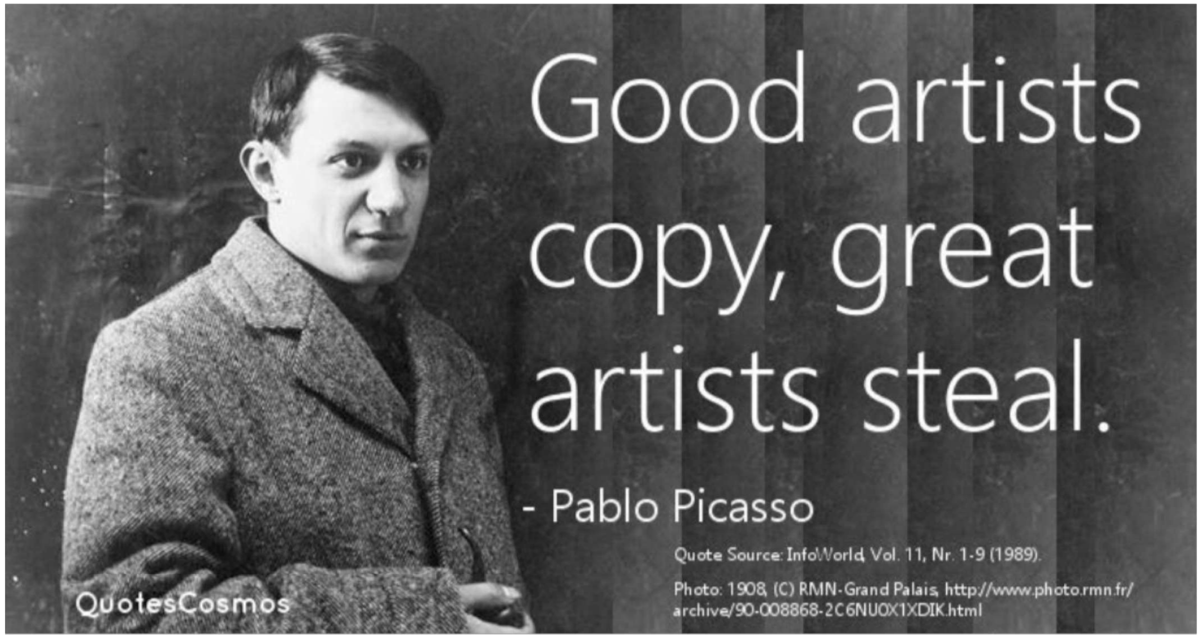
Outline

Let’s do Inverse RL!

1. 프로젝트 및 멤버 소개
2. 6개의 논문 소개
3. 각 논문의 이론 및 구현 소개

프로젝트 및 멤버 소개

프로젝트 및 멤버 소개


Good artists copy, great artists steal.
- Pablo Picasso
Quote Source: InfoWorld, Vol. 11, Nr. 1-9 (1989).
Photo: 1908, (C) RMN-Grand Palais, http://www.photo.rmn.fr/archive/90-008868-2C6NU0X1XDIK.html
QuotesCosmos

프로젝트 이름 - GAIL하자!

프로젝트 소개 - 강화학습의 세부 주제 중 하나인 Inverse RL의 논문들의
이론적 바탕을 이해하고 이를 환경에 구현해보는 프로젝트

프로젝트 및 멤버 소개

이동민

프로젝트 매니저

윤승제

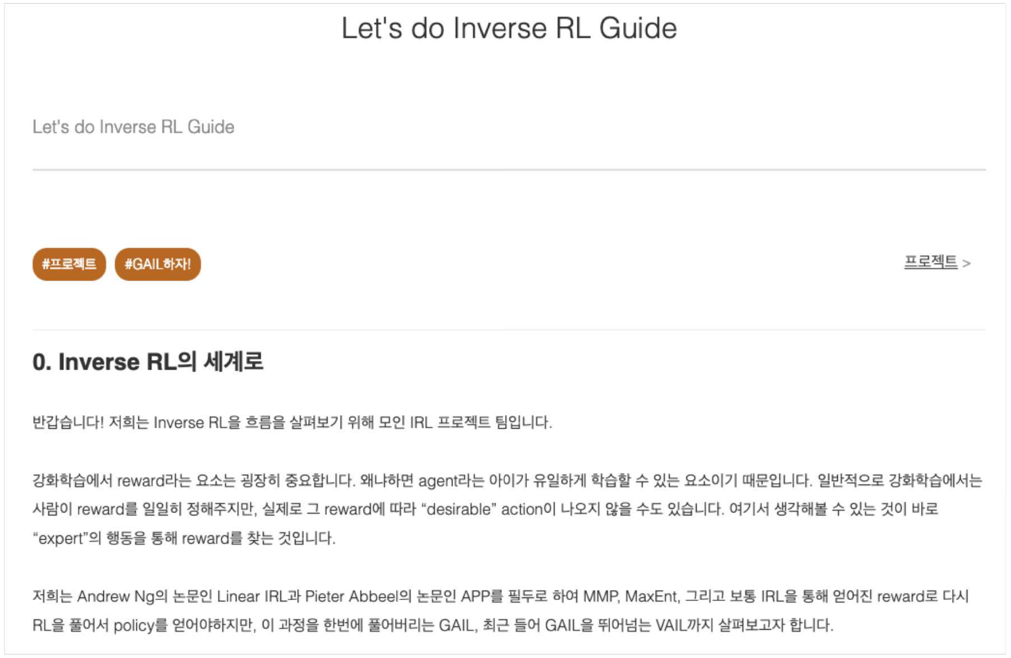
이승현

이건희

김준태

김예찬

블로그 정리 및 코드 구현



Let's do Inverse RL Guide

Let's do Inverse RL Guide

#프로젝트 #GAIL하자!

프로젝트 >

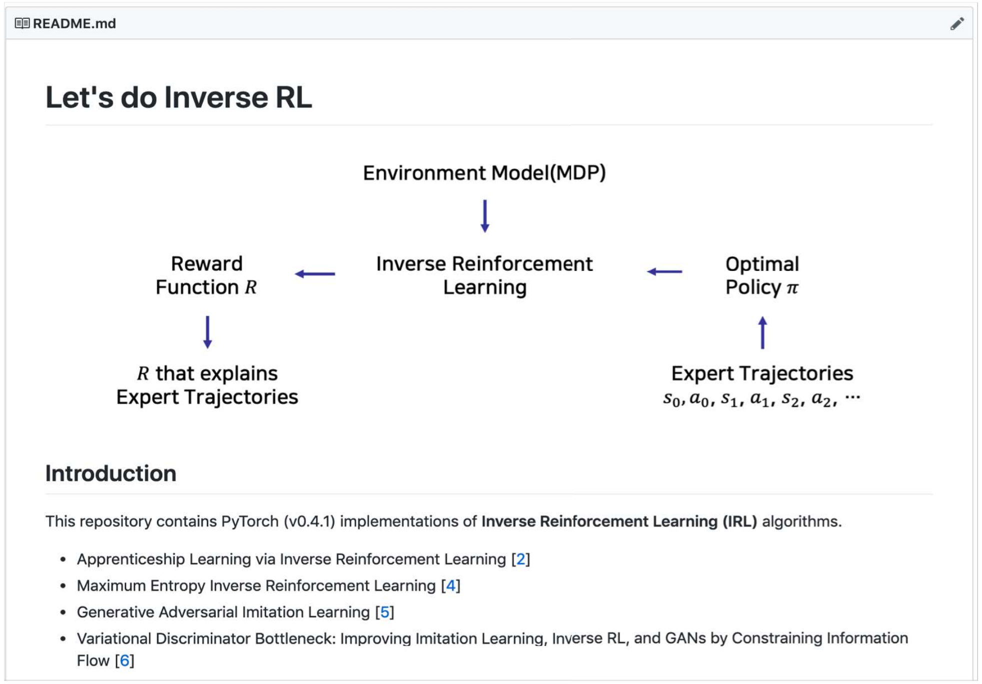
0. Inverse RL의 세계로

반갑습니다! 저희는 Inverse RL을 흐름을 살펴보기 위해 모인 IRL 프로젝트 팀입니다.

강화학습에서 reward라는 요소는 굉장히 중요합니다. 왜냐하면 agent라는 아이가 유일하게 학습할 수 있는 요소이기 때문입니다. 일반적으로 강화학습에서는 사람이 reward를 일일히 정해주지만, 실제로 그 reward에 따라 "desirable" action이 나오지 않을 수도 있습니다. 여기서 생각해볼 수 있는 것이 바로 "expert"의 행동을 통해 reward를 찾는 것입니다.

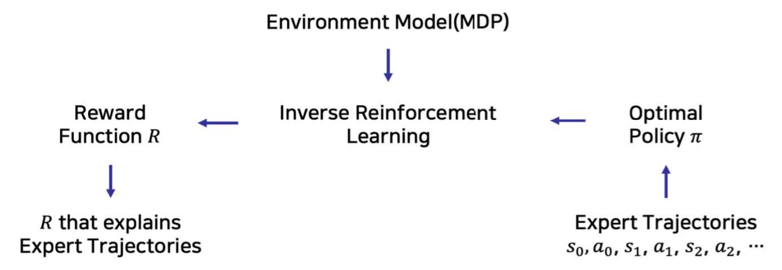
저희는 Andrew Ng의 논문인 Linear IRL과 Pieter Abbeel의 논문인 APP를 필두로 하여 MMP, MaxEnt, 그리고 보통 IRL을 통해 얻어진 reward로 다시 RL을 풀어서 policy를 얻어야하지만, 이 과정을 한번에 풀어버리는 GAIL, 최근 들어 GAIL을 뛰어넘는 VAIL까지 살펴보고자 합니다.

https://reinforcement-learning-kr.github.io/2019/01/22/0_lets-do-irl-guide/



README.md

Let's do Inverse RL


Environment Model(MDP)
Reward Function R
Inverse Reinforcement Learning
Optimal Policy π
R that explains Expert Trajectories
Expert Trajectories
$s_0, a_0, s_1, a_1, s_2, a_2, \cdots$

Introduction

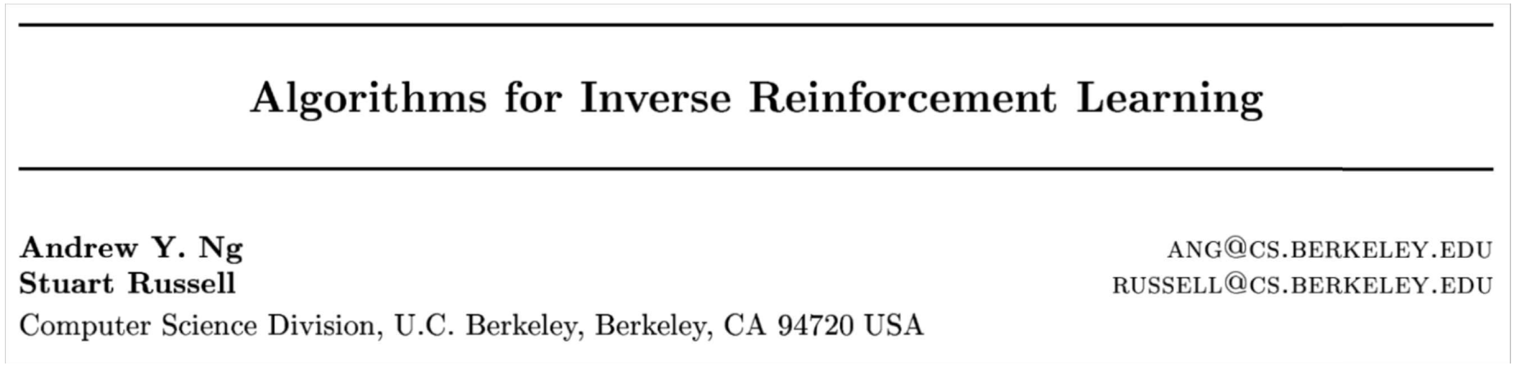
This repository contains PyTorch (v0.4.1) implementations of **Inverse Reinforcement Learning (IRL)** algorithms.

- Apprenticeship Learning via Inverse Reinforcement Learning [2]
- Maximum Entropy Inverse Reinforcement Learning [4]
- Generative Adversarial Imitation Learning [5]
- Variational Discriminator Bottleneck: Improving Imitation Learning, Inverse RL, and GANs by Constraining Information Flow [6]

https://github.com/reinforcement-learning-kr/lets-do-irl

논문 소개

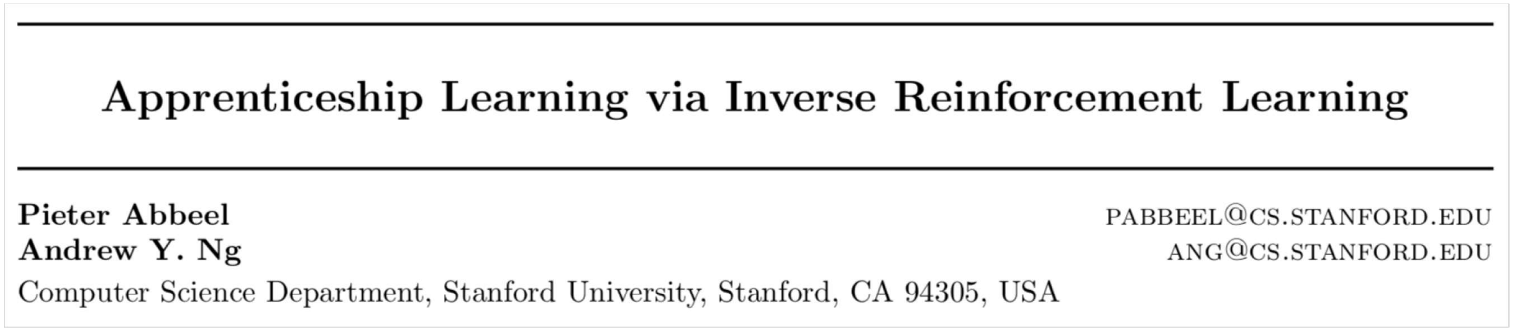
논문 소개



Algorithms for Inverse Reinforcement Learning

Andrew Y. Ng ANG@CS.BERKELEY.EDU
Stuart Russell RUSSELL@CS.BERKELEY.EDU
Computer Science Division, U.C. Berkeley, Berkeley, CA 94720 USA

1. Linear IRL (2000)

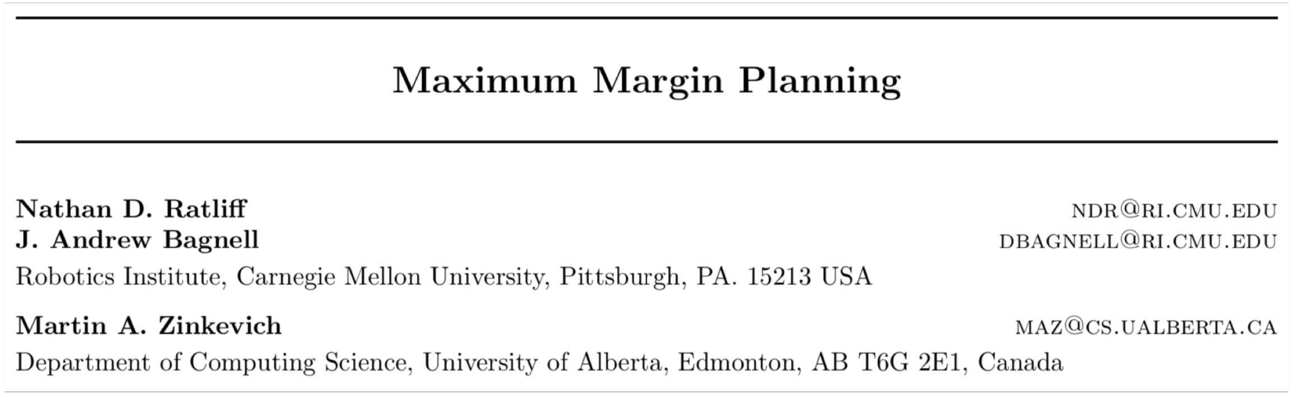


Apprenticeship Learning via Inverse Reinforcement Learning

Pieter Abbeel PABBEEL@CS.STANFORD.EDU
Andrew Y. Ng ANG@CS.STANFORD.EDU
Computer Science Department, Stanford University, Stanford, CA 94305, USA

2. APP (2004)

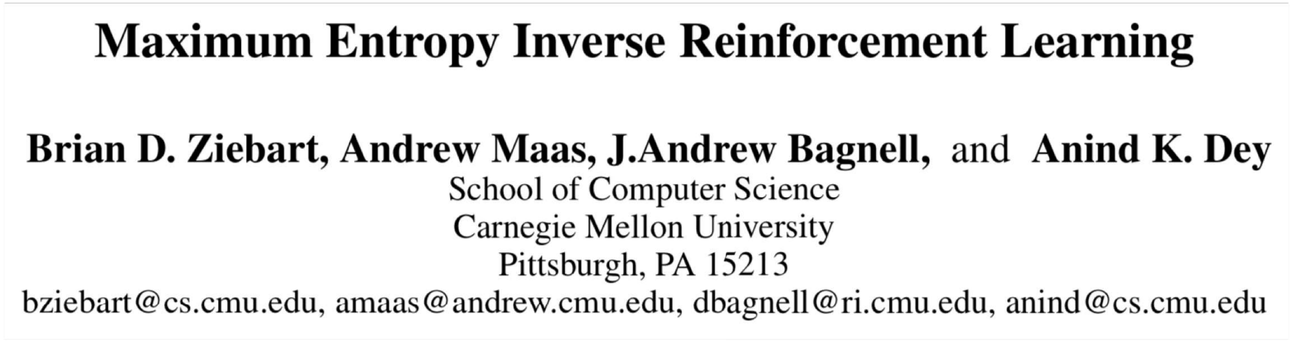
논문 소개


Maximum Margin Planning

Nathan D. Ratliff NDR@RI.CMU.EDU
J. Andrew Bagnell DBAGNELL@RI.CMU.EDU
Robotics Institute, Carnegie Mellon University, Pittsburgh, PA. 15213 USA

Martin A. Zinkevich MAZ@CS.UALBERTA.CA
Department of Computing Science, University of Alberta, Edmonton, AB T6G 2E1, Canada

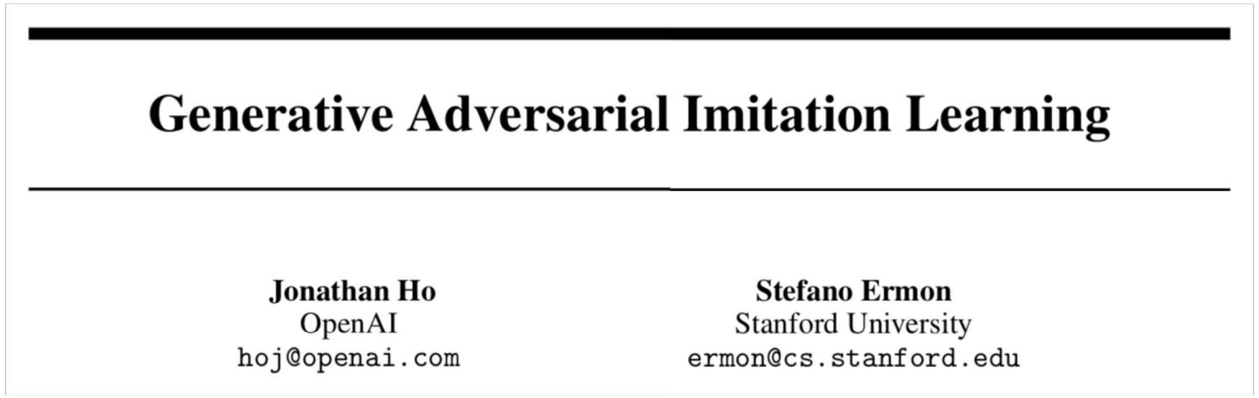
3. MMP (2006)


Maximum Entropy Inverse Reinforcement Learning

Brian D. Ziebart, Andrew Maas, J.Andrew Bagnell, and Anind K. Dey
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
bziebart@cs.cmu.edu, amaas@andrew.cmu.edu, dbagnell@ri.cmu.edu, anind@cs.cmu.edu

4. MaxEnt (2008)

논문 소개

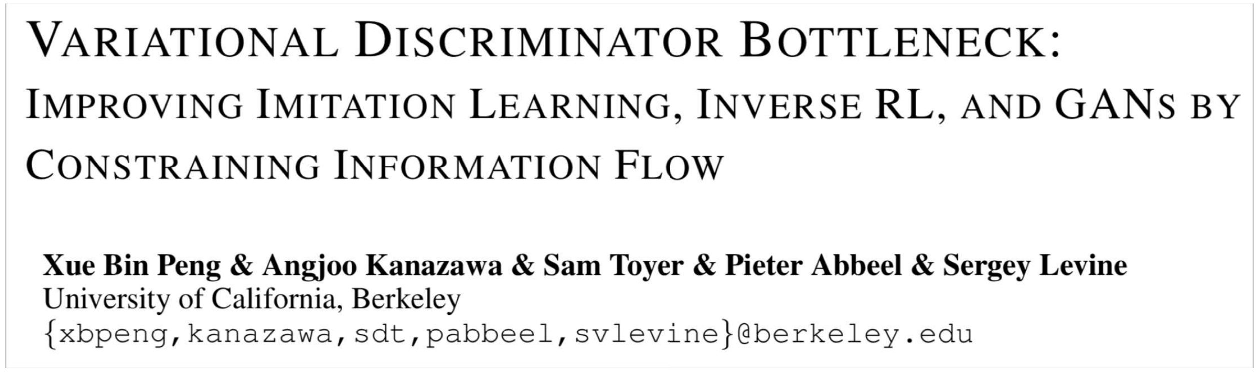


Generative Adversarial Imitation Learning

Jonathan Ho
OpenAI
hoj@openai.com

Stefano Ermon
Stanford University
ermon@cs.stanford.edu

5. GAIL (2016)



VARIATIONAL DISCRIMINATOR BOTTLENECK: IMPROVING IMITATION LEARNING, INVERSE RL, AND GANS BY CONSTRAINING INFORMATION FLOW

Xue Bin Peng & Angjoo Kanazawa & Sam Toyer & Pieter Abbeel & Sergey Levine
University of California, Berkeley
{xbpeng,kanazawa,sdt,pabbeel,svlevine}@berkeley.edu

6. VAIL (2018)

논문은 구현한 것을 중심으로 설명하려고 합니다!

구현한 알고리즘 -> APP, MaxEnt, GAIL, VAIL

1. Linear IRL

Algorithms for Inverse Reinforcement Learning (2000)

References – Linear IRL

- Algorithms for Inverse Reinforcement Learning (Paper Link)
- Cool Inverse Reinforcement Learning (IRL) Papers - IRL survey (최성준 박사님) (PDF Link)
- Introduction of Inverse Reinforcement Learning (곽동현님) (PPT Link)

Summary - Linear IRL

- Imitation Learning(or Learning from Demonstrations(LfD))
    - What is the Imitation Learning?
    - Why do you need the Imitation Learning?

- What is the Inverse RL?

- Principles of Inverse RL

- Proposed Algorithm
  1) state space is finite, model is known
  2) state space is large or infinite, model is known
  3) the policy is known only through sampled trajectories, model is not known

- Proposed Idea : Linear Programming

Imitation Learning

The goal of Imitation Learning is to reproduce the expert's demonstrations with generalized intention.

Why do you need the Imitation Learning?

1. Reward Shaping

2. Safe Learning

3. Exploration process

Imitation Learning

Methods of Imitation Learning

1. Behavioral Cloning
2. Inverse Reinforcement Learning
3. Teacher Advice

Imitation Learning

Methods of Imitation Learning

1. Behavioral Cloning
2. $\boxed{\text{2. Inverse Reinforcement Learning}}$ - Selected!
3. Teacher Advice

강화학습이 처음이신 분께 다소 어려울 수 있으니

전체적인 흐름 파악에 집중해주시면 감사하겠습니다!

Reinforcement Learning

$$\begin{tikzcd}
 & \text{Environment Model(MDP)} \arrow[d] & \\
\begin{array}{c}\text{Reward}\\ \text{Function } R\end{array} \arrow[r] & \begin{array}{c}\text{Reinforcement}\\ \text{Learning}\\ \arg\max_\pi \mathrm{E}[\sum_t \gamma^t R(s_t)|\pi]\end{array} \arrow[r] & \begin{array}{c}\text{Optimal}\\ \text{Policy } \pi\end{array}
\end{tikzcd}$$

Inverse Reinforcement Learning

$$\begin{tikzcd}
 & \text{Environment Model(MDP)} \arrow[d] & \\
\begin{array}{c}\text{Reward}\\ \text{Function } R\end{array} \arrow[d] & \begin{array}{c}\text{Inverse Reinforcement}\\ \text{Learning}\end{array} \arrow[l] & \begin{array}{c}\text{Optimal}\\ \text{Policy } \pi\end{array} \arrow[l] \\
\begin{array}{c}R \text{ that explains}\\ \text{Expert Trajectories}\end{array} & & \begin{array}{c}\text{Expert Trajectories}\\ s_0, a_0, s_1, a_1, s_2, a_2, \cdots\end{array} \arrow[u]
\end{tikzcd}$$

Inverse Reinforcement Learning

$$\begin{tikzcd}
 & \text{Environment Model(MDP)} \arrow[d] & \\
\text{Reward} \atop \text{Function } R \arrow[d] & \text{Inverse Reinforcement} \atop \text{Learning} \arrow[l] & \text{Optimal} \atop \text{Policy } \pi \arrow[l] \\
R \text{ that explains} \atop \text{Expert Trajectories} & & \text{Expert Trajectories} \atop s_0, a_0, s_1, a_1, s_2, a_2, \cdots \arrow[u]
\end{tikzcd}$$

Principles of Inverse RL

1) IRL is fundamentally optimization problem.

2) We need more constraints / regularizations / priors
   to determine the optimal reward function.

3) Reward Function is never unique. (i.e. ill-posed problem)

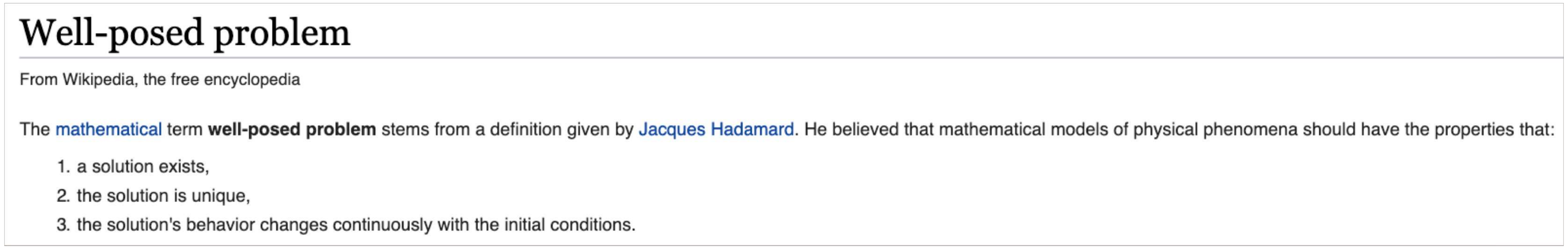
Principles of Inverse RL

1) IRL is fundamentally optimization problem.

2) We need more constraints / regularizations / priors
   to determine the optimal reward function.

3) Reward Function is never unique. (i.e. ill-posed problem)

ill-posed problem


Well-posed problem

From Wikipedia, the free encyclopedia

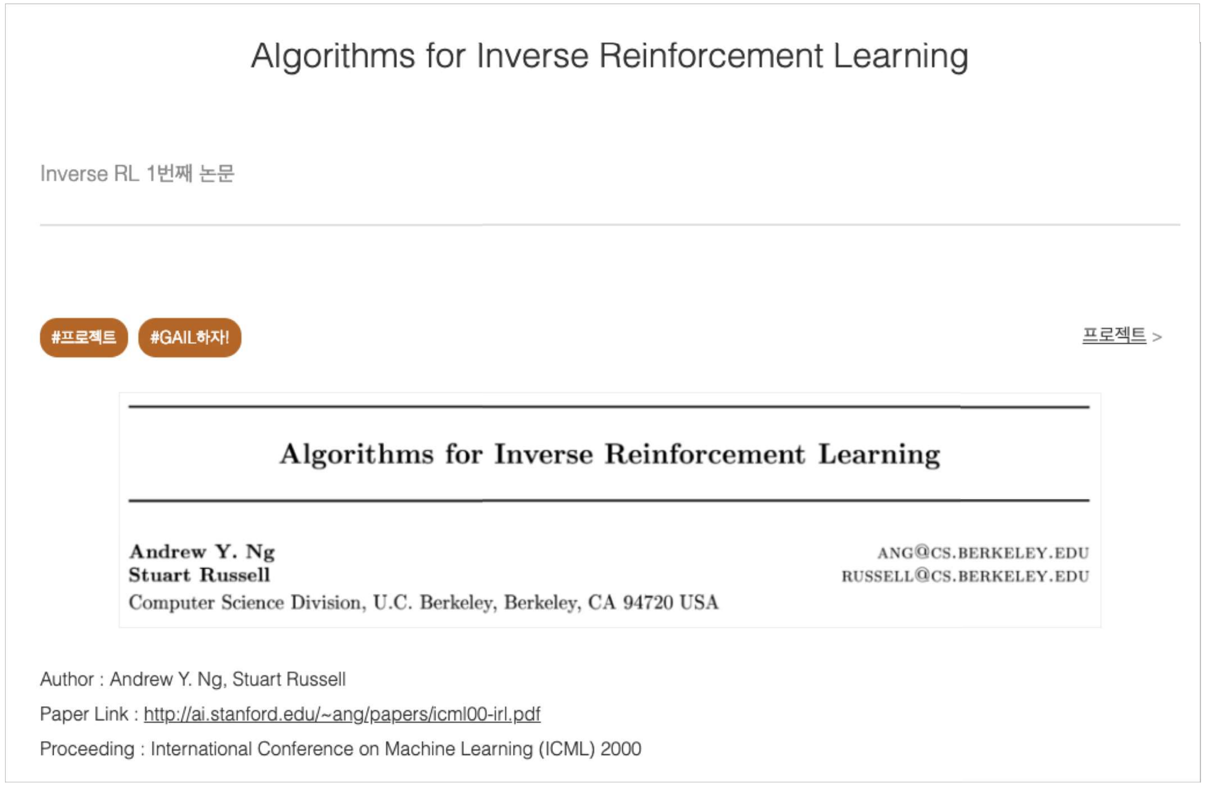
The mathematical term **well-posed problem** stems from a definition given by Jacques Hadamard. He believed that mathematical models of physical phenomena should have the properties that:

1. a solution exists,
2. the solution is unique,
3. the solution's behavior changes continuously with the initial conditions.

특정한 solution이 존재하고, 그 solution이 unique하다.

ill-posed problem

반대로, 역강화학습에서는 reward가 정해진 것이 아니라
여러가지 형태의 값으로 나타날 수 있기 때문에
ill-posed problem



Algorithms for Inverse Reinforcement Learning

Inverse RL 1번째 논문

#프로젝트 #GAIL하자!

프로젝트 >

Algorithms for Inverse Reinforcement Learning

Andrew Y. Ng ANG@CS.BERKELEY.EDU
Stuart Russell RUSSELL@CS.BERKELEY.EDU
Computer Science Division, U.C. Berkeley, Berkeley, CA 94720 USA

Author : Andrew Y. Ng, Stuart Russell

Paper Link : http://ai.stanford.edu/~ang/papers/icml00-irl.pdf

Proceeding : International Conference on Machine Learning (ICML) 2000

Linear IRL 정리

알고리즘에 대한 추가적인 내용은 블로그를 참고해주세요!

2. APP

Apprenticeship Learning via Inverse Reinforcement Learning (2004)

References – APP

- Apprenticeship Learning via Inverse Reinforcement Learning (Paper Link)
- Cool Inverse Reinforcement Learning (IRL) Papers - IRL survey (최성준 박사님) (PDF Link)
- Introduction of Inverse Reinforcement Learning (곽동현님) (PPT Link)
- PR-029: Apprenticeship Learning via Inverse Reinforcement Learning (서기호님) (PPT Link)
- LP 문제와 QP 문제 (Blog Link)
- Apprenticeship learning using Inverse Reinforcement Learning (Blog Link)
- Using IRL to train a toy car in a 2D game to learn given Expert Behaviors (GitHub Link)

Summary - APP

- Feature Expectation Matching
- Proposed Algorithm : Apprenticeship Learning (IRL -> RL)
- Proposed Idea : Linear Programming, Quadratic Programming(or SVM), Projection Method

Preliminaries

- MDP = (S, A, T, γ, D, R)
  - S : finite set of states
  - A : set of actions
  - $T = \{P_{sa}\}$: set of state transition probabilities
  - $\gamma \in [0,1)$: discount factor
  - D : initial-state distribution
  - R : reward function
- Vector of features $\phi : S \rightarrow [0,1]^k$
- "True" reward function $R^*(s) = w^* \cdot \phi(s)$, where $w^* \in \mathbb{R}^k$

Preliminaries

- A policy π is a mapping from states to probability distributions over actions.
  The value of a policy π is

$$\begin{aligned} E_{s_0 \sim D}[V^\pi(s_0)] &= E[\textstyle\sum_{t=0}^{\infty} \gamma^t R(s_t)|\pi] \\ &= E[\textstyle\sum_{t=0}^{\infty} \gamma^t w \cdot \phi(s_t)|\pi] \\ &= w \cdot E[\textstyle\sum_{t=0}^{\infty} \gamma^t \phi(s_t)|\pi] \end{aligned}$$

- The expected discounted accumulated feature value vector $\mu(\pi)$ = Feature expectations

$$\mu(\pi) = E[\textstyle\sum_{t=0}^{\infty} \gamma^t \phi(s_t)|\pi] \in \mathbb{R}^k$$

- Using this notation, the value of a policy may be written

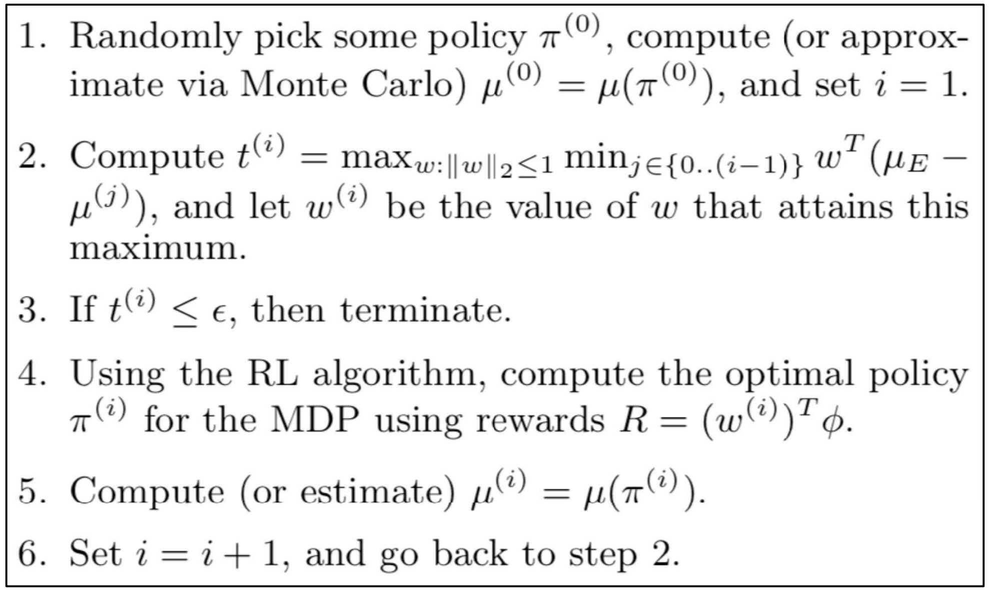
$$E_{s_0 \sim D}[V^\pi(s_0)] = w \cdot \mu(\pi)$$

- Estimate of the expert's feature expectations $\hat{\mu}_E = \mu(\pi_E)$. Given a set of m trajectories $\{s_0^{(i)}, s_1^{(i)}, \ldots\}_{i=1}^m$,

$$\hat{\mu}_E = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{\infty} \gamma^t \phi(s_t^{(i)})$$

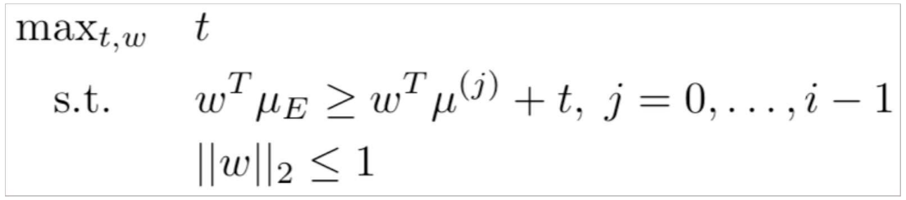
Proposed Algorithm

- Apprenticeship Learning Algorithm



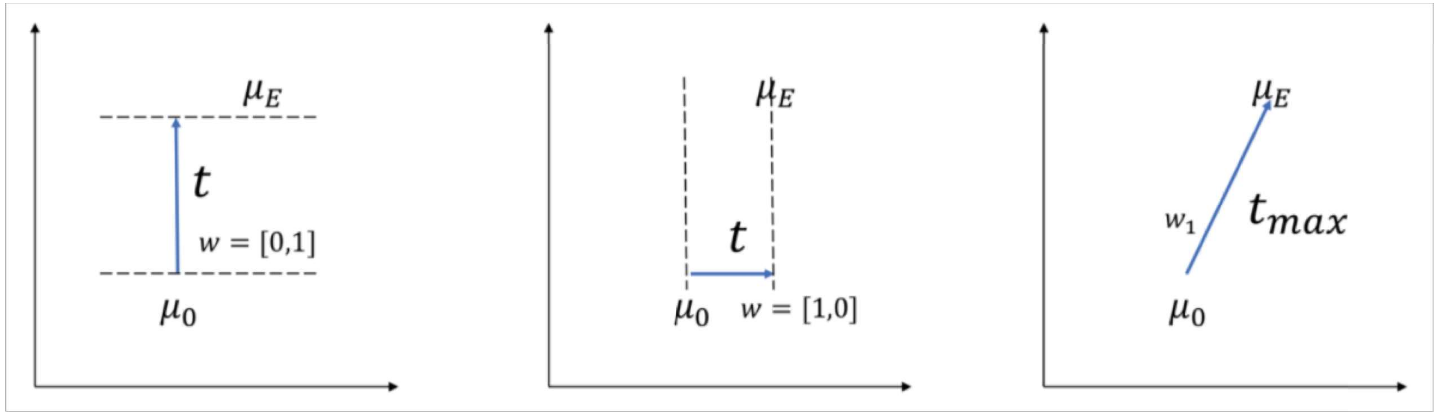
1. Randomly pick some policy $\pi^{(0)}$, compute (or approximate via Monte Carlo) $\mu^{(0)} = \mu(\pi^{(0)})$, and set $i = 1$.
2. Compute $t^{(i)} = \max_{w:\|w\|_2 \le 1} \min_{j \in \{0..(i-1)\}} w^T(\mu_E - \mu^{(j)})$, and let $w^{(i)}$ be the value of w that attains this maximum.
3. If $t^{(i)} \le \epsilon$, then terminate.
4. Using the RL algorithm, compute the optimal policy $\pi^{(i)}$ for the MDP using rewards $R = (w^{(i)})^T \phi$.
5. Compute (or estimate) $\mu^{(i)} = \mu(\pi^{(i)})$.
6. Set $i = i + 1$, and go back to step 2.

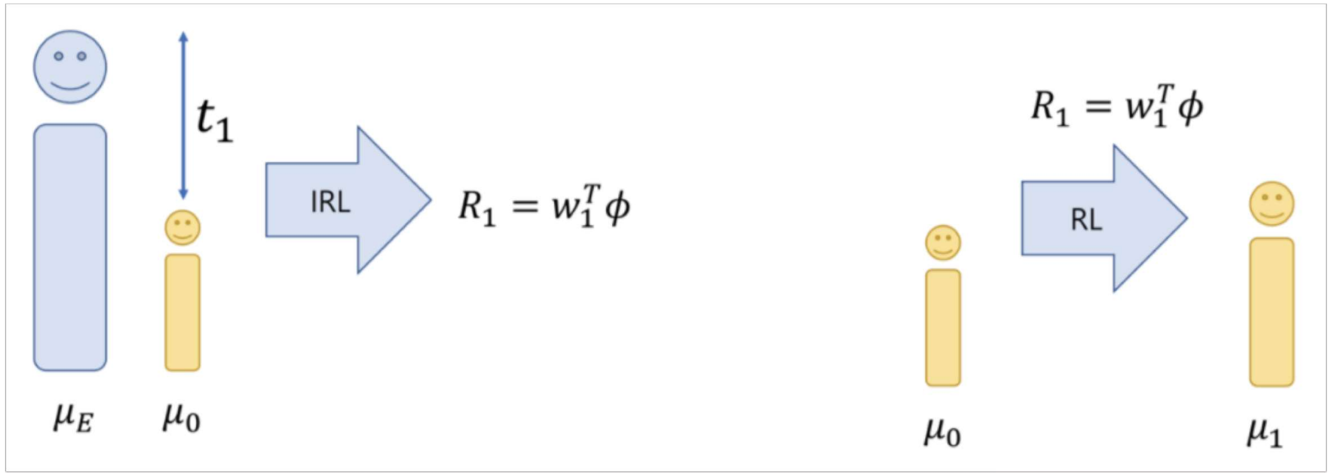
- Maximization step using Linear Programming



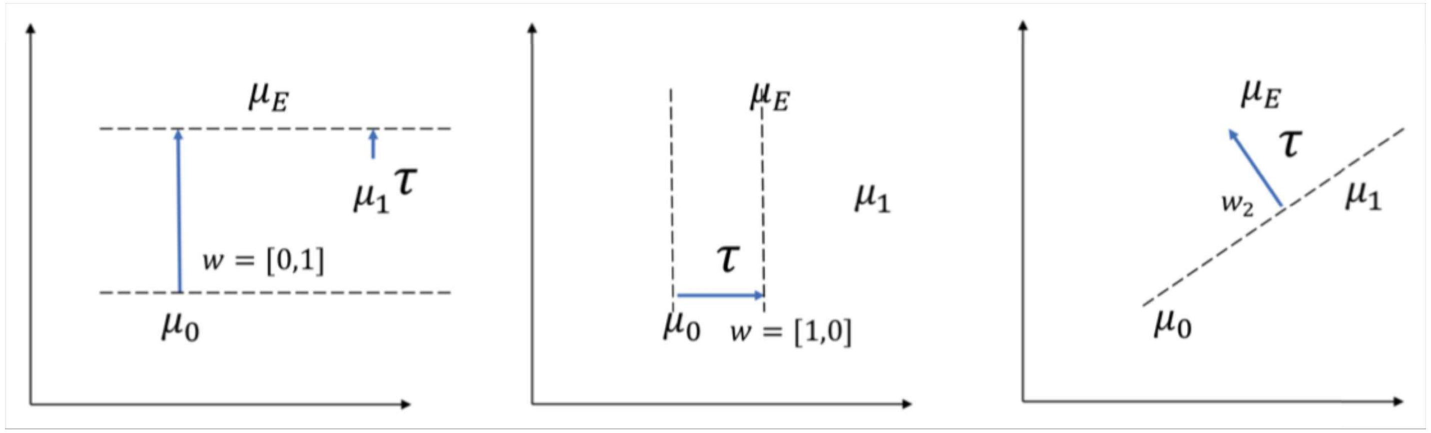
$$\begin{aligned} \max_{t,w} \quad & t \\ \text{s.t.} \quad & w^T \mu_E \ge w^T \mu^{(j)} + t, \; j = 0, \ldots, i-1 \\ & ||w||_2 \le 1 \end{aligned}$$

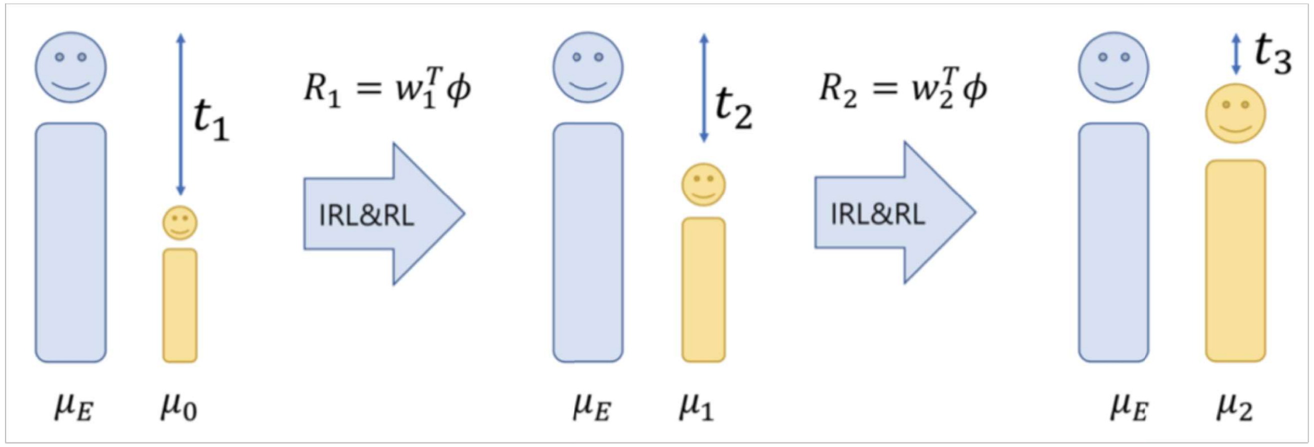
- Start with μ_0



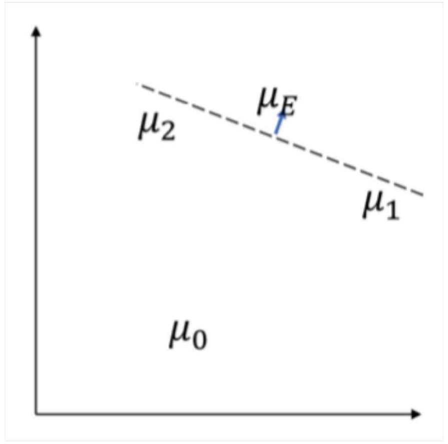


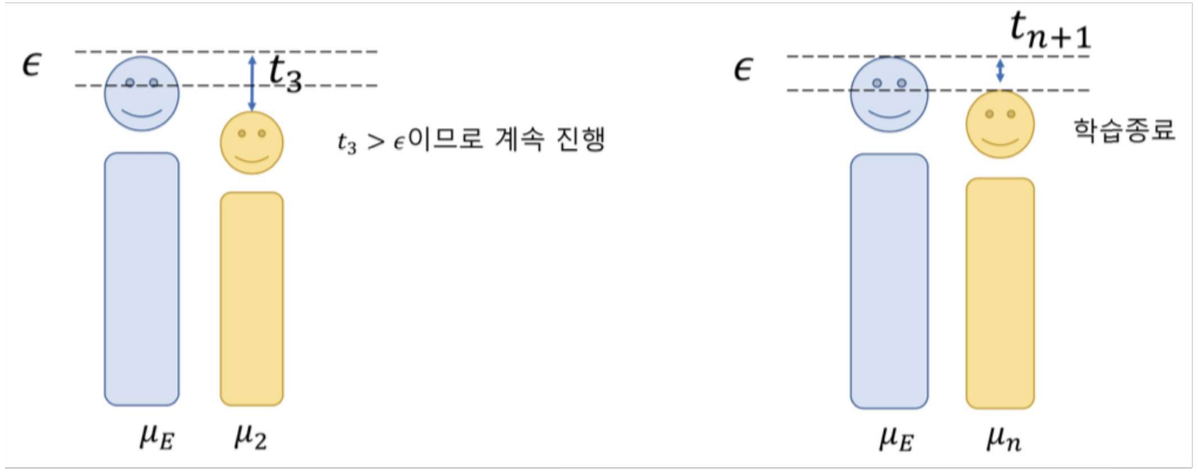
- Next, from μ_0 to μ_1





- Finally, from μ_0 to μ_2





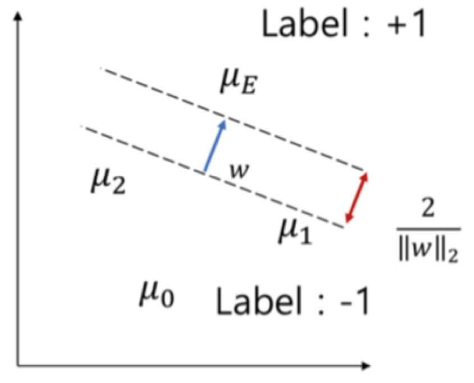
Proposed Algorithm

- Maximization step using Linear Programming

$$\begin{aligned} \max_{t,w} \quad & t \\ \text{s.t.} \quad & w^T \mu_E \geq w^T \mu^{(j)} + t, \; j = 0, \ldots, i-1 \\ & \underline{\|w\|_2 \leq 1} \end{aligned}$$

L2 constraint on w cannot be posed as a LP! ㅜ_ㅜ

- Quadratic Programming(or SVM)



$$\max\left(t = \frac{2}{\|w\|_2}\right) \Rightarrow \min \|w\|_2^2$$

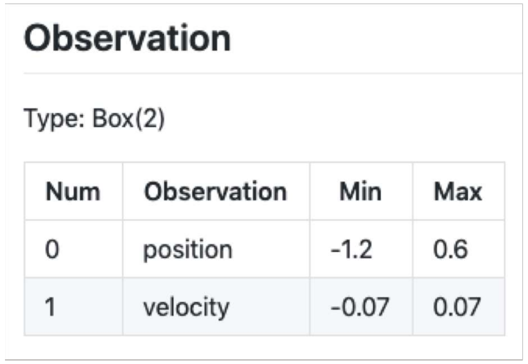
Quadratic loss

$$\min \|w\|_2^2$$

$$s.t \;\; w^T \mu_E \geq w^T \mu_i + 2$$

APP의 구현이야기

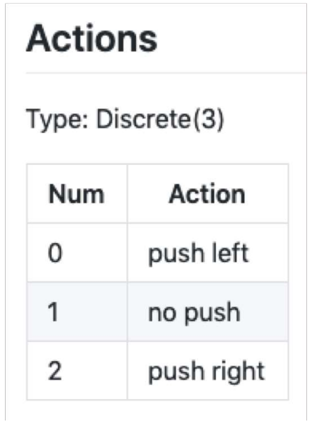
- Environment : Mountain Car v0
- Continuous observation spaces, Discrete action spaces



Observation

Type: Box(2)

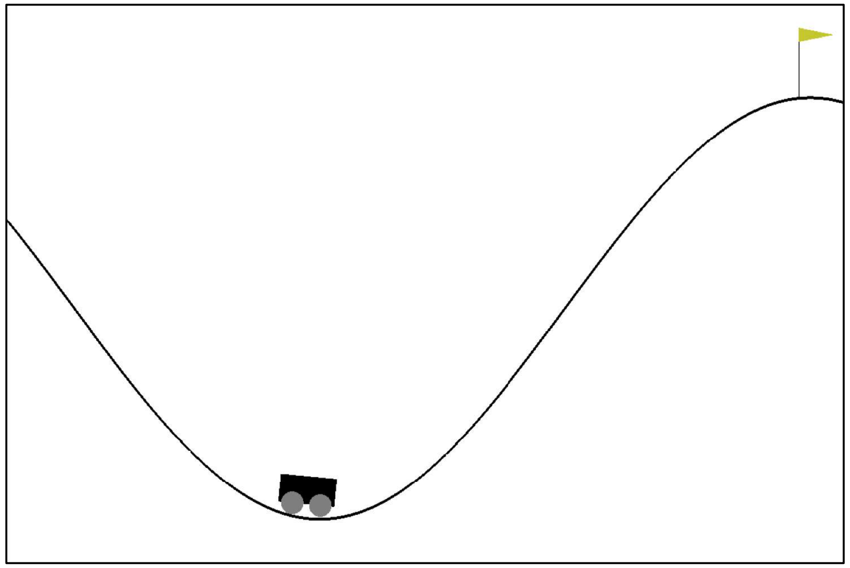
Num	Observation	Min	Max
0	position	-1.2	0.6
1	velocity	-0.07	0.07



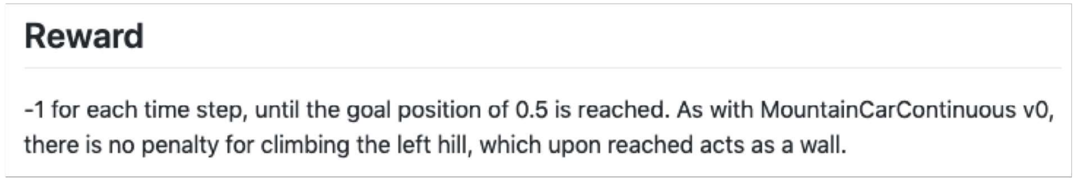
Actions

Type: Discrete(3)

Num	Action
0	push left
1	no push
2	push right



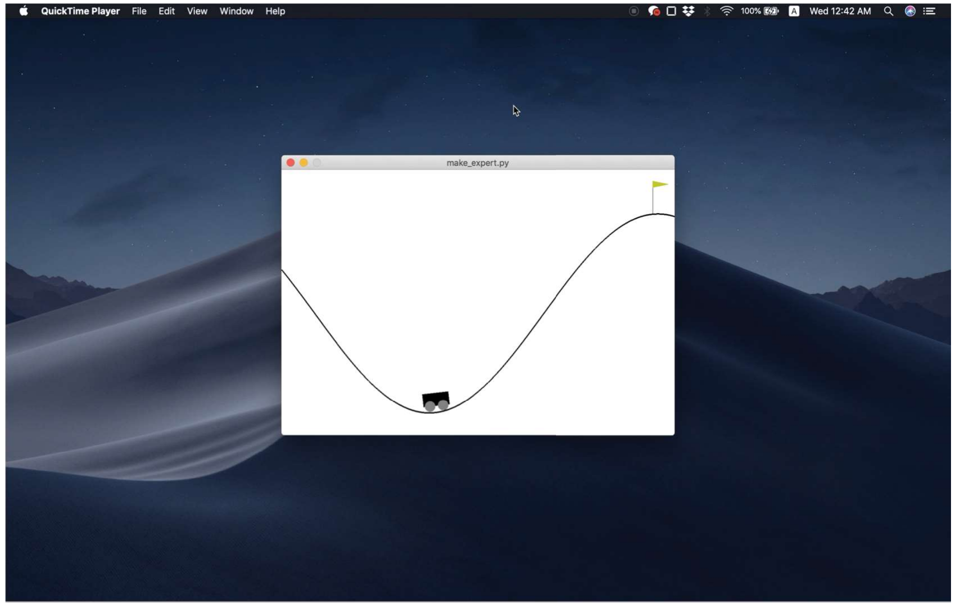
- Reward function

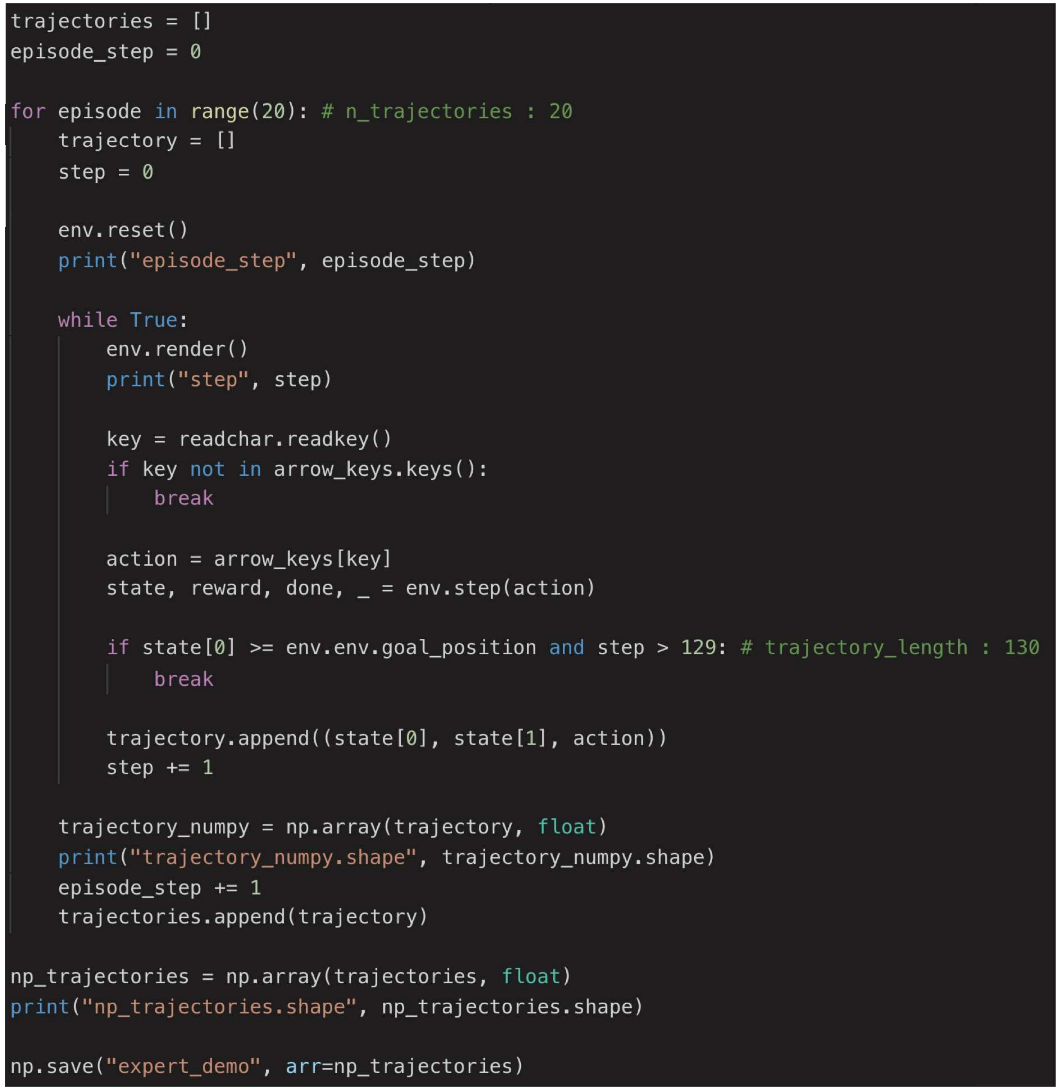


Reward

-1 for each time step, until the goal position of 0.5 is reached. As with MountainCarContinuous v0, there is no penalty for climbing the left hill, which upon reached acts as a wall.

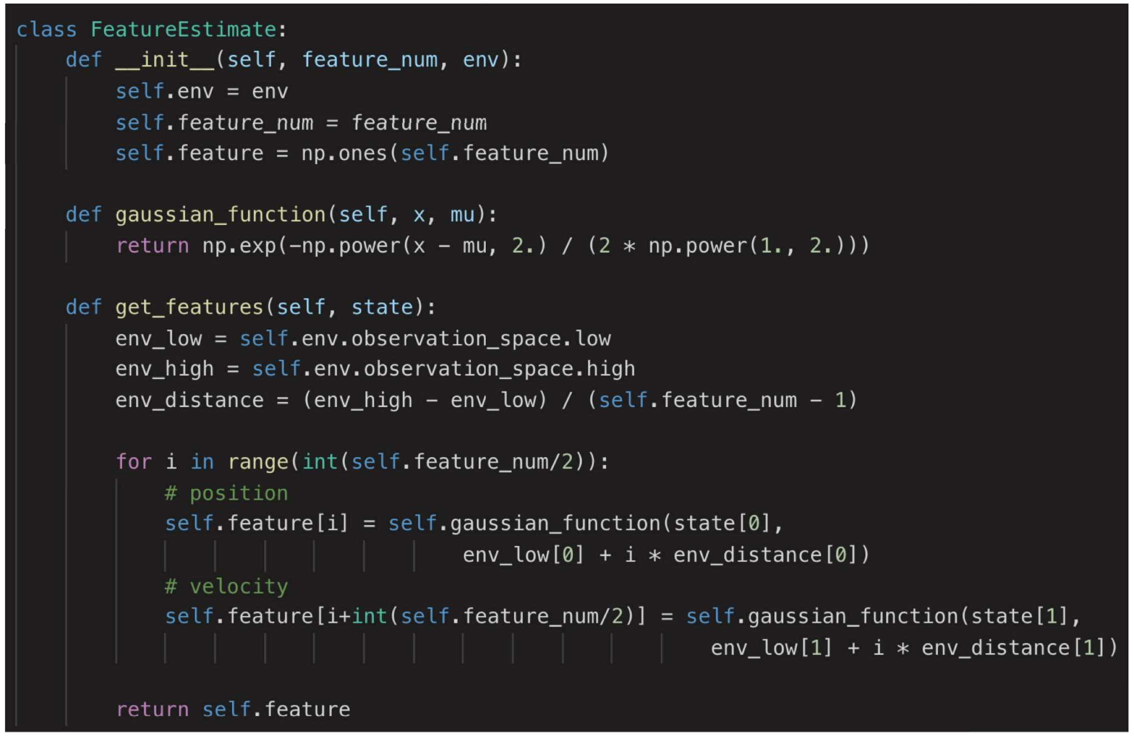
- Demonstraions of 동민's expert
  - (20, 130, 3)
  - (demo_num, demo_length, state + action)





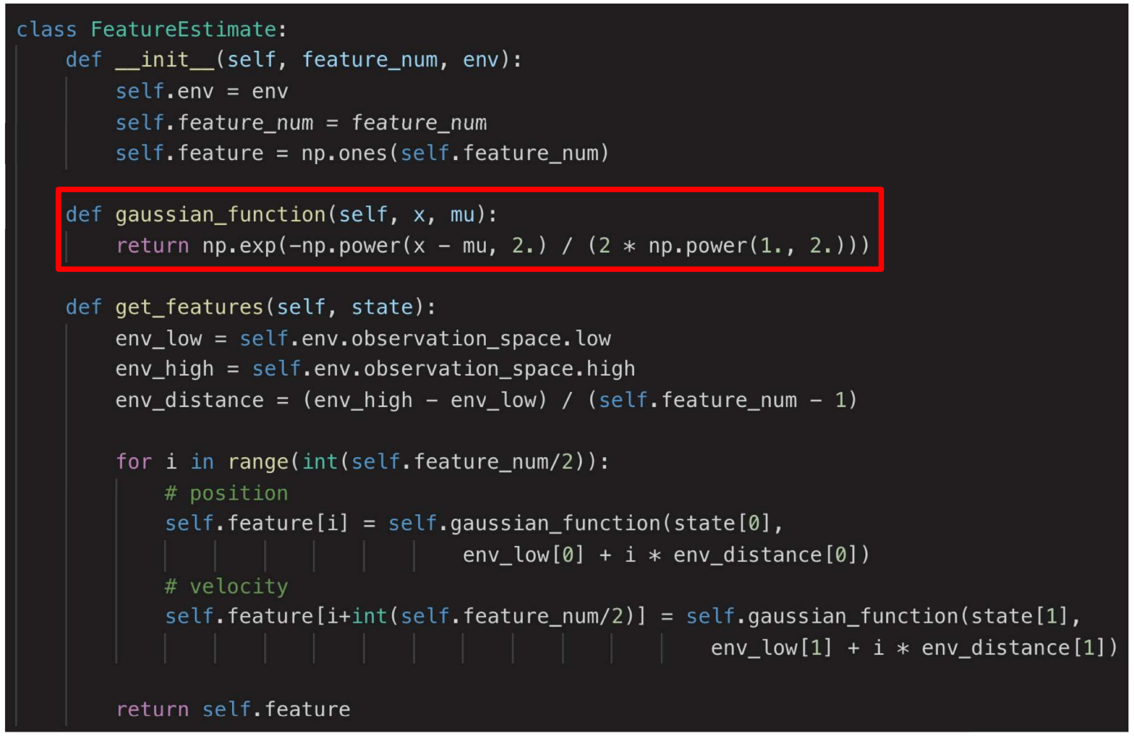
```python
trajectories = []
episode_step = 0

for episode in range(20): # n_trajectories : 20
    trajectory = []
    step = 0

    env.reset()
    print("episode_step", episode_step)

    while True:
        env.render()
        print("step", step)

        key = readchar.readkey()
        if key not in arrow_keys.keys():
            break

        action = arrow_keys[key]
        state, reward, done, _ = env.step(action)

        if state[0] >= env.env.goal_position and step > 129: # trajectory_length : 130
            break

        trajectory.append((state[0], state[1], action))
        step += 1

    trajectory_numpy = np.array(trajectory, float)
    print("trajectory_numpy.shape", trajectory_numpy.shape)
    episode_step += 1
    trajectories.append(trajectory)

np_trajectories = np.array(trajectories, float)
print("np_trajectories.shape", np_trajectories.shape)

np.save("expert_demo", arr=np_trajectories)
```

- Vector of features $\phi : S \rightarrow [0,1]^k$
    - 구현할 때, k = 4 = feature_num
- “True” reward function $R^*(s) = w^* \cdot \phi(s)$



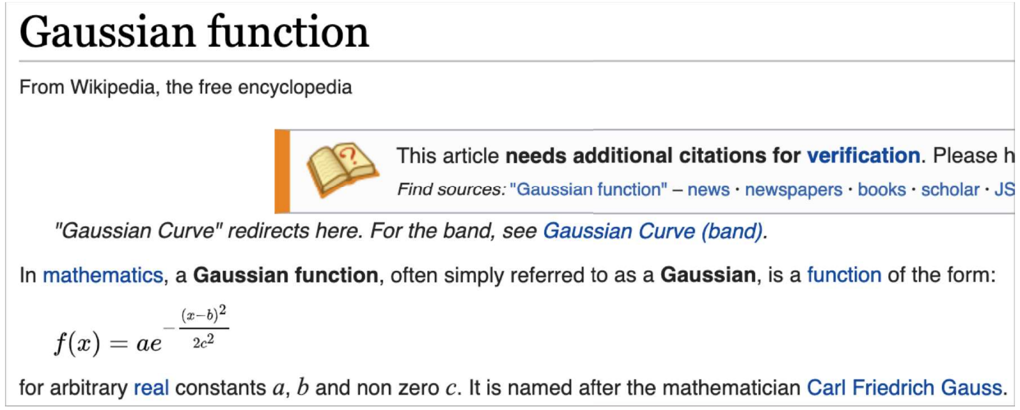
```python
class FeatureEstimate:
    def __init__(self, feature_num, env):
        self.env = env
        self.feature_num = feature_num
        self.feature = np.ones(self.feature_num)

    def gaussian_function(self, x, mu):
        return np.exp(-np.power(x - mu, 2.) / (2 * np.power(1., 2.)))

    def get_features(self, state):
        env_low = self.env.observation_space.low
        env_high = self.env.observation_space.high
        env_distance = (env_high - env_low) / (self.feature_num - 1)

        for i in range(int(self.feature_num/2)):
            # position
            self.feature[i] = self.gaussian_function(state[0],
                                        env_low[0] + i * env_distance[0])
            # velocity
            self.feature[i+int(self.feature_num/2)] = self.gaussian_function(state[1],
                                                        env_low[1] + i * env_distance[1])

        return self.feature
```

- Vector of features $\phi : S \to [0,1]^k$
  - 구현할 때, k = 4 = feature_num
- "True" reward function $R^*(s) = w^* \cdot \phi(s)$



```python
class FeatureEstimate:
    def __init__(self, feature_num, env):
        self.env = env
        self.feature_num = feature_num
        self.feature = np.ones(self.feature_num)

    def gaussian_function(self, x, mu):
        return np.exp(-np.power(x - mu, 2.) / (2 * np.power(1., 2.)))

    def get_features(self, state):
        env_low = self.env.observation_space.low
        env_high = self.env.observation_space.high
        env_distance = (env_high - env_low) / (self.feature_num - 1)

        for i in range(int(self.feature_num/2)):
            # position
            self.feature[i] = self.gaussian_function(state[0],
                                    env_low[0] + i * env_distance[0])
            # velocity
            self.feature[i+int(self.feature_num/2)] = self.gaussian_function(state[1],
                                    env_low[1] + i * env_distance[1])

        return self.feature
```

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}}$$



Gaussian function

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please h
Find sources: "Gaussian function" – news · newspapers · books · scholar · JS

"Gaussian Curve" redirects here. For the band, see Gaussian Curve (band).

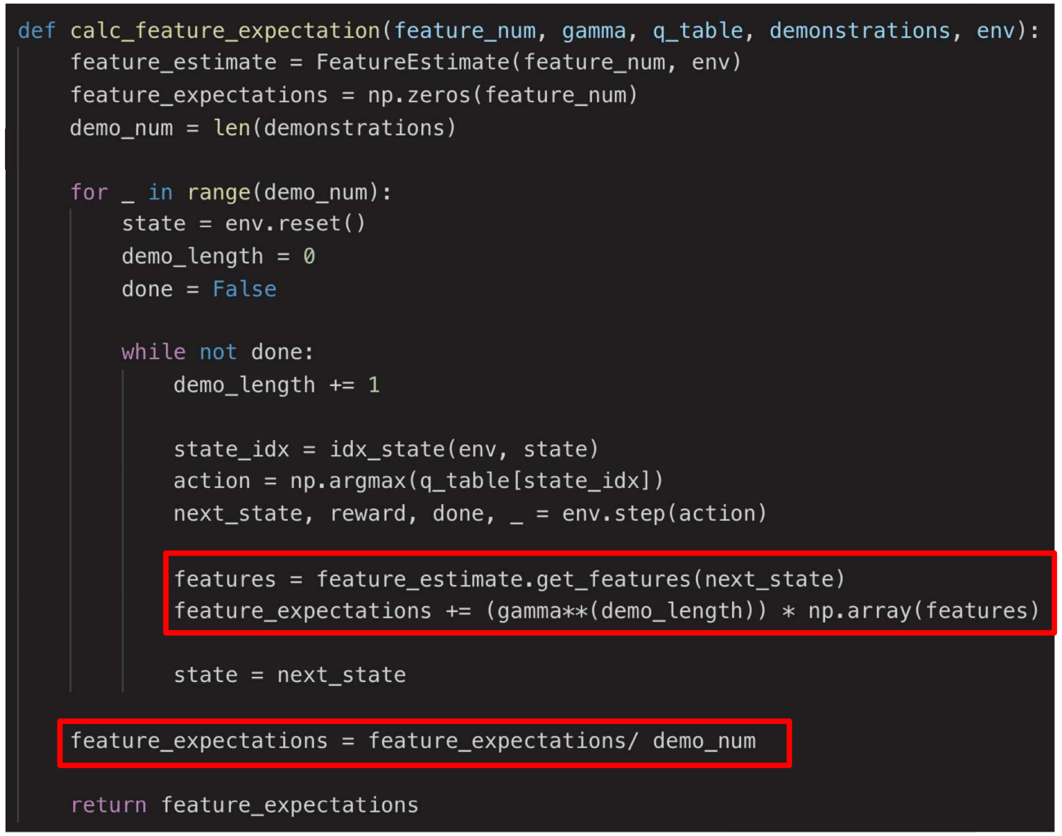
In mathematics, a **Gaussian function**, often simply referred to as a **Gaussian**, is a function of the form:

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}}$$

for arbitrary real constants a, b and non zero c. It is named after the mathematician Carl Friedrich Gauss.

- Vector of features $\phi : S \rightarrow [0,1]^k$
- "True" reward function $R^*(s) = w^* \cdot \phi(s)$
- $\mu(\pi)$ = Learner's Feature expectations

$$\mu(\pi) = E[\sum_{t=0}^{\infty} \gamma^t \phi(s_t) | \pi] \in \mathbb{R}^k$$



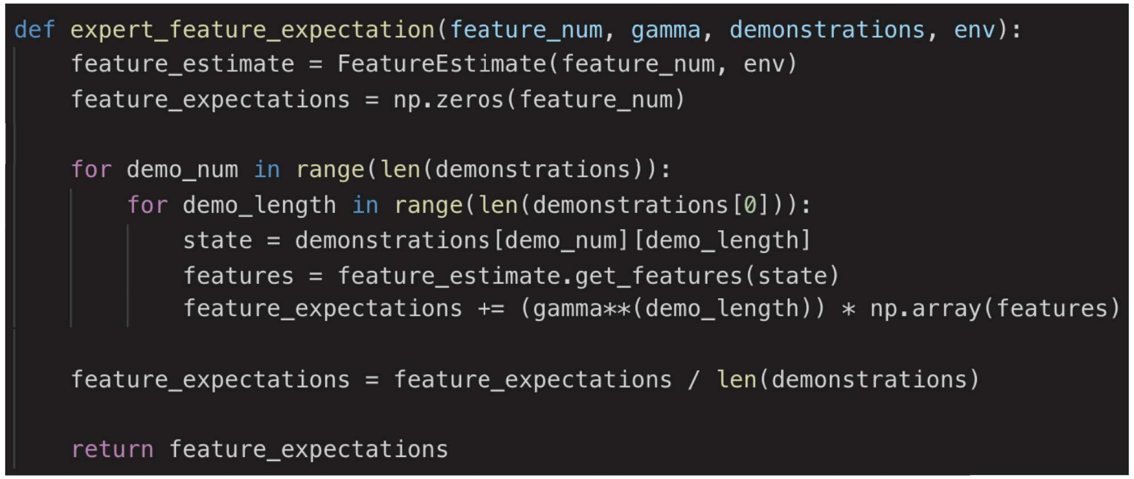
```python
def calc_feature_expectation(feature_num, gamma, q_table, demonstrations, env):
    feature_estimate = FeatureEstimate(feature_num, env)
    feature_expectations = np.zeros(feature_num)
    demo_num = len(demonstrations)

    for _ in range(demo_num):
        state = env.reset()
        demo_length = 0
        done = False

        while not done:
            demo_length += 1

            state_idx = idx_state(env, state)
            action = np.argmax(q_table[state_idx])
            next_state, reward, done, _ = env.step(action)

            features = feature_estimate.get_features(next_state)
            feature_expectations += (gamma**(demo_length)) * np.array(features)

            state = next_state

    feature_expectations = feature_expectations/ demo_num

    return feature_expectations
```

- Vector of features $\phi : S \rightarrow [0,1]^k$
- "True" reward function $R^*(s) = w^* \cdot \phi(s)$
- $\mu(\pi)$ = Learner's Feature expectations

$$\mu(\pi) = E[\sum_{t=0}^{\infty} \gamma^t \phi(s_t)|\pi] \in \mathbb{R}^k$$

- $\hat{\mu}(\pi)$ = Expert's Feature expectations

$$\hat{\mu}_E = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{\infty} \gamma^t \phi(s_t^{(i)})$$



```python
def expert_feature_expectation(feature_num, gamma, demonstrations, env):
    feature_estimate = FeatureEstimate(feature_num, env)
    feature_expectations = np.zeros(feature_num)

    for demo_num in range(len(demonstrations)):
        for demo_length in range(len(demonstrations[0])):
            state = demonstrations[demo_num][demo_length]
            features = feature_estimate.get_features(state)
            feature_expectations += (gamma**(demo_length)) * np.array(features)

    feature_expectations = feature_expectations / len(demonstrations)

    return feature_expectations
```

- Vector of features $\phi : S \rightarrow [0,1]^k$
- “True” reward function $R^*(s) = w^* \cdot \phi(s)$
- $\mu(\pi)$ = Learner’s Feature expectations

$$\mu(\pi) = E[\sum_{t=0}^{\infty} \gamma^t \phi(s_t)|\pi] \in \mathbb{R}^k$$

- $\hat{\mu}(\pi)$ = Expert’s Feature expectations

$$\hat{\mu}_E = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{\infty} \gamma^t \phi(s_t^{(i)})$$

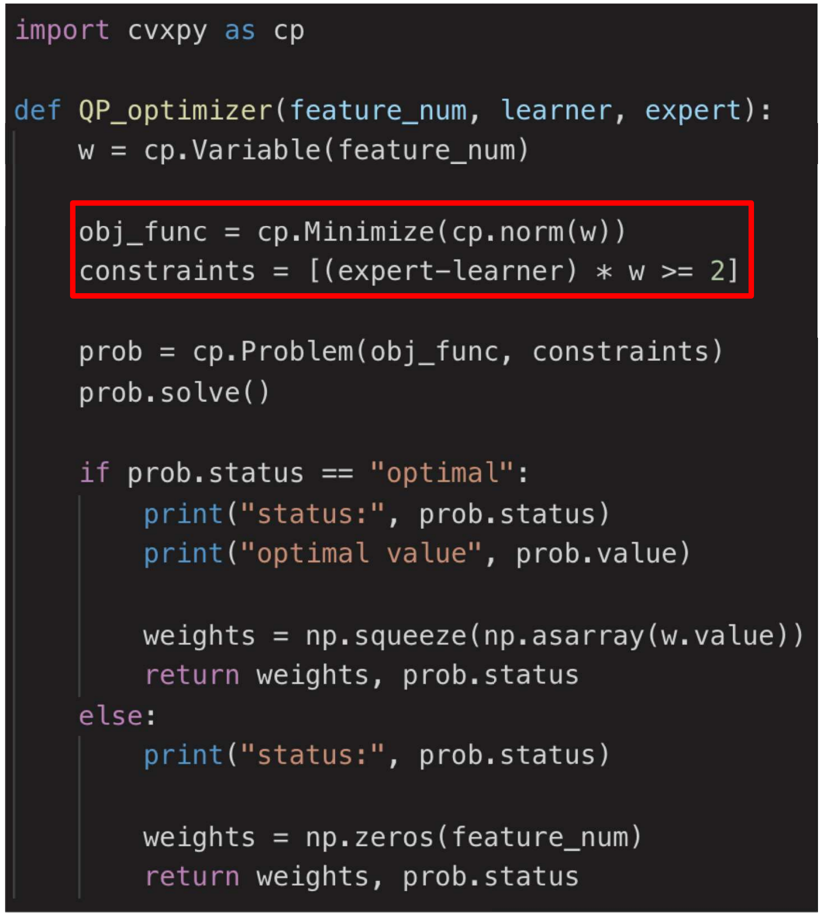
- Quadratic loss

$$\max(t = \frac{2}{\|w\|_2}) \Rightarrow \min\|w\|_2^2$$

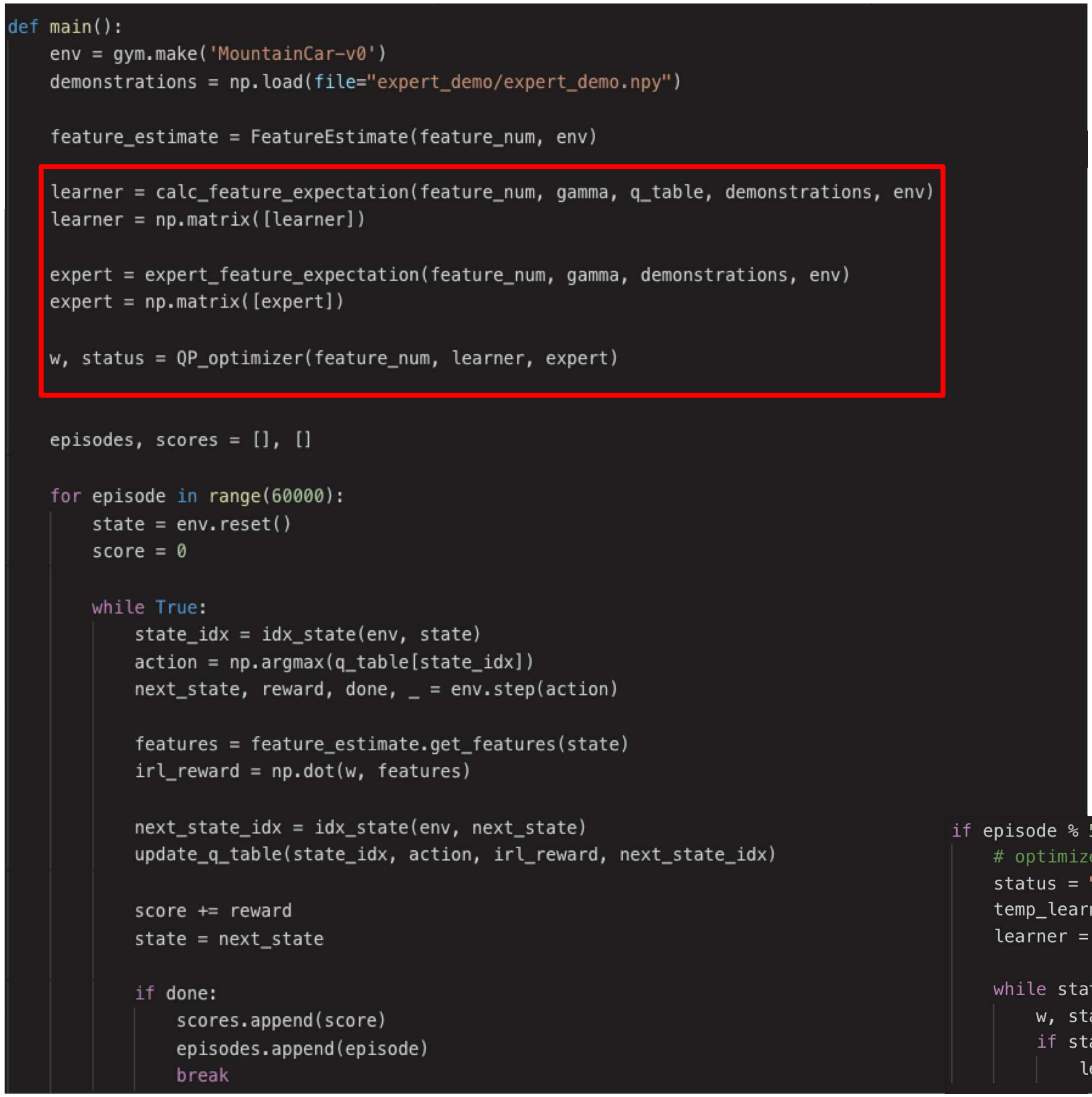
Quadratic loss

$$\min\|w\|_2^2$$

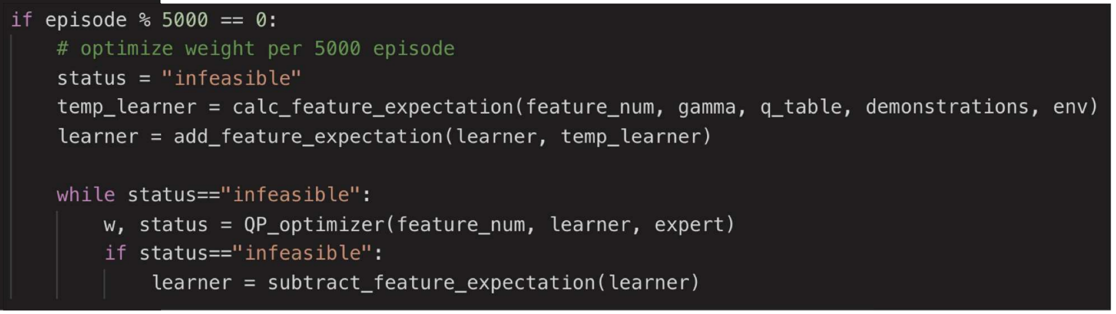
$$s.t\ w^T\mu_E \geq w^T\mu_i + 2$$



```
import cvxpy as cp

def QP_optimizer(feature_num, learner, expert):
    w = cp.Variable(feature_num)

    obj_func = cp.Minimize(cp.norm(w))
    constraints = [(expert-learner) * w >= 2]

    prob = cp.Problem(obj_func, constraints)
    prob.solve()

    if prob.status == "optimal":
        print("status:", prob.status)
        print("optimal value", prob.value)

        weights = np.squeeze(np.asarray(w.value))
        return weights, prob.status
    else:
        print("status:", prob.status)

        weights = np.zeros(feature_num)
        return weights, prob.status
```

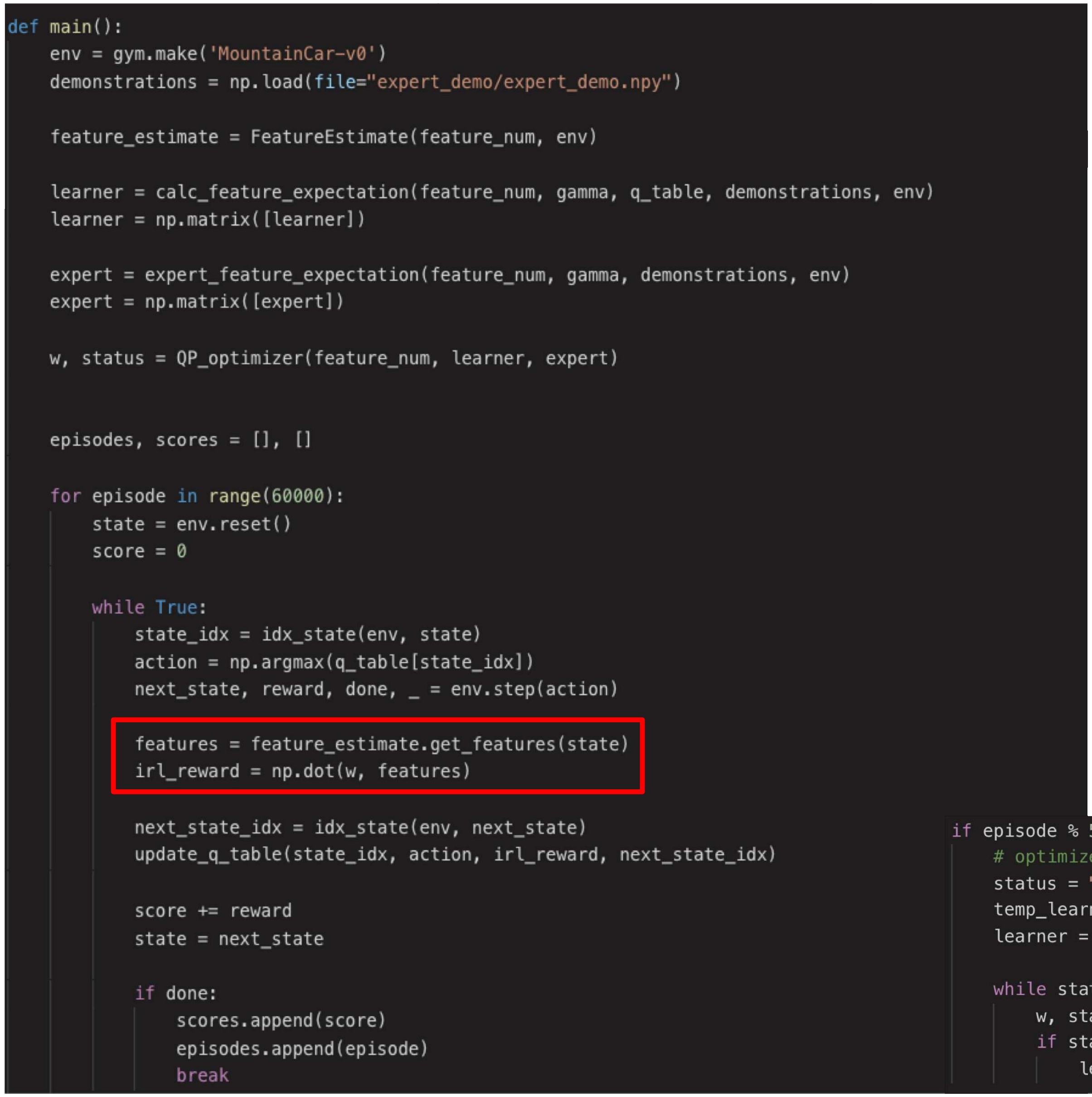


```python
def main():
    env = gym.make('MountainCar-v0')
    demonstrations = np.load(file="expert_demo/expert_demo.npy")

    feature_estimate = FeatureEstimate(feature_num, env)

    learner = calc_feature_expectation(feature_num, gamma, q_table, demonstrations, env)
    learner = np.matrix([learner])

    expert = expert_feature_expectation(feature_num, gamma, demonstrations, env)
    expert = np.matrix([expert])

    w, status = QP_optimizer(feature_num, learner, expert)


    episodes, scores = [], []

    for episode in range(60000):
        state = env.reset()
        score = 0

        while True:
            state_idx = idx_state(env, state)
            action = np.argmax(q_table[state_idx])
            next_state, reward, done, _ = env.step(action)

            features = feature_estimate.get_features(state)
            irl_reward = np.dot(w, features)

            next_state_idx = idx_state(env, next_state)
            update_q_table(state_idx, action, irl_reward, next_state_idx)

            score += reward
            state = next_state

            if done:
                scores.append(score)
                episodes.append(episode)
                break
```

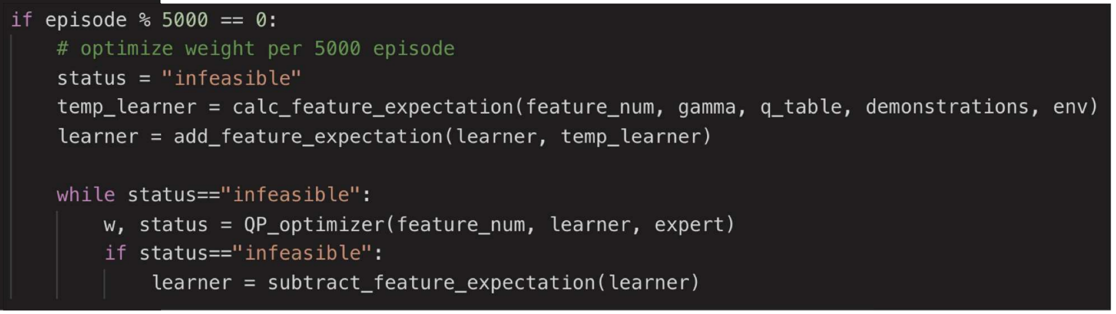


```python
if episode % 5000 == 0:
    # optimize weight per 5000 episode
    status = "infeasible"
    temp_learner = calc_feature_expectation(feature_num, gamma, q_table, demonstrations, env)
    learner = add_feature_expectation(learner, temp_learner)

    while status=="infeasible":
        w, status = QP_optimizer(feature_num, learner, expert)
        if status=="infeasible":
            learner = subtract_feature_expectation(learner)
```

- RL step : Q-learning 사용 (sampling)
  - Model-free Q-learning만 사용 시, 학습 불가능
- IRL step : APP 사용

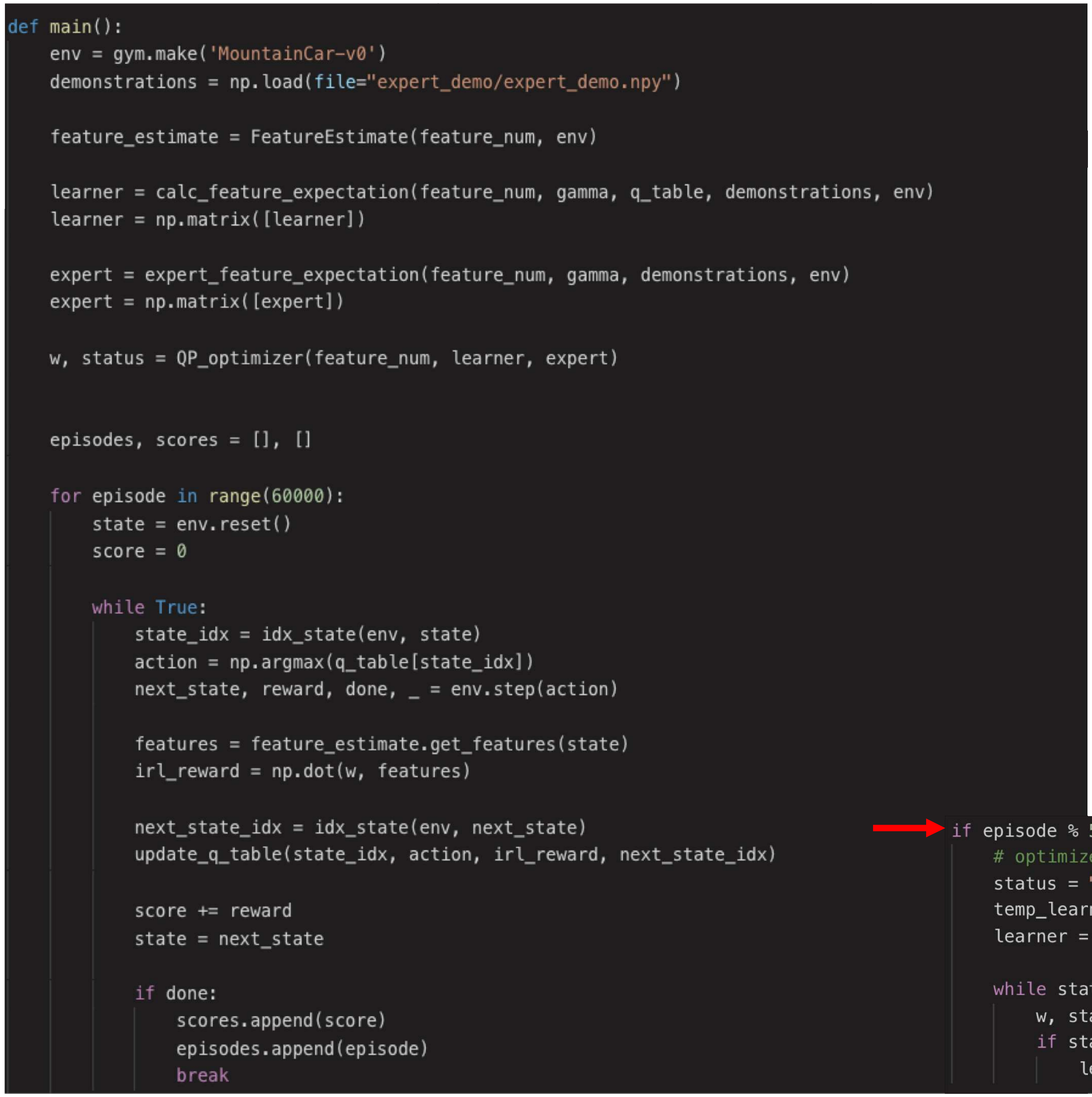


```python
def main():
    env = gym.make('MountainCar-v0')
    demonstrations = np.load(file="expert_demo/expert_demo.npy")

    feature_estimate = FeatureEstimate(feature_num, env)

    learner = calc_feature_expectation(feature_num, gamma, q_table, demonstrations, env)
    learner = np.matrix([learner])

    expert = expert_feature_expectation(feature_num, gamma, demonstrations, env)
    expert = np.matrix([expert])

    w, status = QP_optimizer(feature_num, learner, expert)


    episodes, scores = [], []

    for episode in range(60000):
        state = env.reset()
        score = 0

        while True:
            state_idx = idx_state(env, state)
            action = np.argmax(q_table[state_idx])
            next_state, reward, done, _ = env.step(action)

            features = feature_estimate.get_features(state)
            irl_reward = np.dot(w, features)

            next_state_idx = idx_state(env, next_state)
            update_q_table(state_idx, action, irl_reward, next_state_idx)

            score += reward
            state = next_state

            if done:
                scores.append(score)
                episodes.append(episode)
                break
```

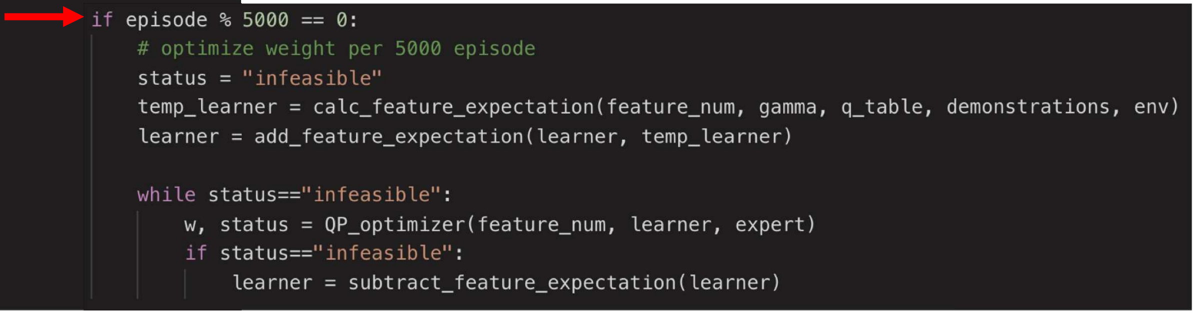


```python
if episode % 5000 == 0:
    # optimize weight per 5000 episode
    status = "infeasible"
    temp_learner = calc_feature_expectation(feature_num, gamma, q_table, demonstrations, env)
    learner = add_feature_expectation(learner, temp_learner)

    while status=="infeasible":
        w, status = QP_optimizer(feature_num, learner, expert)
        if status=="infeasible":
            learner = subtract_feature_expectation(learner)
```

- RL step : Q-learning 사용 (sampling)
    - Model-free Q-learning만 사용 시, 학습 불가능
- IRL step : APP 사용



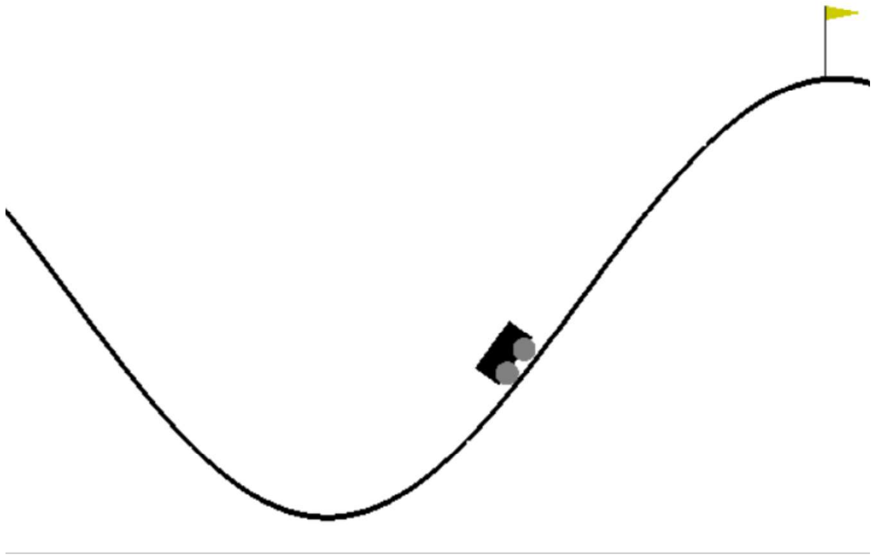
```python
def main():
    env = gym.make('MountainCar-v0')
    demonstrations = np.load(file="expert_demo/expert_demo.npy")

    feature_estimate = FeatureEstimate(feature_num, env)

    learner = calc_feature_expectation(feature_num, gamma, q_table, demonstrations, env)
    learner = np.matrix([learner])

    expert = expert_feature_expectation(feature_num, gamma, demonstrations, env)
    expert = np.matrix([expert])

    w, status = QP_optimizer(feature_num, learner, expert)


    episodes, scores = [], []

    for episode in range(60000):
        state = env.reset()
        score = 0

        while True:
            state_idx = idx_state(env, state)
            action = np.argmax(q_table[state_idx])
            next_state, reward, done, _ = env.step(action)

            features = feature_estimate.get_features(state)
            irl_reward = np.dot(w, features)

            next_state_idx = idx_state(env, next_state)
            update_q_table(state_idx, action, irl_reward, next_state_idx)

            score += reward
            state = next_state

            if done:
                scores.append(score)
                episodes.append(episode)
                break
```



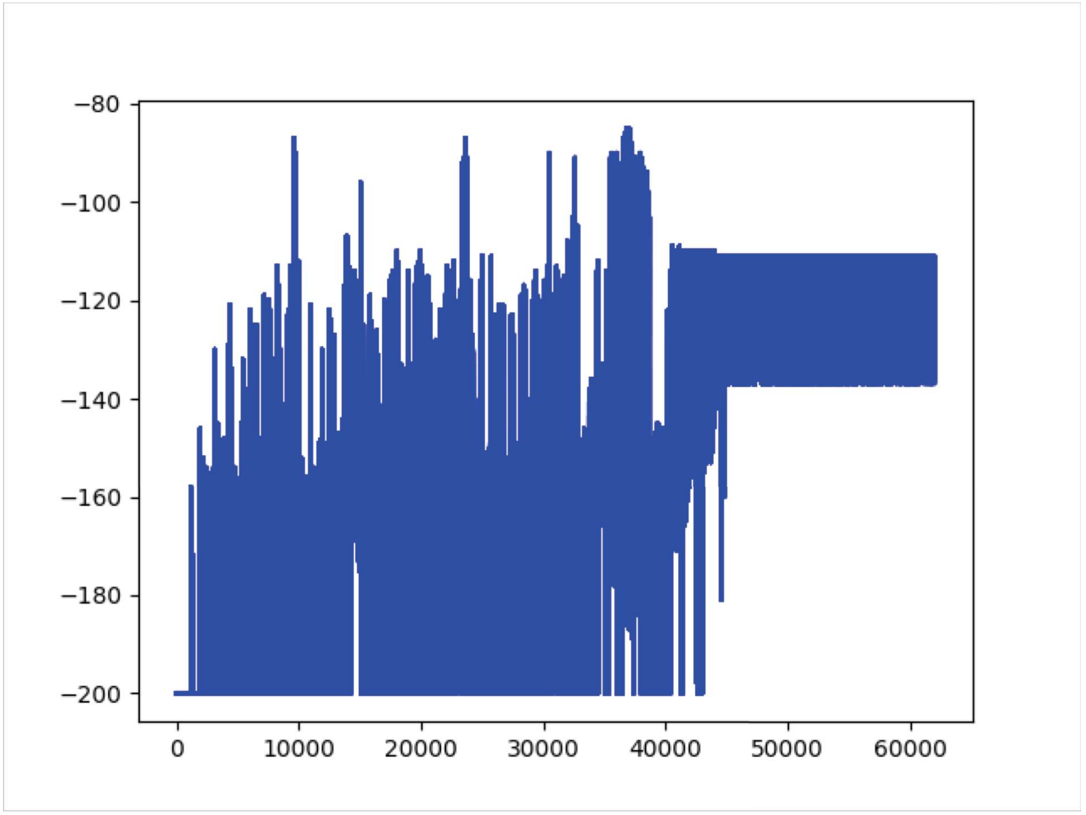
```python
if episode % 5000 == 0:
    # optimize weight per 5000 episode
    status = "infeasible"
    temp_learner = calc_feature_expectation(feature_num, gamma, q_table, demonstrations, env)
    learner = add_feature_expectation(learner, temp_learner)

    while status=="infeasible":
        w, status = QP_optimizer(feature_num, learner, expert)
        if status=="infeasible":
            learner = subtract_feature_expectation(learner)
```

- RL step : Q-learning 사용 (sampling)
  - Model-free Q-learning만 사용 시, 학습 불가능
- IRL step : APP 사용

APP Result



Mountain car rendering



Learning curve

3. MMP

Maximum Margin Planning (2006)

References – MMP

- Maximum Margin Planning (Paper Link)
- Cool Inverse Reinforcement Learning (IRL) Papers - IRL survey (최성준 박사님) (PDF Link)
- SVM | Lecture 1 Decision boundary with Margin (문일철 교수님) (YouTube Link)
- SVM | Lecture 2 Maximizing the Margin (문일철 교수님) (YouTube Link)
- SVM | Lecture 3 SVM with Matlab (문일철 교수님) (YouTube Link)
- SVM | Lecture 4 Error Handling in SVM (문일철 교수님) (YouTube Link)
- SVM | Lecture 5 Soft-Margin SVM (문일철 교수님) (YouTube Link)
- SVM (Support Vector Machine) Part 1 (Blog Link)
- SVM (Support Vector Machine) Part 2 (Blog Link)
- 모두를 위한 컨벡스 최적화 – Subgradient (Wikidocs Link)
- 모두를 위한 컨벡스 최적화 – Subgradient Method (Wikidocs Link)

Summary - MMP

- Learn behaviors as a maximum margin structured prediction problem
- Learn mappings from features to cost
- In robotics view, automate the mapping from perception to costs
  -> Perception + Planning
- What is the state-action frequency counts?
- Proposed Algorithm : Maximum Margin Planning (RL step X)
- Proposed Idea : Soft margin SVM, Subgradient method

Preliminaries

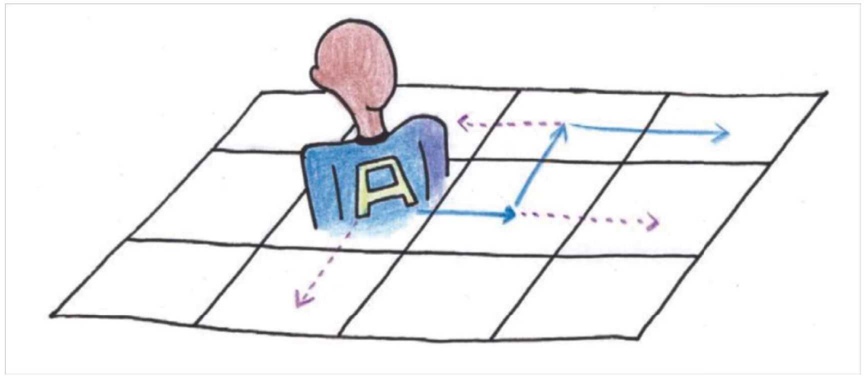
- Modeling the planning problem with discrete MDPs
  - $\mathcal{X}$: state spaces
  - $\mathcal{A}$: action spaces
  - $p(y|s,a)$: transition probabilities
  - s : initial-state distribution
  - γ : discount factor
  - R : learned from supervised examples to produce policies that mimic demonstrated behavior
  - $v \in \mathcal{V}$: primal variables of value function
  - $\mu \in \mathcal{G}$: dual state-action visitation frequency counts (equal to y)

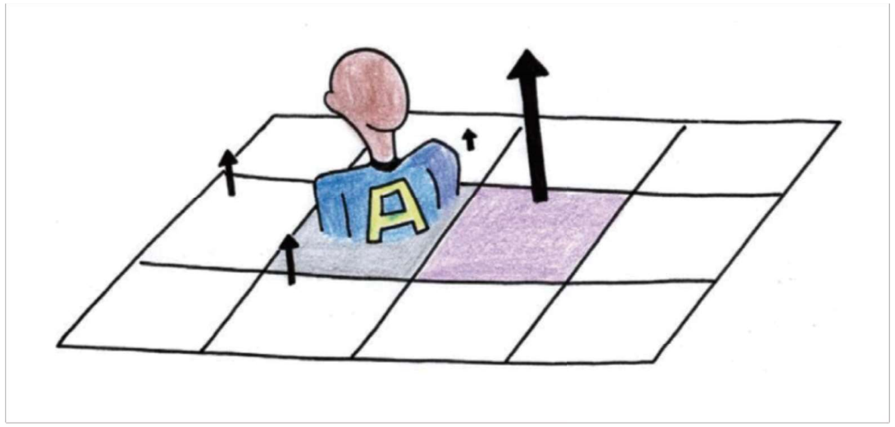
State-action visitation frequency counts

- IRL의 궁극적인 목표를 다르게 말해보면, expert가 어떠한 행동을 했을 때 여기서의 state-action visitation frequency를 구하고 expert와 최대한 비슷한 visitation frequency를 만들어내는 reward를 찾는 것

- 또한 RL의 problem은 reward가 주어졌을 때 이 reward의 expected sum을 최대로 하는 policy를 찾는 것
  그런데 RL의 dual problem은 visitation frequency를 찾는 것이라고도 말할 수 있다.

- 다시 말해 optimal policy와 optimal visitation frequency는 1:1관계





source : 파이썬과 케라스로 배우는 강화학습 저자특강
image source : https://www.slideshare.net/WoongwonLee/ss-78783597

Preliminaries

- The input to our Algorithm : $\mathcal{D} = \{(\mathcal{X}_i, \mathcal{A}_i, p_i, F_i, y_i, \mathcal{L}_i)\}_{i=1}^n$
  - F_i : feature matrix (or mapping)
  - y_i : expert's demonstration.
  - $\mathcal{L}_i$: some additional loss function (heuristic)
- $\mathcal{D} = \{(\mathcal{X}_i, \mathcal{A}_i, p_i, f_i, y_i, \mathcal{L}_i)\}_{i=1}^n \equiv \{(\mathcal{X}_i, \mathcal{A}_i, \mathcal{G}_i, F_i, \mu_i, l_i)\}_{i=1}^n$
- $\mu_i^{x,a}$: expert's state-action visitation frequency for state x and action a of example i
- $\mathcal{L}(y_i, y) = \mathcal{L}_i(y) = l_i^T \mu$
  - $\mathcal{L}(y_i, y)$: number of states where y_i and y do not match
  - l_i = 1 - expert's visitation frequency

The goal of this paper

The best policy over the resulting reward function

$\mu^* = arg\max_{\mu \in \mathcal{G}} w^T F_i \, \mu$ is "close" to the expert's demonstrated policy μ_i

Proposed Algorithm

- **Quadratic Program**
  - Given a training set $\mathcal{D} = \{(\mathcal{X}_i, \mathcal{A}_i, p_i, f_i, y_i, \mathcal{L}_i)\}_{i=1}^{n}$,
  - Quadratic Program is

$$\min_{w,\zeta_i} \frac{1}{2} \parallel w \parallel^2 + \frac{\gamma}{n} \sum_i \beta_i \zeta_i$$

$$s.\,t.\ \ \forall i \quad w^T f_i(y_i) \geq \max_{\mu \in \mathcal{G}_i} (w^T f_i(y) + \mathcal{L}(y_i, y)) - \zeta_i$$

    - $\frac{r}{n} \sum_i \beta_i \zeta_i$: soft margin term
    - ζ_i : slack variable
    - r : scaling for a penalty
    - β_i : data dependent scalars that can be used for normalization when the examples are of different lengths
    - $w^T f_i(y_i)$: expert's reward / $w^T f_i(y)$: other's reward

Proposed Algorithm

- Quadratic Program

$$\min_{w,\zeta_i} \frac{1}{2} \parallel w \parallel^2 + \frac{\gamma}{n} \sum_i \beta_i \zeta_i$$

$$s.t. \ \ \forall i \quad w^T f_i(y_i) \geq \max_{\mu \in \mathcal{G}_i} (w^T f_i(y) + \mathcal{L}(y_i, y)) - \zeta_i$$

$$\Big\downarrow$$

- Maximum Margin structured prediction problem

$$\min_{w,\zeta_i} \frac{1}{2} \parallel w \parallel^2 + \frac{\gamma}{n} \sum_i \beta_i \zeta_i$$

$$s.t. \ \ \forall i \quad w^T F_i \mu_i \geq \max_{\mu \in \mathcal{G}_i} (w^T F_i \mu + l_i^T \mu) - \zeta_i$$

Maximum Margin Planning

Inverse RL 3번째 논문

#프로젝트 #GAIL하자!

프로젝트 >

Maximum Margin Planning

Nathan D. Ratliff NDR@RI.CMU.EDU
J. Andrew Bagnell DBAGNELL@RI.CMU.EDU
Robotics Institute, Carnegie Mellon University, Pittsburgh, PA. 15213 USA
Martin A. Zinkevich MAZ@CS.UALBERTA.CA
Department of Computing Science, University of Alberta, Edmonton, AB T6G 2E1, Canada

Author : Nathan D. Ratliff, J. Andrew Bagnell, Martin A. Zinkevich

Paper Link : https://www.ri.cmu.edu/pub_files/pub4/ratliff_nathan_2006_1/ratliff_nathan_2006_1.pdf

Proceeding : International Conference on Machine Learning (ICML) 2006

MMP 정리

알고리즘에 대한 추가적인 내용은 블로그를 참고해주세요!

4. MaxEnt

Maximum Entropy Inverse Reinforcement Learning (2008)

References – MaxEnt

- Maximum Entropy Inverse Reinforcement Learning (Paper Link)
- Modeling Purposeful Adaptive Behavior with the Principle of Maximum Causal Entropy (Paper Link)
- Cool Inverse Reinforcement Learning (IRL) Papers - IRL survey (최성준 박사님) (PDF Link)
- 정보이론 기초 (Blog Link)
- 최대엔트로피모델(Maximum Entropy Models) (Blog Link)
- Introduction of Inverse Reinforcement Learning (곽동현님) (PPT Link)
- Revisit Maximum Entropy Inverse Reinforcement Learning (Blog Link)
- Inverse-Reinforcement-Learning (GitHub Link)

Summary - MaxEnt

- Principle of maximum entropy
- Maximum Likelihood Estimation
- Path-based distribution
- Gradient ascent method
- Proposed Algorithm : Expected Edge Frequency Calculation
- Proposed Idea : Gradient ascent using principle of maximum entropy

Preliminaries

- **Imitation Learning setting**
  - Learner's behavior
    - ζ : trajectory or path
    - s_i : states
    - a_i : actions
  - $f_{s_j} \in \Re^k$: mapping features of each state
  - θ : reward weights parameterizing reward function

Preliminaries

- Sum of the state features along the trajectory (or path) = Learner's expected feature counts

$$\mathbf{f}_\zeta = \sum_{s_j \in \zeta} \mathbf{f}_{s_j}$$

- Reward function

$$\text{reward}(\mathbf{f}_\zeta) = \theta^\top \mathbf{f}_\zeta = \sum_{s_j \in \zeta} \theta^\top \mathbf{f}_{s_j}$$

- Expert's expected feature counts based on many (m) demonstrated trajectories

$$\tilde{\mathbf{f}} = \frac{1}{m} \sum_i \mathbf{f}_{\tilde{\zeta}_i}$$

Principle of maximum entropy

- Maximizing the entropy of the distribution over paths subject to the feature constraints from observed data implies that we maximize the likelihood of the observed data under the maximum entropy distribution (Jaynes 1957)

- The principle of maximum entropy states that the probability distribution which best represents the current state of knowledge is the one with largest entropy, in the context of precisely stated prior data (Wikipedia, CMU 10703 Fall 2018)

- The maximum entropy distribution minimizes the worst case prediction log-loss. The maximum entropy distribution is the one where the decision maker chooses P to minimize the worst log-loss that nature can possible create by choosing $\tilde{P}$ adversarially (Theorem 5.2, thesis ziebart 2010)

$$\inf_{\mathrm{P}(X)} \sup_{\tilde{\mathrm{P}}(X)} -\sum_{X} \tilde{\mathrm{P}}(X) \log \mathrm{P}(X)$$

  - Given e.g., feature expectation constraints that P and $\tilde{P}$ both match: $E_{\tilde{P}(X)}[F(X)]$.

Proposed Algorithm

- In this paper, **the principle of maximum entropy**

$$P(\zeta_i|\theta) = \frac{1}{Z(\theta)} e^{\theta^\top \mathbf{f}_{\zeta_i}} = \frac{1}{Z(\theta)} e^{\sum_{s_j \in \zeta_i} \theta^\top \mathbf{f}_{s_j}}$$

  - $Z(\theta) = \sum_{\zeta_i} e^{\theta^T f_{\zeta_i}}$: partition function

- Learning from Demonstrated Behavior (Maximum Likelihood Estimation)

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, L(\theta) = \underset{\theta}{\operatorname{argmax}} \sum_{\text{examples}} \log \underline{P(\tilde{\zeta}|\theta)}$$

- Gradient for difference between expert's expected feature counts and learner's expected feature counts

$$\nabla L(\theta) = \tilde{f} - \sum_{\zeta} P(\zeta|\theta) f_{\zeta} = \tilde{f} - \sum_{s_i} D_{s_i} f_{s_i}$$

Proposed Algorithm

- In this paper, **the principle of maximum entropy**

$$P(\zeta_i|\theta) = \frac{1}{Z(\theta)} e^{\theta^\top \mathbf{f}_{\zeta_i}} = \frac{1}{Z(\theta)} e^{\sum_{s_j \in \zeta_i} \theta^\top \mathbf{f}_{s_j}}$$

  - $Z(\theta) = \sum_{\zeta_i} e^{\theta^T f_{\zeta_i}}$: partition function

- Learning from Demonstrated Behavior (Maximum Likelihood Estimation)

$$\theta^* = \underset{\theta}{\operatorname{argmax}} L(\theta) = \underset{\theta}{\operatorname{argmax}} \sum_{\text{examples}} \log P(\tilde{\zeta}|\theta)$$

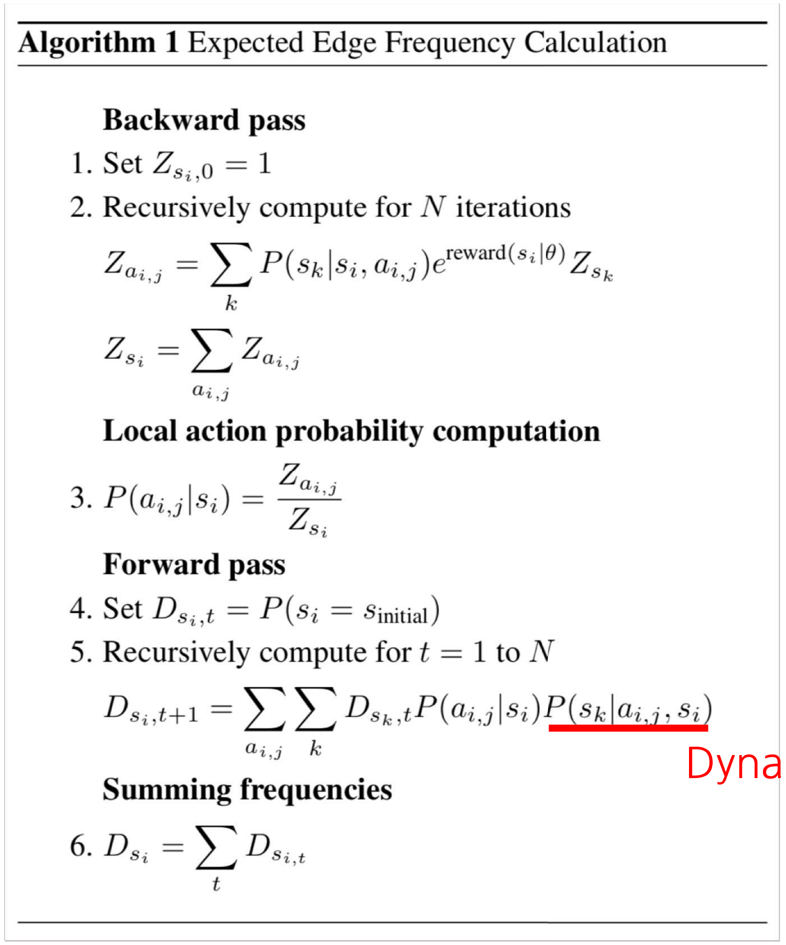
- Gradient for difference between expert's expected feature counts and learner's expected feature counts

$$\nabla L(\theta) = \tilde{f} - \sum_{\zeta} P(\zeta|\theta) f_\zeta = \tilde{f} - \sum_{s_i} \underline{D_{s_i}} f_{s_i}$$

state visitation frequency!

Proposed Algorithm

- Efficient State Visitation Frequency Calculations



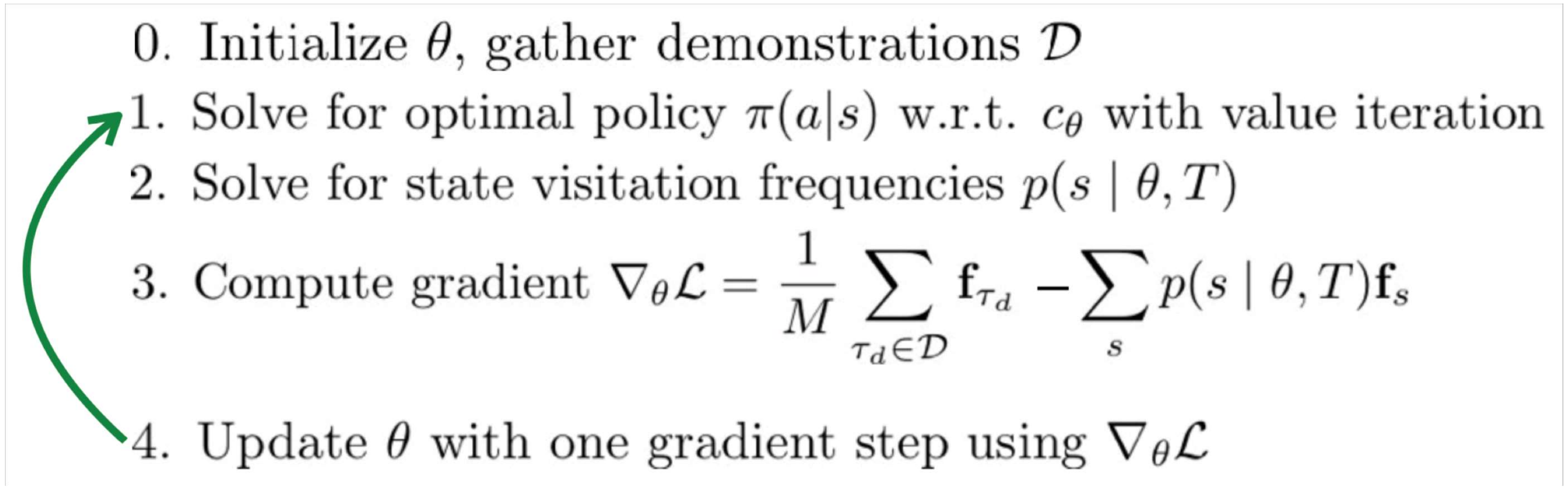
Algorithm 1 Expected Edge Frequency Calculation

```
    Backward pass
1. Set $Z_{s_i,0} = 1$
2. Recursively compute for $N$ iterations
    $Z_{a_{i,j}} = \sum_k P(s_k|s_i, a_{i,j}) e^{\text{reward}(s_i|\theta)} Z_{s_k}$
    $Z_{s_i} = \sum_{a_{i,j}} Z_{a_{i,j}}$
    Local action probability computation
3. $P(a_{i,j}|s_i) = \frac{Z_{a_{i,j}}}{Z_{s_i}}$
    Forward pass
4. Set $D_{s_i,t} = P(s_i = s_{\text{initial}})$
5. Recursively compute for $t = 1$ to $N$
    $D_{s_i,t+1} = \sum_{a_{i,j}} \sum_k D_{s_k,t} P(a_{i,j}|s_i) \underline{P(s_k|a_{i,j}, s_i)}$
    Summing frequencies
6. $D_{s_i} = \sum_t D_{s_i,t}$
```

Dynamics are known!

Proposed Algorithm

- Known dynamics, linear costs

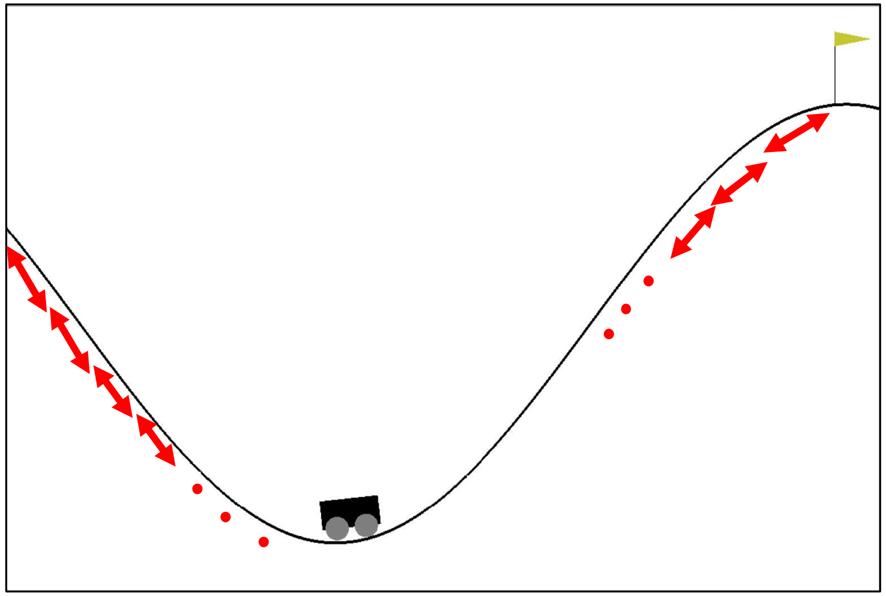


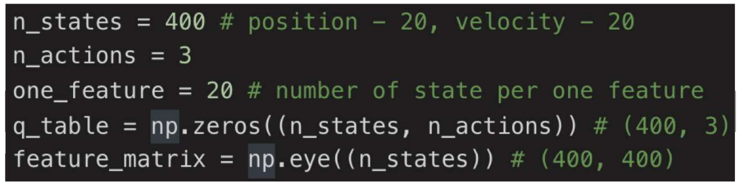
0. Initialize θ, gather demonstrations $\mathcal{D}$
1. Solve for optimal policy $\pi(a|s)$ w.r.t. c_θ with value iteration
2. Solve for state visitation frequencies $p(s \mid \theta, T)$
3. Compute gradient $\nabla_\theta \mathcal{L} = \frac{1}{M} \sum_{\tau_d \in \mathcal{D}} \mathbf{f}_{\tau_d} - \sum_s p(s \mid \theta, T)\mathbf{f}_s$
4. Update θ with one gradient step using $\nabla_\theta \mathcal{L}$

source : Maximum Entropy Inverse RL, Adversarial imitation learning
image source : http://www.andrew.cmu.edu/course/10-703/slides/Lecture_IRL_GAIL.pdf

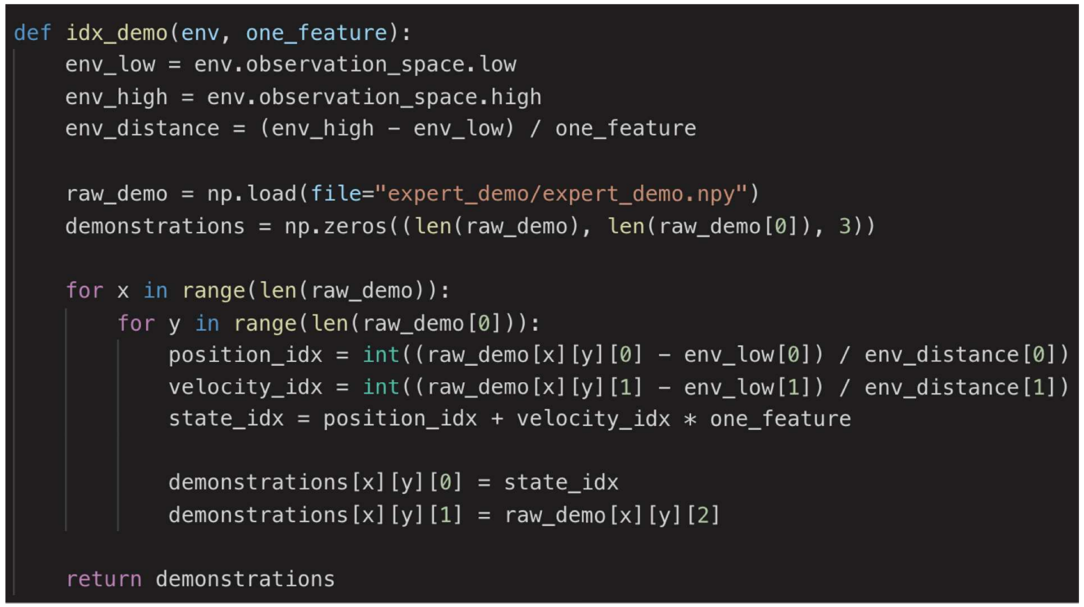
MaxEnt의 구현이야기

- Demonstraions of 동민's expert
  - (20, 130, 3)
  - (demo_num, demo_length, state + action)





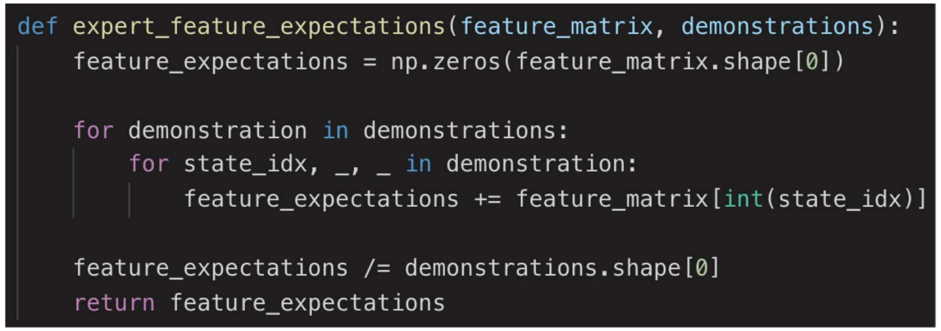
```
n_states = 400 # position - 20, velocity - 20
n_actions = 3
one_feature = 20 # number of state per one feature
q_table = np.zeros((n_states, n_actions)) # (400, 3)
feature_matrix = np.eye((n_states)) # (400, 400)
```



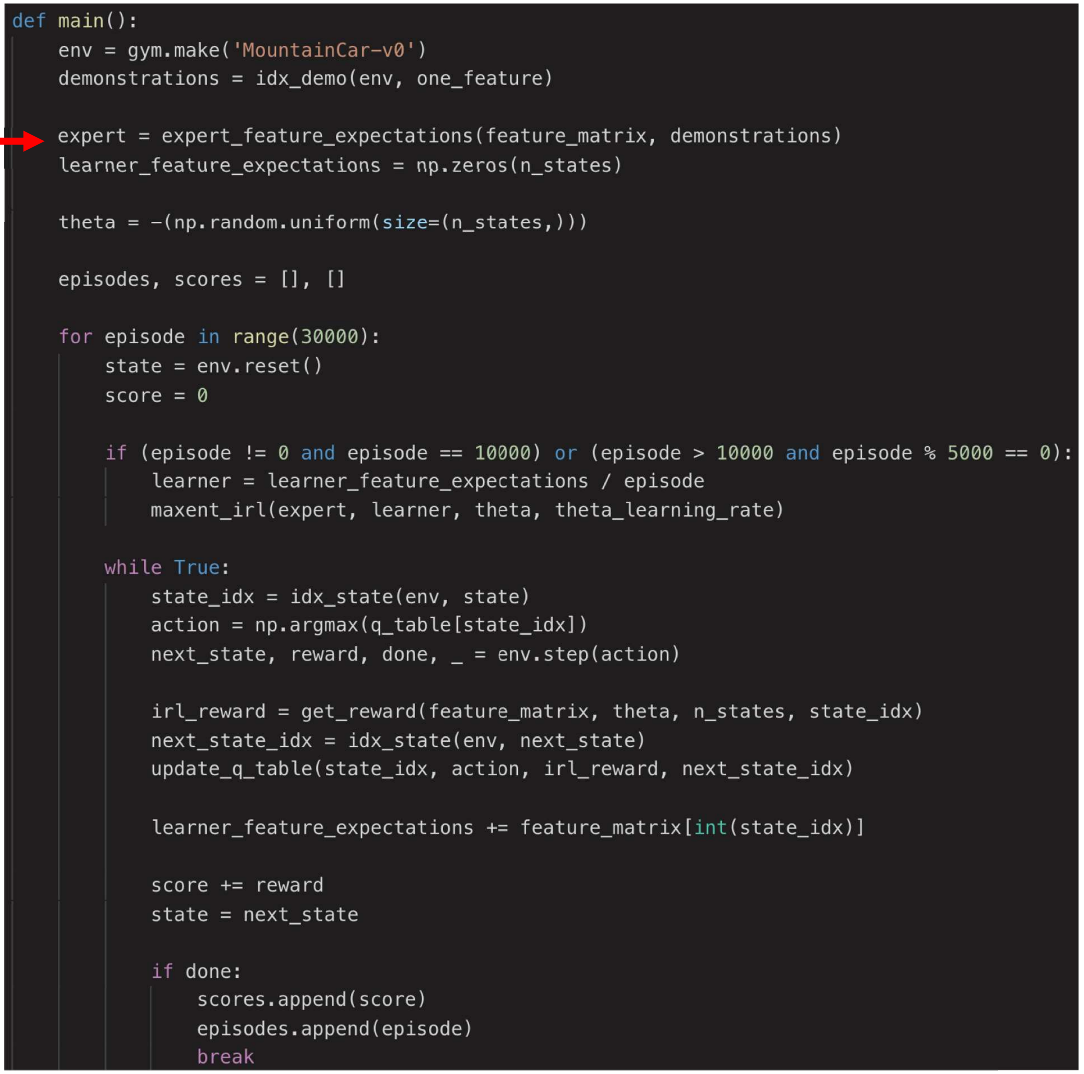
```
def idx_demo(env, one_feature):
    env_low = env.observation_space.low
    env_high = env.observation_space.high
    env_distance = (env_high - env_low) / one_feature

    raw_demo = np.load(file="expert_demo/expert_demo.npy")
    demonstrations = np.zeros((len(raw_demo), len(raw_demo[0]), 3))

    for x in range(len(raw_demo)):
        for y in range(len(raw_demo[0])):
            position_idx = int((raw_demo[x][y][0] - env_low[0]) / env_distance[0])
            velocity_idx = int((raw_demo[x][y][1] - env_low[1]) / env_distance[1])
            state_idx = position_idx + velocity_idx * one_feature

            demonstrations[x][y][0] = state_idx
            demonstrations[x][y][1] = raw_demo[x][y][2]

    return demonstrations
```

- Expert's expected feature counts

$$\tilde{\mathbf{f}} = \frac{1}{m}\sum_i \mathbf{f}_{\tilde{\zeta}_i}$$



```python
def expert_feature_expectations(feature_matrix, demonstrations):
    feature_expectations = np.zeros(feature_matrix.shape[0])

    for demonstration in demonstrations:
        for state_idx, _, _ in demonstration:
            feature_expectations += feature_matrix[int(state_idx)]

    feature_expectations /= demonstrations.shape[0]
    return feature_expectations
```



```python
def main():
    env = gym.make('MountainCar-v0')
    demonstrations = idx_demo(env, one_feature)

    expert = expert_feature_expectations(feature_matrix, demonstrations)
    learner_feature_expectations = np.zeros(n_states)

    theta = -(np.random.uniform(size=(n_states,)))

    episodes, scores = [], []

    for episode in range(30000):
        state = env.reset()
        score = 0

        if (episode != 0 and episode == 10000) or (episode > 10000 and episode % 5000 == 0):
            learner = learner_feature_expectations / episode
            maxent_irl(expert, learner, theta, theta_learning_rate)

        while True:
            state_idx = idx_state(env, state)
            action = np.argmax(q_table[state_idx])
            next_state, reward, done, _ = env.step(action)

            irl_reward = get_reward(feature_matrix, theta, n_states, state_idx)
            next_state_idx = idx_state(env, next_state)
            update_q_table(state_idx, action, irl_reward, next_state_idx)

            learner_feature_expectations += feature_matrix[int(state_idx)]

            score += reward
            state = next_state

            if done:
                scores.append(score)
                episodes.append(episode)
                break
```

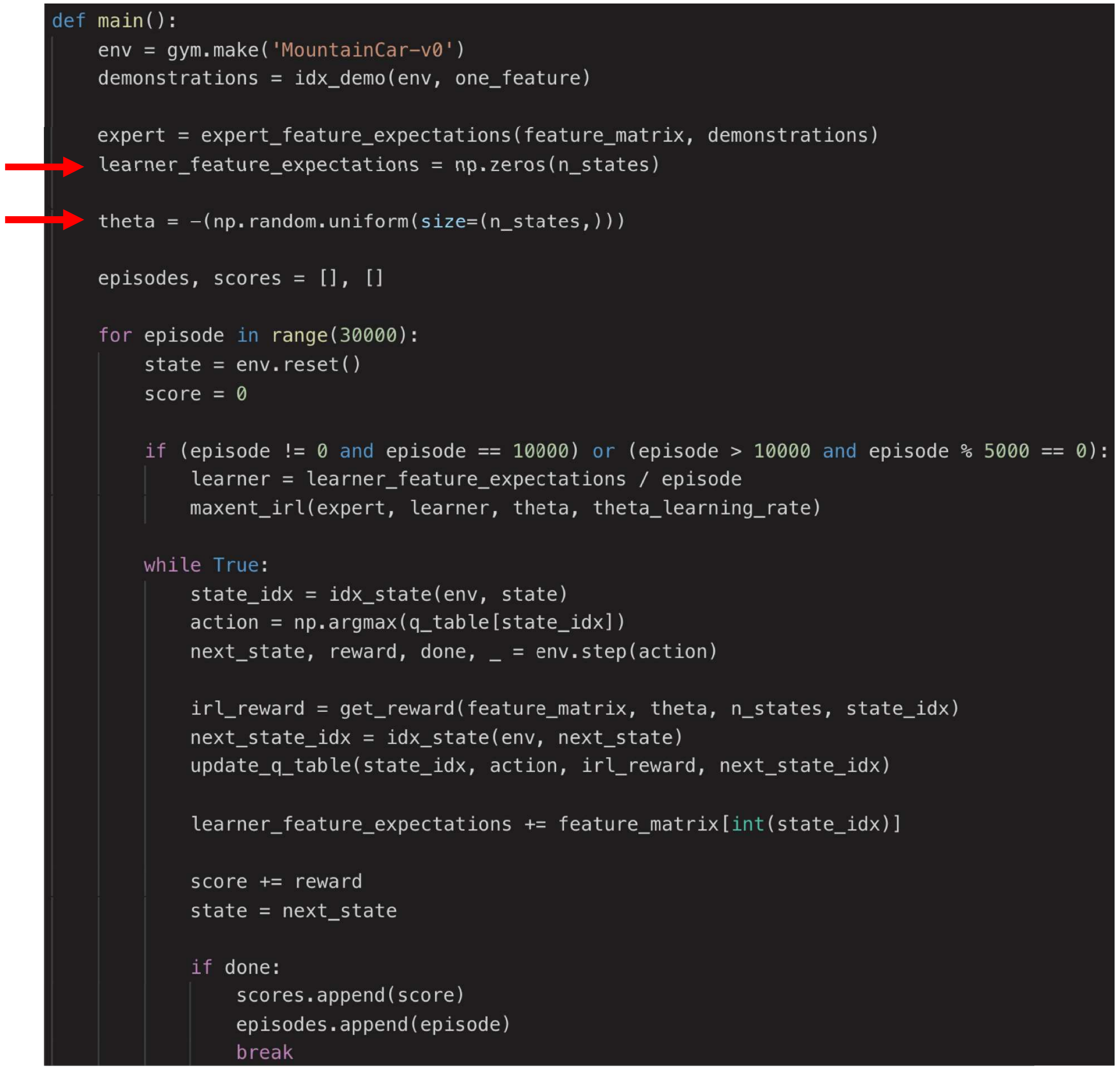
- Expert's expected feature counts

$$\tilde{\mathbf{f}} = \frac{1}{m}\sum_i \mathbf{f}_{\tilde{\zeta}_i}$$

- Learner's expected feature counts

$$\mathbf{f}_{\zeta} = \sum_{s_j \in \zeta} \mathbf{f}_{s_j}$$

- θ : reward weights



```
def main():
    env = gym.make('MountainCar-v0')
    demonstrations = idx_demo(env, one_feature)

    expert = expert_feature_expectations(feature_matrix, demonstrations)
    learner_feature_expectations = np.zeros(n_states)

    theta = -(np.random.uniform(size=(n_states,)))

    episodes, scores = [], []

    for episode in range(30000):
        state = env.reset()
        score = 0

        if (episode != 0 and episode == 10000) or (episode > 10000 and episode % 5000 == 0):
            learner = learner_feature_expectations / episode
            maxent_irl(expert, learner, theta, theta_learning_rate)

        while True:
            state_idx = idx_state(env, state)
            action = np.argmax(q_table[state_idx])
            next_state, reward, done, _ = env.step(action)

            irl_reward = get_reward(feature_matrix, theta, n_states, state_idx)
            next_state_idx = idx_state(env, next_state)
            update_q_table(state_idx, action, irl_reward, next_state_idx)

            learner_feature_expectations += feature_matrix[int(state_idx)]

            score += reward
            state = next_state

            if done:
                scores.append(score)
                episodes.append(episode)
                break
```

- Expert’s expected feature counts

$$\tilde{\mathbf{f}} = \frac{1}{m}\sum_i \mathbf{f}_{\tilde{\zeta}_i}$$

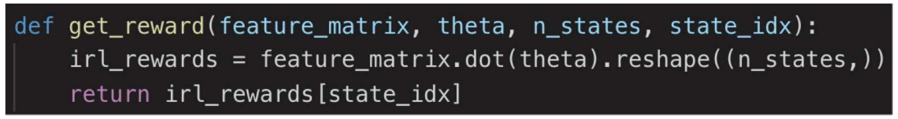
- Learner’s expected feature counts

$$\mathbf{f}_\zeta = \sum_{s_j \in \zeta} \mathbf{f}_{s_j}$$

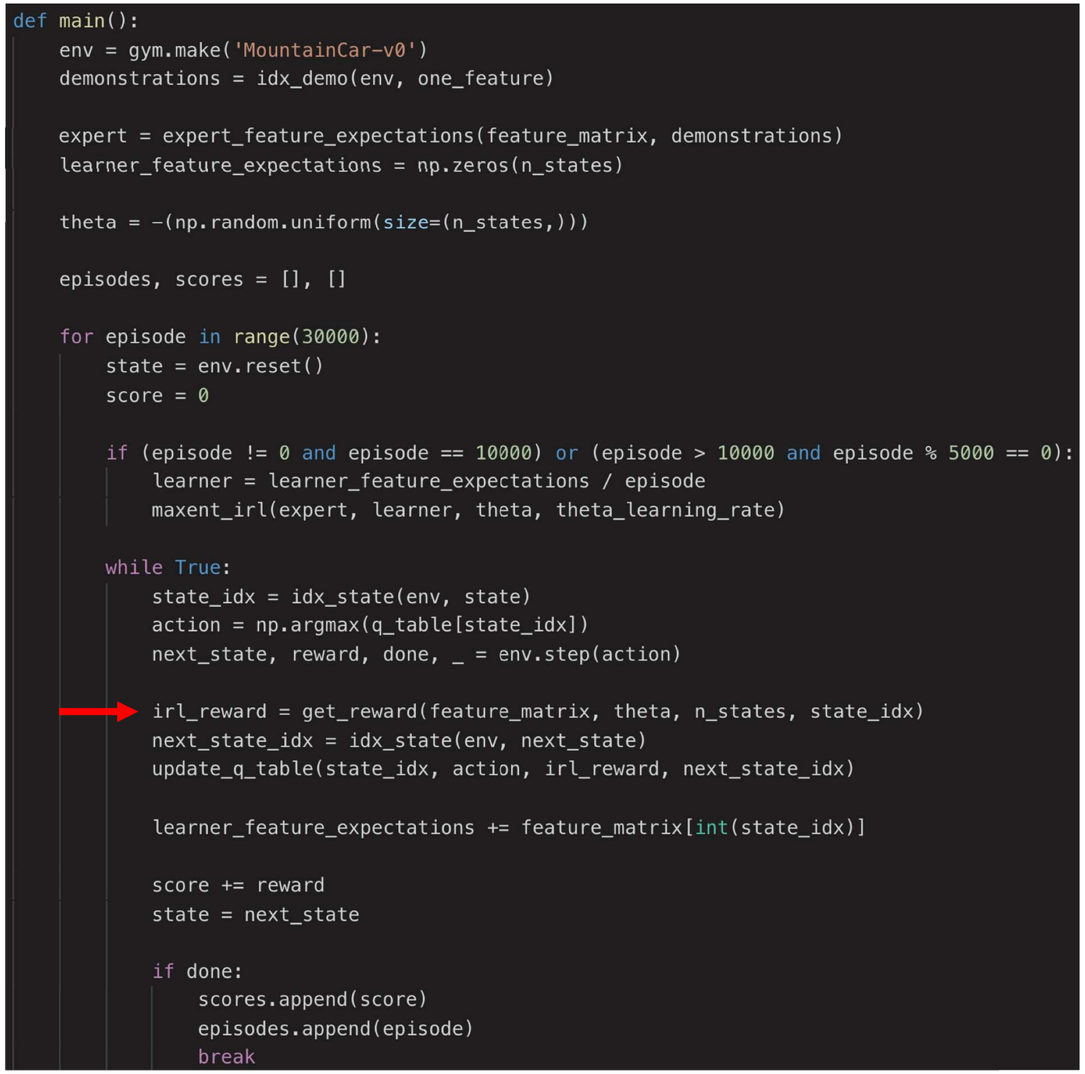
- θ : reward weights

- Reward function

$$\text{reward}(\mathbf{f}_\zeta) = \theta^\top \mathbf{f}_\zeta = \sum_{s_j \in \zeta} \theta^\top \mathbf{f}_{s_j}$$



```python
def get_reward(feature_matrix, theta, n_states, state_idx):
    irl_rewards = feature_matrix.dot(theta).reshape((n_states,))
    return irl_rewards[state_idx]
```



```python
def main():
    env = gym.make('MountainCar-v0')
    demonstrations = idx_demo(env, one_feature)

    expert = expert_feature_expectations(feature_matrix, demonstrations)
    learner_feature_expectations = np.zeros(n_states)

    theta = -(np.random.uniform(size=(n_states,)))

    episodes, scores = [], []

    for episode in range(30000):
        state = env.reset()
        score = 0

        if (episode != 0 and episode == 10000) or (episode > 10000 and episode % 5000 == 0):
            learner = learner_feature_expectations / episode
            maxent_irl(expert, learner, theta, theta_learning_rate)

        while True:
            state_idx = idx_state(env, state)
            action = np.argmax(q_table[state_idx])
            next_state, reward, done, _ = env.step(action)

            irl_reward = get_reward(feature_matrix, theta, n_states, state_idx)
            next_state_idx = idx_state(env, next_state)
            update_q_table(state_idx, action, irl_reward, next_state_idx)

            learner_feature_expectations += feature_matrix[int(state_idx)]

            score += reward
            state = next_state

            if done:
                scores.append(score)
                episodes.append(episode)
                break
```

- Expert's expected feature counts

$$\tilde{\mathbf{f}} = \frac{1}{m}\sum_i \mathbf{f}_{\tilde{\zeta}_i}$$

- Learner's expected feature counts

$$\mathbf{f}_\zeta = \sum_{s_j \in \zeta} \mathbf{f}_{s_j}$$

- θ : reward weights
- Reward function

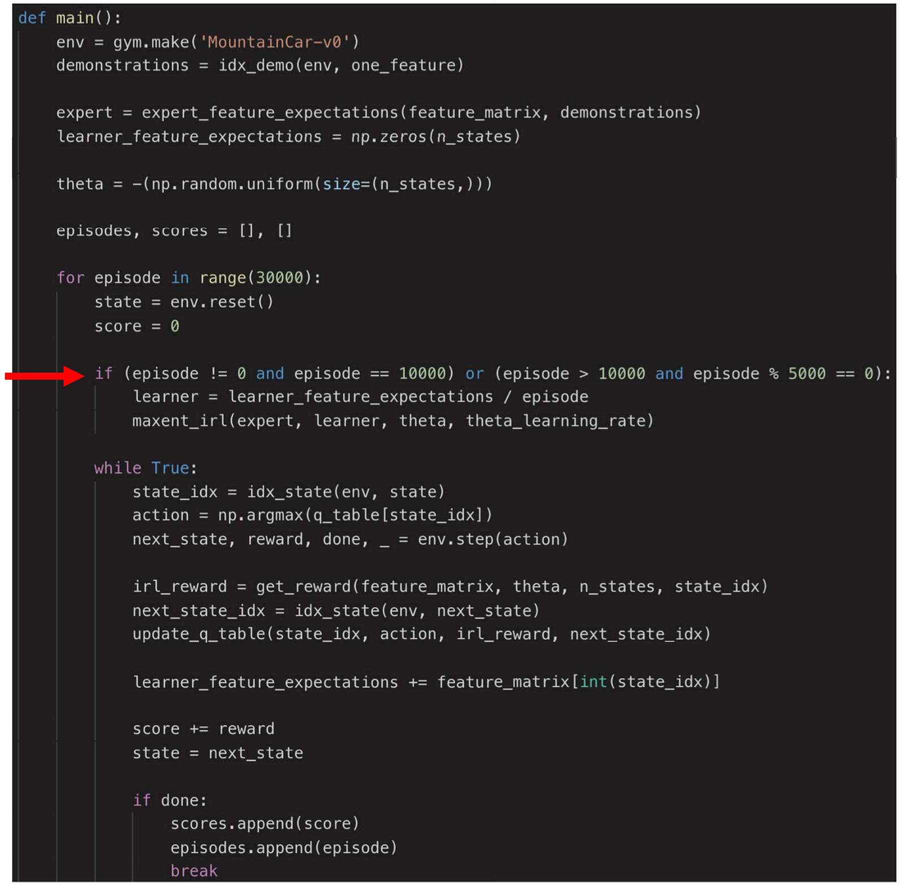
$$\text{reward}(\mathbf{f}_\zeta) = \theta^\top \mathbf{f}_\zeta = \sum_{s_j \in \zeta} \theta^\top \mathbf{f}_{s_j}$$

- Gradient descent on theta

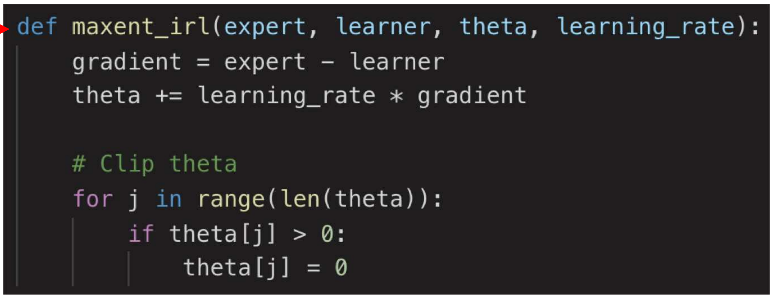
$$\nabla L(\theta) = \tilde{\mathbf{f}} - \sum_{\zeta} P(\zeta|\theta, T)\mathbf{f}_\zeta = \tilde{\mathbf{f}} - \sum_{s_i} D_{s_i}\mathbf{f}_{s_i}$$

$$= \tilde{f} - f_\zeta$$

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, L(\theta) = \underset{\theta}{\operatorname{argmax}} \sum_{\text{examples}} \log P(\tilde{\zeta}|\theta, T)$$



```python
def main():
    env = gym.make('MountainCar-v0')
    demonstrations = idx_demo(env, one_feature)

    expert = expert_feature_expectations(feature_matrix, demonstrations)
    learner_feature_expectations = np.zeros(n_states)

    theta = -(np.random.uniform(size=(n_states,)))

    episodes, scores = [], []

    for episode in range(30000):
        state = env.reset()
        score = 0

        if (episode != 0 and episode == 10000) or (episode > 10000 and episode % 5000 == 0):
            learner = learner_feature_expectations / episode
            maxent_irl(expert, learner, theta, theta_learning_rate)

        while True:
            state_idx = idx_state(env, state)
            action = np.argmax(q_table[state_idx])
            next_state, reward, done, _ = env.step(action)

            irl_reward = get_reward(feature_matrix, theta, n_states, state_idx)
            next_state_idx = idx_state(env, next_state)
            update_q_table(state_idx, action, irl_reward, next_state_idx)

            learner_feature_expectations += feature_matrix[int(state_idx)]

            score += reward
            state = next_state

            if done:
                scores.append(score)
                episodes.append(episode)
                break
```



```python
def maxent_irl(expert, learner, theta, learning_rate):
    gradient = expert - learner
    theta += learning_rate * gradient

    # Clip theta
    for j in range(len(theta)):
        if theta[j] > 0:
            theta[j] = 0
```

- Expert's expected feature counts

$$\tilde{\mathbf{f}} = \frac{1}{m}\sum_i \mathbf{f}_{\tilde{\zeta}_i}$$

- Learner's expected feature counts

$$\mathbf{f}_\zeta = \sum_{s_j \in \zeta} \mathbf{f}_{s_j}$$

- θ : reward weights
- Reward function

$$\text{reward}(\mathbf{f}_\zeta) = \theta^\top \mathbf{f}_\zeta = \sum_{s_j \in \zeta} \theta^\top \mathbf{f}_{s_j}$$

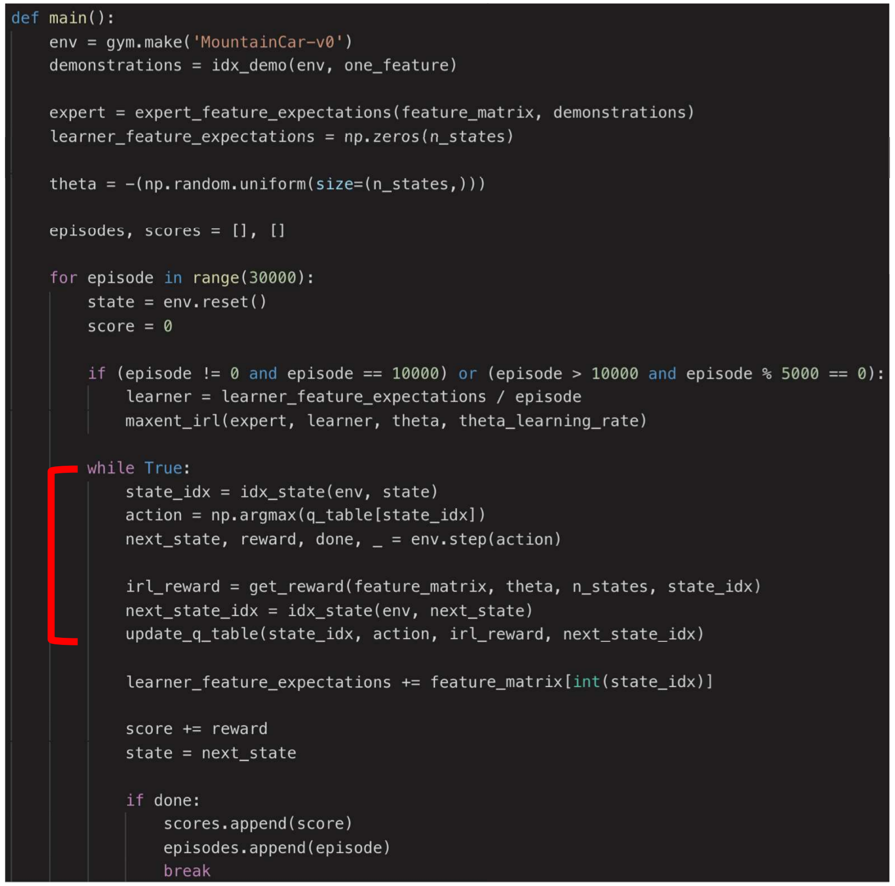
- Gradient descent on theta

$$\nabla L(\theta) = \tilde{\mathbf{f}} - \sum_\zeta P(\zeta|\theta, T)\mathbf{f}_\zeta = \tilde{\mathbf{f}} - \sum_{s_i} D_{s_i}\mathbf{f}_{s_i}$$

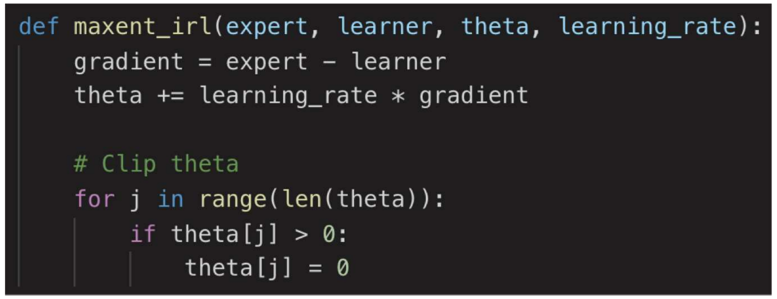
$$= \tilde{f} - f_\zeta$$

$$\theta^* = \operatorname*{argmax}_\theta L(\theta) = \operatorname*{argmax}_\theta \sum_{\text{examples}} \log P(\tilde{\zeta}|\theta, T)$$

- RL step : Q-learning 사용 (sampling)

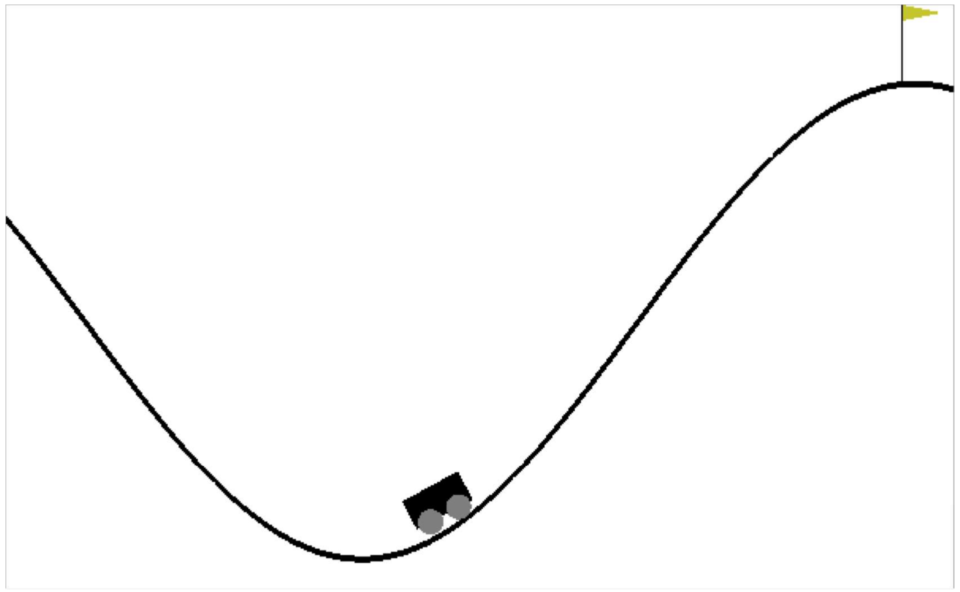


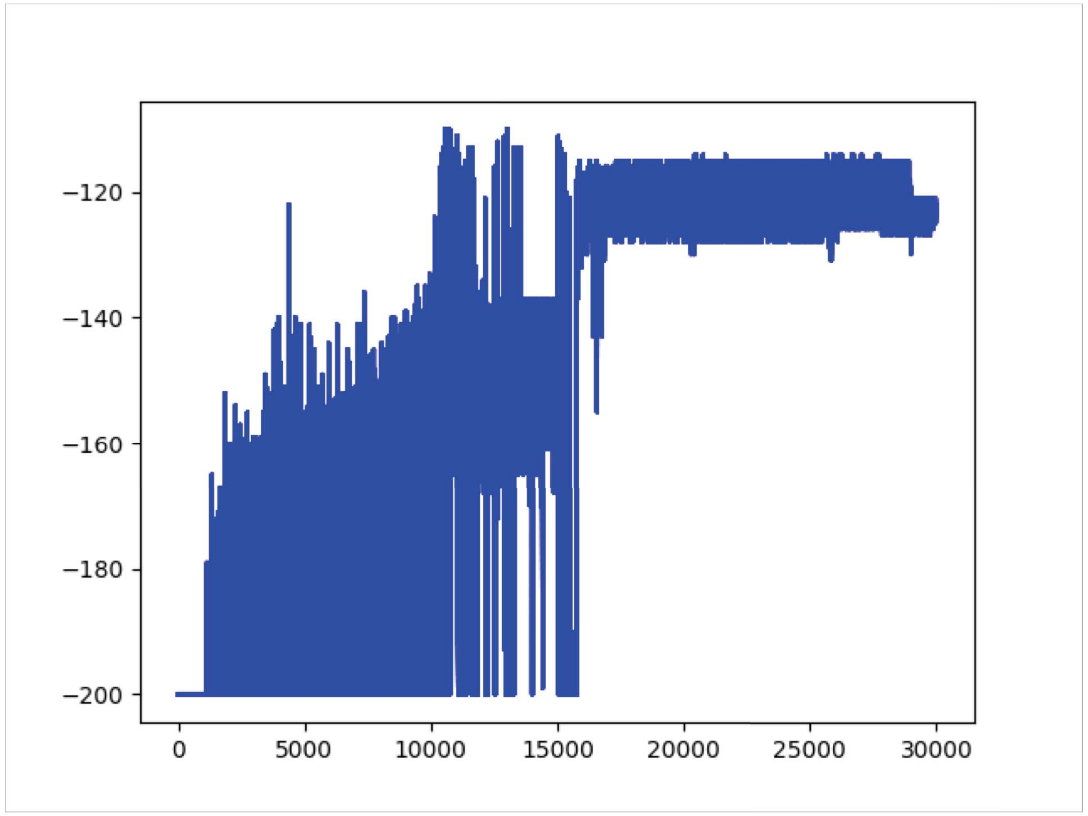
```python
def main():
    env = gym.make('MountainCar-v0')
    demonstrations = idx_demo(env, one_feature)

    expert = expert_feature_expectations(feature_matrix, demonstrations)
    learner_feature_expectations = np.zeros(n_states)

    theta = -(np.random.uniform(size=(n_states,)))

    episodes, scores = [], []

    for episode in range(30000):
        state = env.reset()
        score = 0

        if (episode != 0 and episode == 10000) or (episode > 10000 and episode % 5000 == 0):
            learner = learner_feature_expectations / episode
            maxent_irl(expert, learner, theta, theta_learning_rate)

        while True:
            state_idx = idx_state(env, state)
            action = np.argmax(q_table[state_idx])
            next_state, reward, done, _ = env.step(action)

            irl_reward = get_reward(feature_matrix, theta, n_states, state_idx)
            next_state_idx = idx_state(env, next_state)
            update_q_table(state_idx, action, irl_reward, next_state_idx)

            learner_feature_expectations += feature_matrix[int(state_idx)]

            score += reward
            state = next_state

            if done:
                scores.append(score)
                episodes.append(episode)
                break
```



```python
def maxent_irl(expert, learner, theta, learning_rate):
    gradient = expert - learner
    theta += learning_rate * gradient

    # Clip theta
    for j in range(len(theta)):
        if theta[j] > 0:
            theta[j] = 0
```

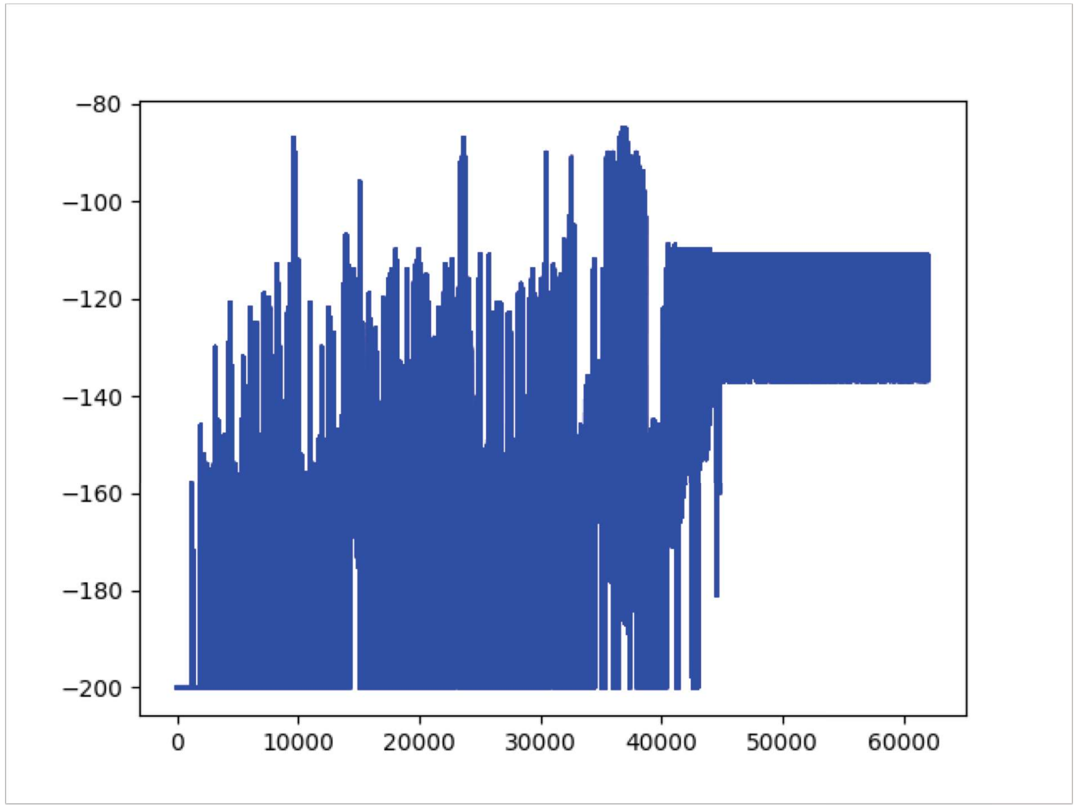
MaxEnt Result


Mountain car rendering

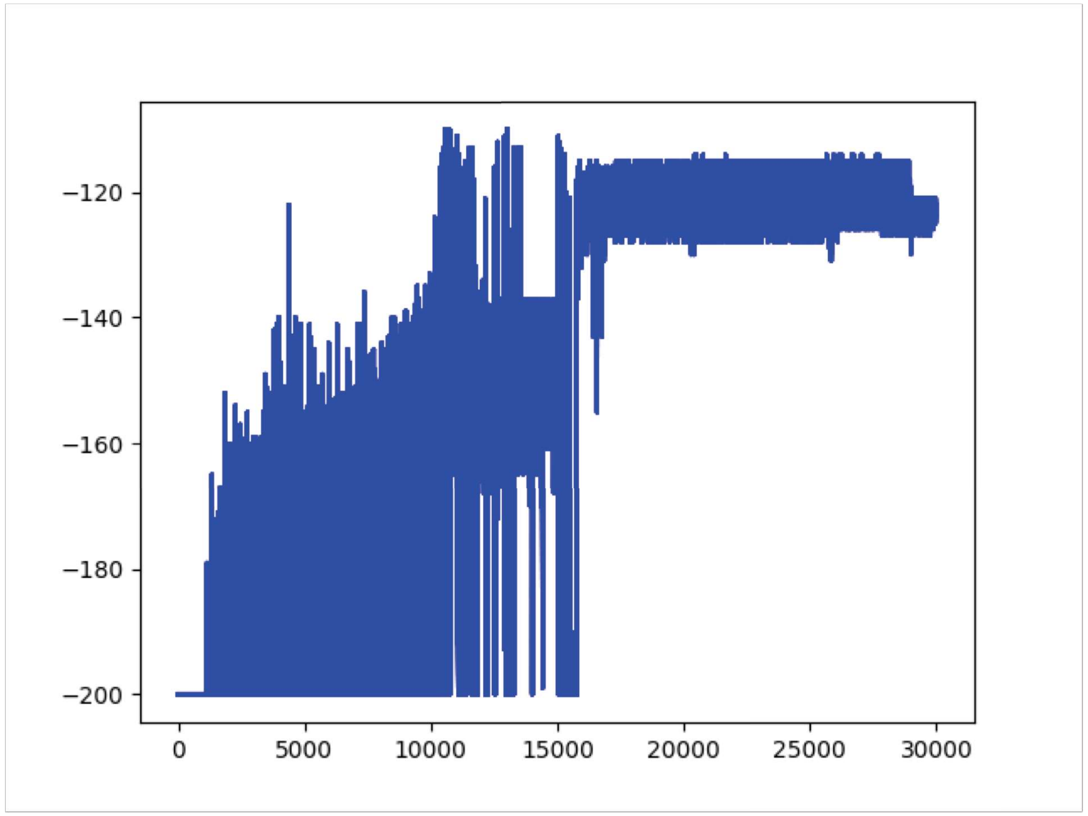


Learning curve

APP vs. MaxEnt Result



APP Learning curve



MaxEnt Learning curve

5. GAIL

Generative Adversarial Imitation Learning (2016)

References – GAIL

- Generative Adversarial Imitation Learning (Paper Link)
- SAIL-Toyota Seminar Series: Stefano Ermon (저자 직강) (YouTube Link)
- Generative Adversarial Imitation Learning (이경재님) (PPT Link)
- Cool Inverse Reinforcement Learning (IRL) Papers - IRL survey (최성준 박사님) (PDF Link)
- Review-Generative-Adversarial-Imitation-Learning (신명재님) (Markdown Link)
- Convex Optimization – Boyd and Vandenberghe (Book Link)
- 모두를 위한 컨벡스 최적화 – The conjugate function (Wikidocs Link)
- 모두를 위한 컨벡스 최적화 – Conjugate function (Wikidocs Link)
- 모두를 위한 컨벡스 최적화 – Lagrangian (Wikidocs Link)
- 모두를 위한 컨벡스 최적화 – Lagrangian dual function (Wikidocs Link)
- 모두를 위한 컨벡스 최적화 – Lagrangian dual problem (Wikidocs Link)
- 모두를 위한 컨벡스 최적화 – Strong duality (Wikidocs Link)
- Kullback-Leibler Divergence & Jensen-Shannon Divergence (Blog Link)
- PyTorch implementation of Deep Reinforcement Learning: GAIL (GitHub Link)

Summary - GAIL

- Imitation Learning + Generative Adversarial Network(GAN)
- Why must we learn a cost(reward) function?
  -> How to act by directly learning a policy
- Maximum causal entropy IRL
- Regularizer ψ, Occupancy measure ρ_π
- Merge RL and IRL
- Practical occupancy measure matching
- Proposed Algorithm : Generative adversarial imitation learning
- Proposed Idea : GAN

이론적인 내용이 워낙 많은 논문입니다.
그래서 간단하게 요약해서 말씀드릴려고 합니다.

추가적인 내용은 블로그를 참고해주세요!

Preliminaries

- S and A : finite state and action spaces
- $P(s'|s,a)$: dynamics model
- Π : set of stationary stochastic policies that select action A at a given state S
- In γ-discounted infinite horizon setting, expectation w.r.t the trajectory a policy π generates

$$\mathbb{E}_\pi[c(s,a)] \triangleq \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t c(s_t, a_t)\right]$$

- π_E : expert's policy

IRL and RL

- Inverse Reinforcement Learning

$$\underset{c\in\mathbb{R}^{\mathcal{S}\times\mathcal{A}}}{\arg\max} -\psi(c) + \left(\min_{\pi\in\Pi} -H(\pi) + \mathbb{E}_{\pi}[c(s,a)]\right) - \mathbb{E}_{\pi_E}[c(s,a)]$$

  - $\psi(c)$: (closed, proper) convex cost function regularizer
  - $H(\pi) \triangleq \mathbb{E}_{\pi}[-\log \pi(a|s)]$: concave causal entropy of the policy π
  - $c = IRL(\pi_E)$
  - $\pi = \mathrm{RL(c)}$

- Reinforcement Learning

$$\min_{\pi\in\Pi} -H(\pi) + \mathbb{E}_{\pi}[c(s,a)]$$

Similarity between GANs and IRL

	GANs	IRL
Problem	$\min_G \max_D$	$\max_c \min_\pi$
Generator	$G(z)$	$\pi(\mathrm{s})$
Discriminator	$D(x)$	$c(\mathrm{s}, \mathrm{a})$

- $\therefore \min_G \max_D = \max_c \min_\pi$
- The goal is to show that

$$RL \circ IRL(\pi_E) = \min_\pi \max_c$$

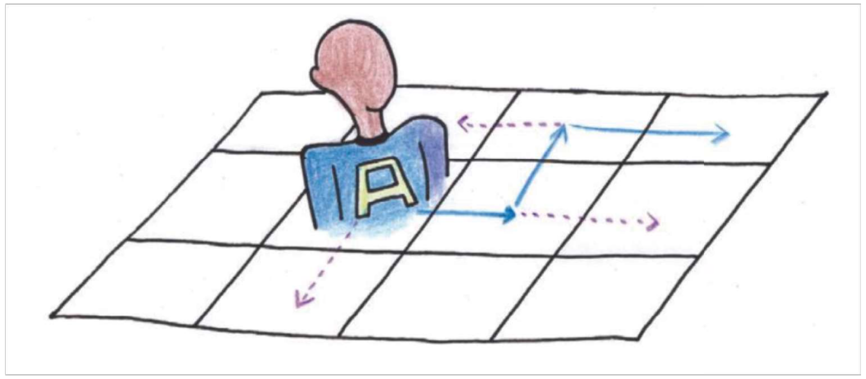
source : SNU AI 스터디 2기 – 170912_GAIL_이경재.pdf
source link : https://drive.google.com/drive/folders/0B8z5oUpB2DysSm1lOV9yeXRULVE

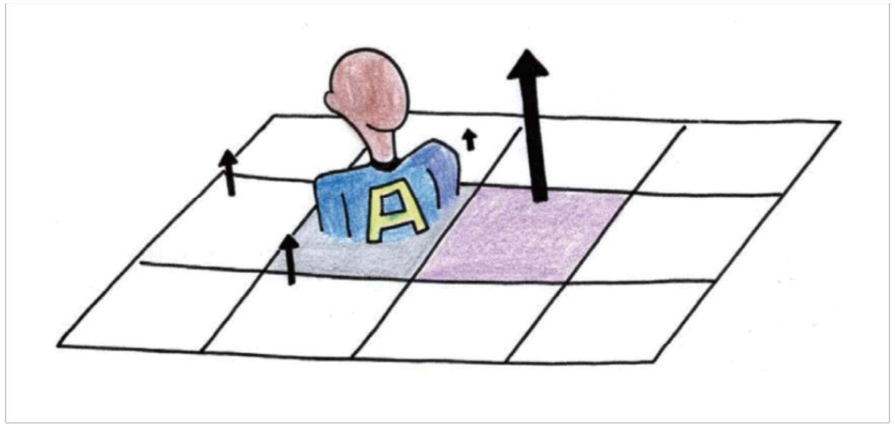
Dual Problem of RL

- Occupancy Measure (= state-action visitation frequency counts used in MMP)

$$\rho_\pi(s,a) = \pi(a|s)\sum_{t=0}^{\infty}\gamma^t P(s_t = s|\pi)$$

  - The occupancy measure can be interpreted as the unnormalized distribution of state-action pairs that an agent encounters when navigating the environment with the policy π





source : 파이썬과 케라스로 배우는 강화학습 저자특강
image source : https://www.slideshare.net/WoongwonLee/ss-78783597

Reinforcement Learning + Inverse Reinforcement Learning

Merge RL and IRL

- Imitation Learning via Inverse Reinforcement Learning :

$$\mathrm{RL} \circ \mathrm{IRL}_{\psi}(\pi_E) = \arg\min_{\pi \in \Pi} -H(\pi) + \psi^*(\rho_\pi - \rho_{\pi_E})$$

  - various settings of ψ lead to various imitation learning algorithms that directly solve the optimization problem
    - If ψ is a constant function
    - If ψ is a indicator function
    - If ψ is another new function

Imitation Learning + GAN

- New cost regularizer

$$\psi_{\text{GA}}(c) \triangleq \begin{cases} \mathbb{E}_{\pi_E}[g(c(s,a))] & \text{if } c < 0 \\ +\infty & \text{otherwise} \end{cases} \quad \text{where } g(x) = \begin{cases} -x - \log(1-e^x) & \text{if } x < 0 \\ +\infty & \text{otherwise} \end{cases}$$

- ψ_{GA} is motivated by the following fact

$$\psi^*_{\text{GA}}(\rho_\pi - \rho_{\pi_E}) = \sup_{D \in (0,1)^{\mathcal{S}\times\mathcal{A}}} \mathbb{E}_\pi[\log(D(s,a))] + \mathbb{E}_{\pi_E}[\log(1 - D(s,a))]$$

- New Imitation Learning Algorithm

$$\underset{\pi}{\text{minimize}}\ \psi^*_{\text{GA}}(\rho_\pi - \rho_{\pi_E}) - \lambda H(\pi) = D_{\text{JS}}(\rho_\pi, \rho_{\pi_E}) - \lambda H(\pi)$$

Imitation Learning + GAN

- New cost regularizer

$$\psi_{\mathrm{GA}}(c) \triangleq \begin{cases} \mathbb{E}_{\pi_E}[g(c(s,a))] & \text{if } c < 0 \\ +\infty & \text{otherwise} \end{cases} \quad \text{where } g(x) = \begin{cases} -x - \log(1 - e^{x}) & \text{if } x < 0 \\ +\infty & \text{otherwise} \end{cases}$$

- ψ_{GA} is motivated by the following fact -> proof?

$$\psi^{*}_{\mathrm{GA}}(\rho_\pi - \rho_{\pi_E}) = \sup_{D \in (0,1)^{\mathcal{S} \times \mathcal{A}}} \mathbb{E}_\pi[\log(D(s,a))] + \mathbb{E}_{\pi_E}[\log(1 - D(s,a))]$$

- New Imitation Learning Algorithm

$$\underset{\pi}{\text{minimize}} \ \psi^{*}_{\mathrm{GA}}(\rho_\pi - \rho_{\pi_E}) - \lambda H(\pi) = D_{\mathrm{JS}}(\rho_\pi, \rho_{\pi_E}) - \lambda H(\pi)$$

Imitation Learning + GAN

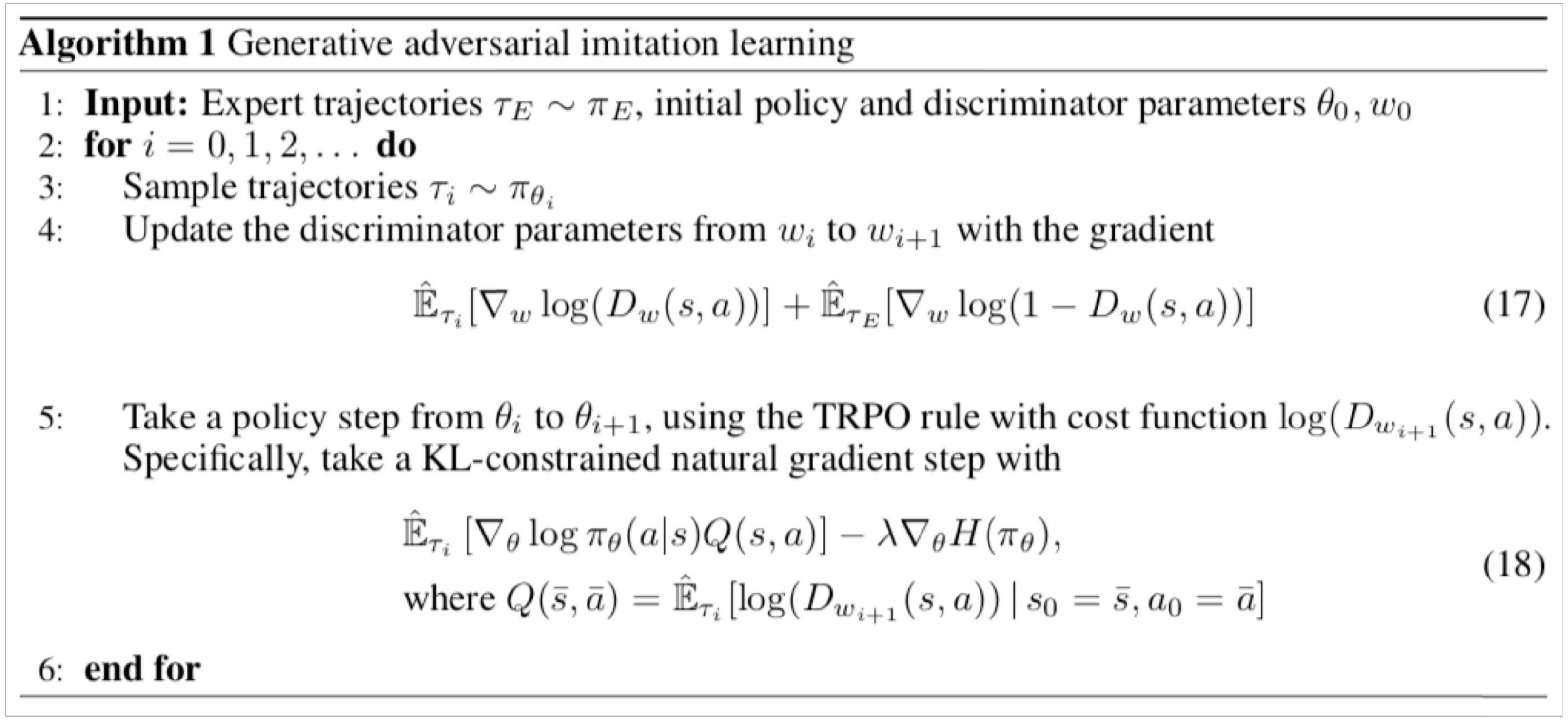
- Derivation of Conjugate Function

$$
\begin{aligned}
&\psi^*_{GA}(\rho_\pi - \rho_{\pi_E}) \;\; (\psi^*(x) = \sup_y x^T y - \psi(y)) \\
&= \max_{c \le 0} \sum_{s,a} (\rho_\pi(s,a) - \rho_{\pi_E}(s,a)) c(s,a) - \sum_{s,a} \rho_{\pi_E} g(c(s,a)) \\
&= \sum_{s,a} \max_{c(s,a) \le 0} [(\rho_\pi(s,a) - \rho_{\pi_E}(s,a)) c(s,a) - \rho_{\pi_E}(s,a) g(c(s,a))] \\
&= \sum_{s,a} \max_{x \le 0} [(\rho_\pi(s,a) - \rho_{\pi_E}(s,a)) x - \rho_{\pi_E}(s,a)(-x - \log(1 - \exp(x)))] \\
&= \sum_{s,a} \max_{y \in \mathcal{R}} [(\rho_\pi(s,a) \log(\frac{1}{1 + \exp(-y)}) + \rho_{\pi_E}(s,a) \log(\frac{\exp(-y)}{1 + \exp(-y)})] \\
&= \sum_{s,a} \max_{d \in (0,1)} [(\rho_\pi(s,a) \log(d) + \rho_{\pi_E}(s,a) \log(1 - d)]
\end{aligned}
$$

source : SNU AI 스터디 2기 – 170912_GAIL_이경재.pdf
source link : https://drive.google.com/drive/folders/0B8z5oUpB2DysSm1lOV9yeXRULVE

Proposed Algorithm

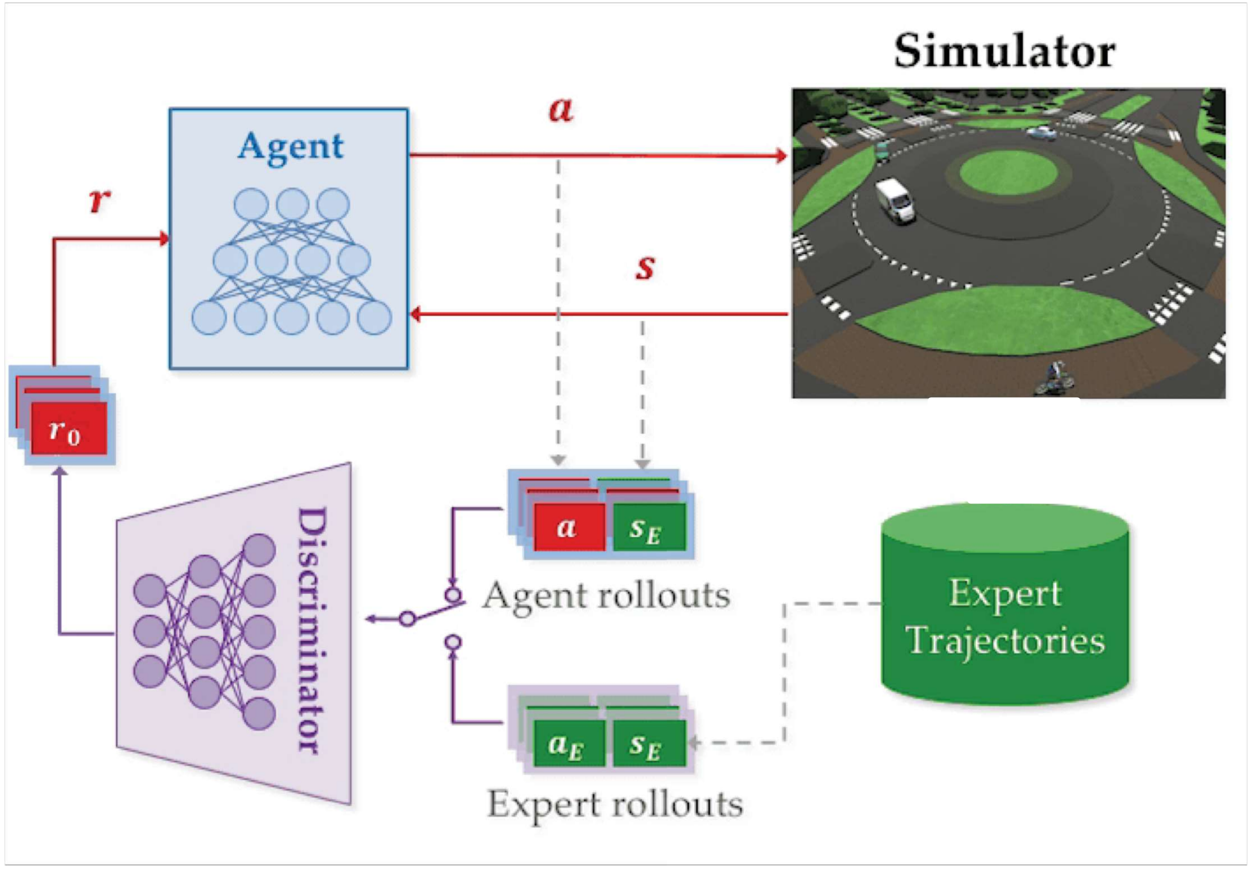
- Generative adversarial imitation learning



Algorithm 1 Generative adversarial imitation learning

```
1: Input: Expert trajectories $\tau_E \sim \pi_E$, initial policy and discriminator parameters $\theta_0, w_0$
2: for $i = 0, 1, 2, \ldots$ do
3:     Sample trajectories $\tau_i \sim \pi_{\theta_i}$
4:     Update the discriminator parameters from $w_i$ to $w_{i+1}$ with the gradient
           $\hat{\mathbb{E}}_{\tau_i}[\nabla_w \log(D_w(s,a))] + \hat{\mathbb{E}}_{\tau_E}[\nabla_w \log(1 - D_w(s,a))]$    (17)
5:     Take a policy step from $\theta_i$ to $\theta_{i+1}$, using the TRPO rule with cost function $\log(D_{w_{i+1}}(s,a))$.
       Specifically, take a KL-constrained natural gradient step with
           $\hat{\mathbb{E}}_{\tau_i}\left[\nabla_\theta \log \pi_\theta(a|s) Q(s,a)\right] - \lambda \nabla_\theta H(\pi_\theta),$
           where $Q(\bar{s}, \bar{a}) = \hat{\mathbb{E}}_{\tau_i}[\log(D_{w_{i+1}}(s,a)) \mid s_0 = \bar{s}, a_0 = \bar{a}]$    (18)
6: end for
```

GAIL

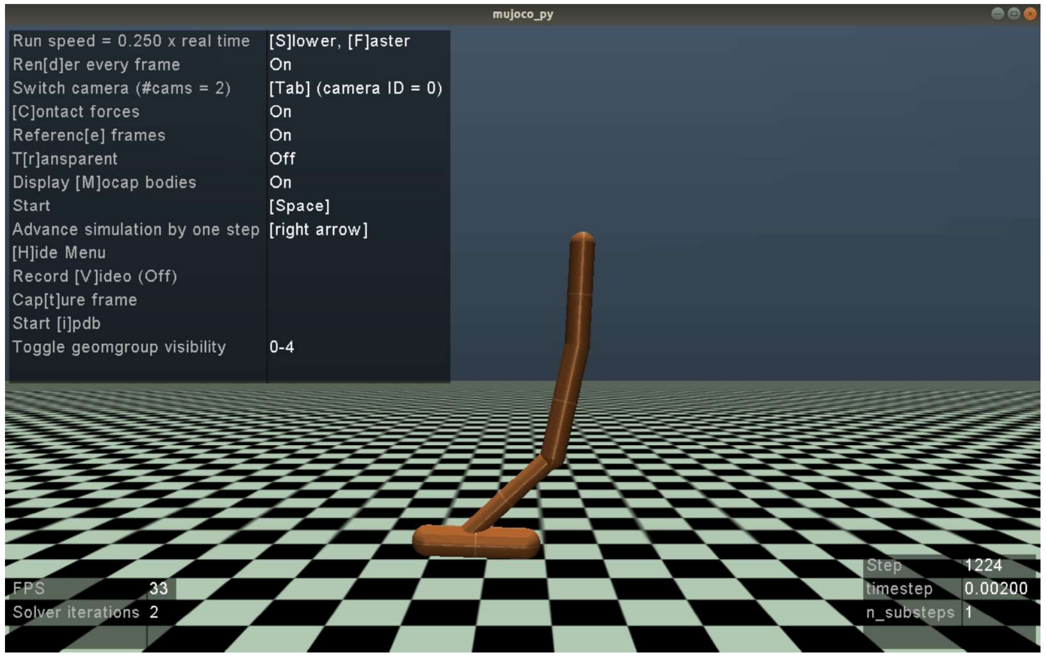


source : Learning from Demonstration in the Wild
image source : http://bitly.kr/Bcup4

GAIL의 구현이야기

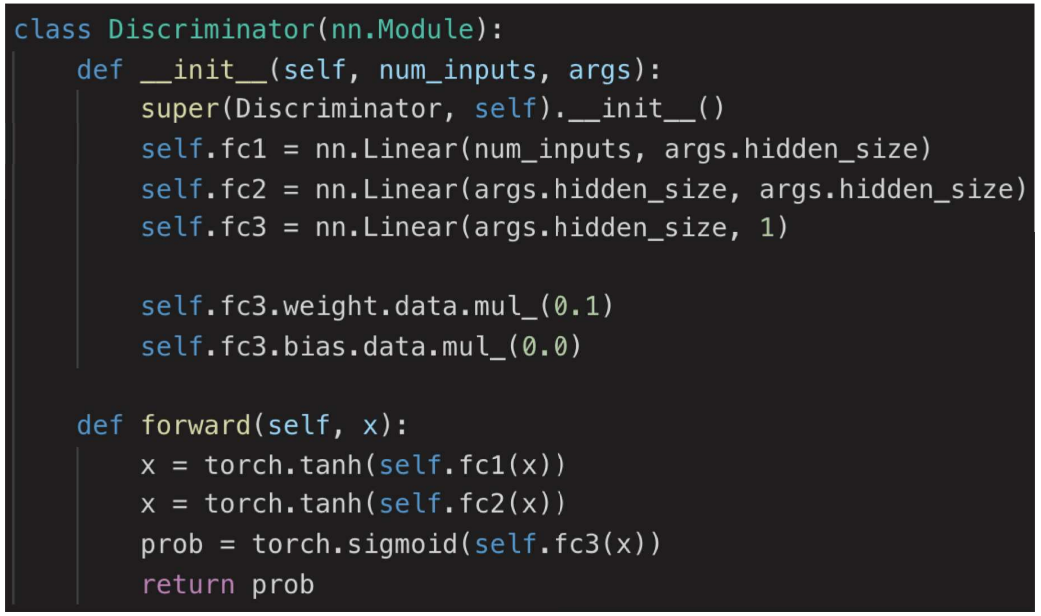
- Environment : Mujoco Hopper v2
- 11 continuous observation spaces
    - 관절의 위치, 각도, 각속도
- 3 continuous action spaces
    - 관절에 가해지는 토크
- Reward function
    - 앞으로 나아가는 속도
- Goal
    - 최대한 앞으로 많이 나아가기

- Expert’s demonstrations
    - (50000, 14)
    - (demo_num, state + action)
    - On average, scores between about 2200 and 2600



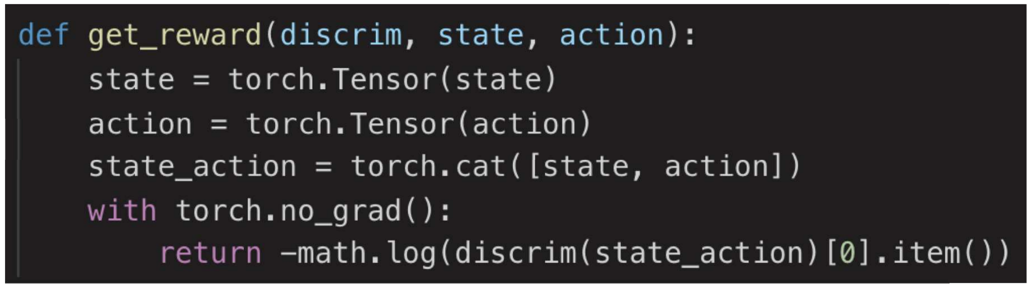
```
Run speed = 0.250 x real time   [S]lower, [F]aster
Ren[d]er every frame            On
Switch camera (#cams = 2)       [Tab] (camera ID = 0)
[C]ontact forces                On
Referenc[e] frames              On
T[r]ansparent                   Off
Display [M]ocap bodies          On
Start                           [Space]
Advance simulation by one step  [right arrow]
[H]ide Menu
Record [V]ideo (Off)
Cap[t]ure frame
Start [i]pdb
Toggle geomgroup visibility     0-4

FPS                33
Solver iterations  2

Step        1224
timestep    0.00200
n_substeps  1
```

- Discriminator
    - Num_inputs : 14, hidden_size : 100



```python
class Discriminator(nn.Module):
    def __init__(self, num_inputs, args):
        super(Discriminator, self).__init__()
        self.fc1 = nn.Linear(num_inputs, args.hidden_size)
        self.fc2 = nn.Linear(args.hidden_size, args.hidden_size)
        self.fc3 = nn.Linear(args.hidden_size, 1)

        self.fc3.weight.data.mul_(0.1)
        self.fc3.bias.data.mul_(0.0)

    def forward(self, x):
        x = torch.tanh(self.fc1(x))
        x = torch.tanh(self.fc2(x))
        prob = torch.sigmoid(self.fc3(x))
        return prob
```

- Discriminator
  - Num_inputs : 14, hidden_size : 100
- Get reward
  - $-\log(D_{w_{i+1}}(s, a))$

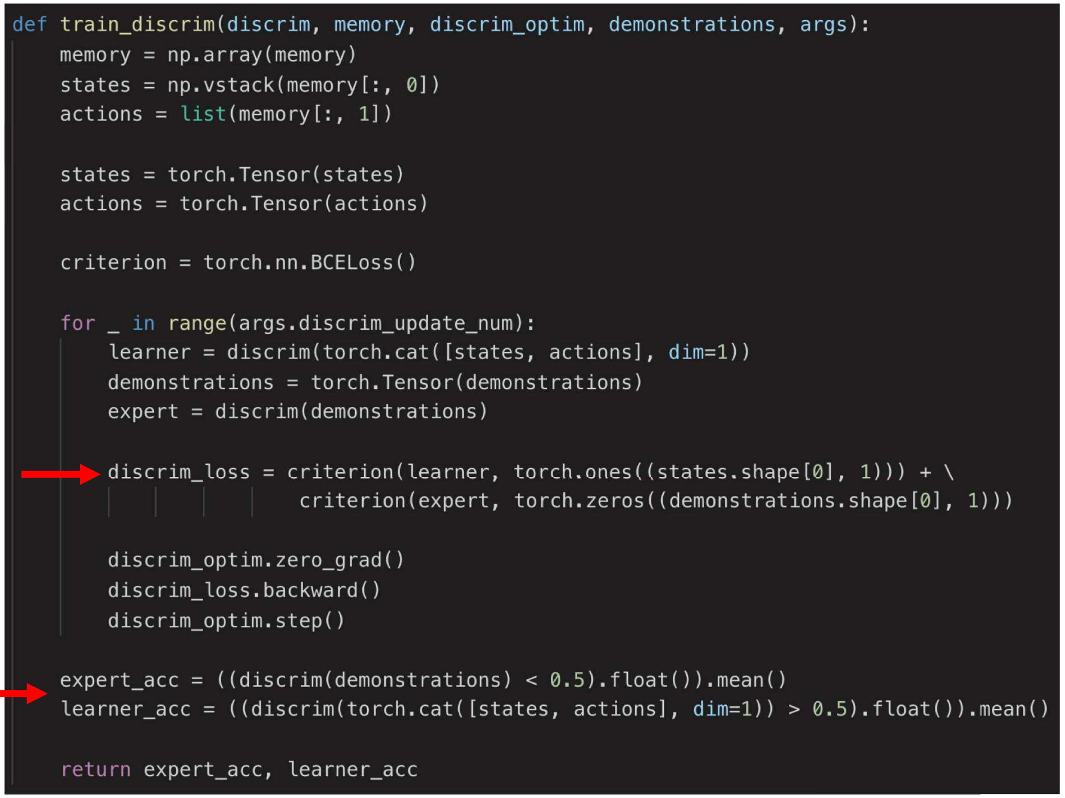


```python
def get_reward(discrim, state, action):
    state = torch.Tensor(state)
    action = torch.Tensor(action)
    state_action = torch.cat([state, action])
    with torch.no_grad():
        return -math.log(discrim(state_action)[0].item())
```

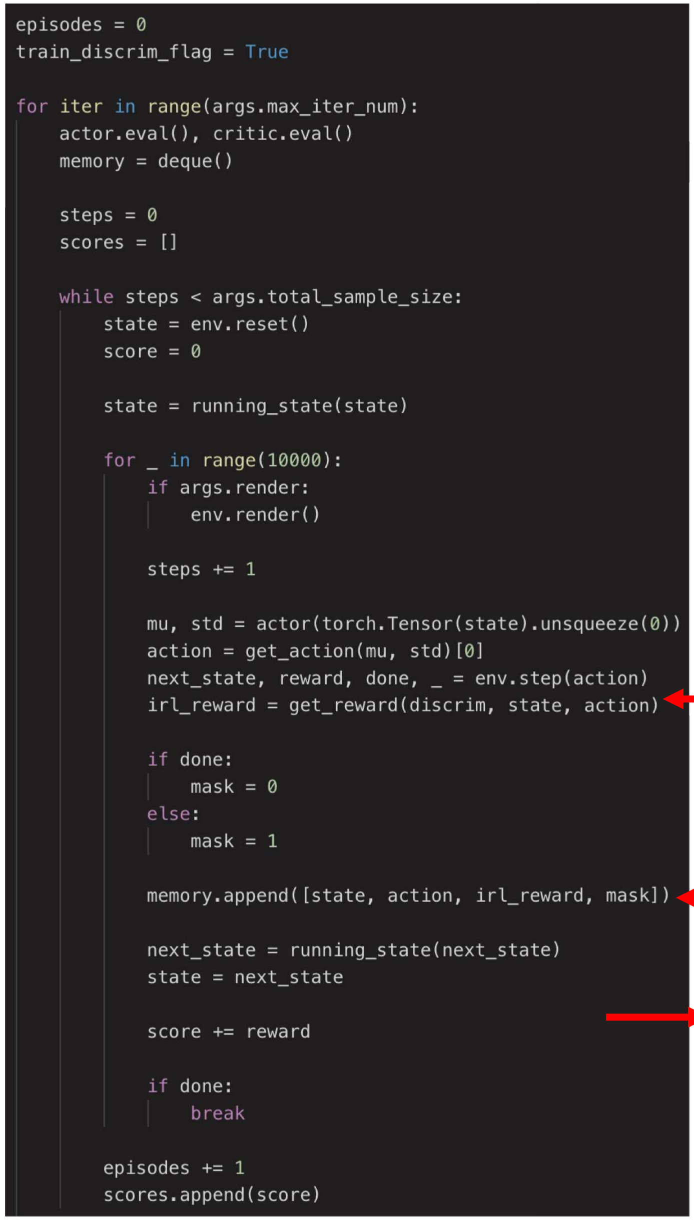
- Discriminator
  - Num_inputs : 14, hidden_size : 100
- Get reward
  - $-\log(D_{w_{i+1}}(s,a))$
- Train Discriminator
  - Learner's behavior이면 1을 출력
    Expert's demonstration이면 0을 출력

$$\hat{\mathbb{E}}_{\tau_i}[\nabla_w \log(D_w(s,a))] + \hat{\mathbb{E}}_{\tau_E}[\nabla_w \log(1 - D_w(s,a))]$$

  - Check accuracy of expert and learner

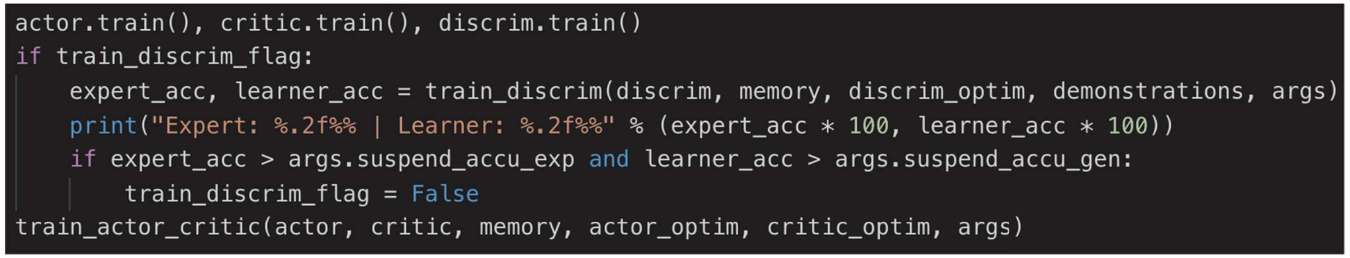


```python
def train_discrim(discrim, memory, discrim_optim, demonstrations, args):
    memory = np.array(memory)
    states = np.vstack(memory[:, 0])
    actions = list(memory[:, 1])

    states = torch.Tensor(states)
    actions = torch.Tensor(actions)

    criterion = torch.nn.BCELoss()

    for _ in range(args.discrim_update_num):
        learner = discrim(torch.cat([states, actions], dim=1))
        demonstrations = torch.Tensor(demonstrations)
        expert = discrim(demonstrations)

        discrim_loss = criterion(learner, torch.ones((states.shape[0], 1))) + \
                        criterion(expert, torch.zeros((demonstrations.shape[0], 1)))

        discrim_optim.zero_grad()
        discrim_loss.backward()
        discrim_optim.step()

    expert_acc = ((discrim(demonstrations) < 0.5).float()).mean()
    learner_acc = ((discrim(torch.cat([states, actions], dim=1)) > 0.5).float()).mean()

    return expert_acc, learner_acc
```



```python
episodes = 0
train_discrim_flag = True

for iter in range(args.max_iter_num):
    actor.eval(), critic.eval()
    memory = deque()

    steps = 0
    scores = []

    while steps < args.total_sample_size:
        state = env.reset()
        score = 0

        state = running_state(state)

        for _ in range(10000):
            if args.render:
                env.render()

            steps += 1

            mu, std = actor(torch.Tensor(state).unsqueeze(0))
            action = get_action(mu, std)[0]
            next_state, reward, done, _ = env.step(action)
            irl_reward = get_reward(discrim, state, action)

            if done:
                mask = 0
            else:
                mask = 1

            memory.append([state, action, irl_reward, mask])

            next_state = running_state(next_state)
            state = next_state

            score += reward

            if done:
                break

        episodes += 1
        scores.append(score)
```

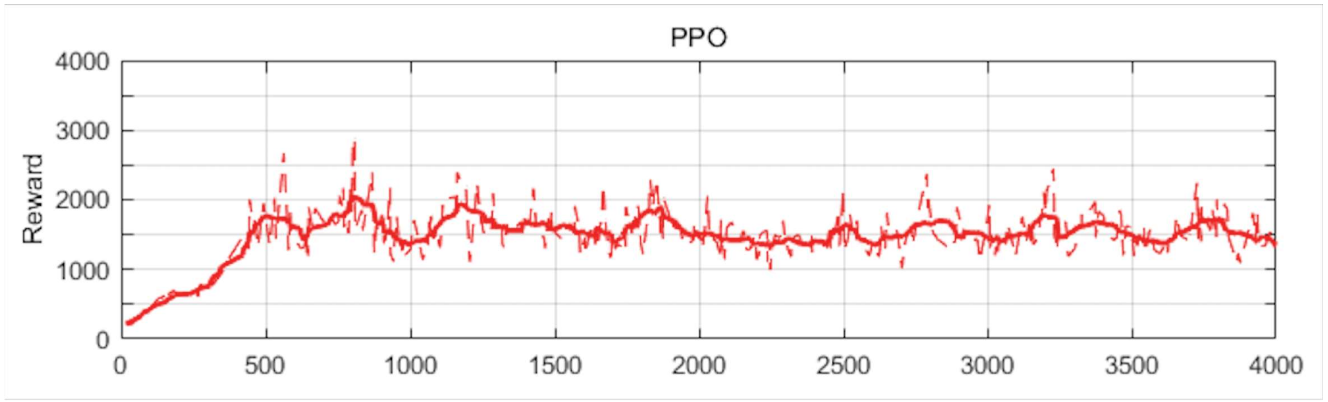
- IRL Step
  - 2 update number of discriminator
- RL Step : PPO 사용
  - 10 update number of actor-critic
- Maximum iteration number : 4000
- Stop when accuracy of expert and learner from discriminator > 80%



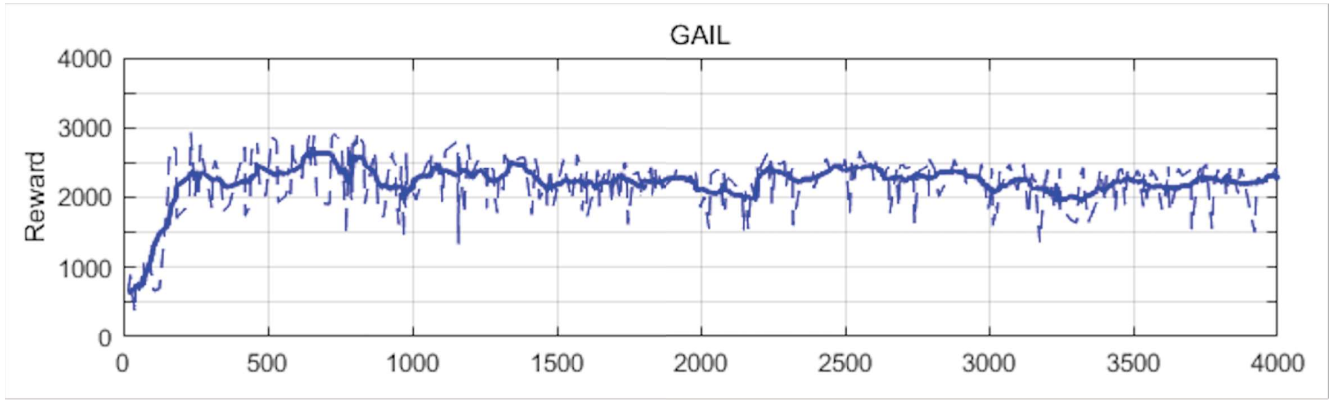
```python
actor.train(), critic.train(), discrim.train()
if train_discrim_flag:
    expert_acc, learner_acc = train_discrim(discrim, memory, discrim_optim, demonstrations, args)
    print("Expert: %.2f%% | Learner: %.2f%%" % (expert_acc * 100, learner_acc * 100))
    if expert_acc > args.suspend_accu_exp and learner_acc > args.suspend_accu_gen:
        train_discrim_flag = False
train_actor_critic(actor, critic, memory, actor_optim, critic_optim, args)
```

GAIL Result

PPO Learning curve
(to compare)



GAIL Learning curve



5. VAIL

Variational Adversarial Imitation Learning (2018)

References – VAIL

- Variational Discriminator Bottleneck: Improving Imitation Learning, Inverse RL, and GANs by constraining information flow (Paper Link)
- 오토인코더의 모든 것 – 1/3 (이활석님) (YouTube Link)
- 오토인코더의 모든 것 – 2/3 (이활석님) (YouTube Link)
- 오토인코더의 모든 것 – 3/3 (이활석님) (YouTube Link)
- Variational Autoencoders Explained (Blog Link)

Summary - VAIL

- Technique to constrain information flow in the discriminator by means of an information bottleneck

- New regularizer
  - Directly limit discriminator’s accuracy
  - Ignore irrelevant cues
  - improving the most discerning differences between real and fake samples

- Variational Information Bottleneck (VIB) <- similar approach in VAE

- Variational Discriminator Bottleneck (VDB)

- Proposed Algorithm : Variational adversarial imitation learning

- Proposed Idea : GAN + VDB

Variational Information Bottleneck

- Given a dataset $\{x_i, y_i\}$, with features x_i and labels y_i, the **standard maximum likelihood estimate** $q(y_i|x_i)$

$$\min_{q} \quad \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p(\mathbf{x},\mathbf{y})} \left[-\log q(\mathbf{y}|\mathbf{x})\right]$$

- Regularizing model using an Information Bottleneck to encourage the model to focus only on the most discriminative features

$$\begin{aligned} J(q,E) = \min_{q,E} \quad & \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p(\mathbf{x},\mathbf{y})} \left[\mathbb{E}_{\mathbf{z}\sim E(\mathbf{z}|\mathbf{x})} \left[-\log q(\mathbf{y}|\mathbf{z})\right]\right] \\ \text{s.t.} \quad & I(X,Z) \leq I_c \end{aligned}$$

  - $E(z|x)$: an encoder that maps the features x to a latent distribution over Z
  - $I(X,Z)$: mutual information between the encoding and the original features
  - I_c : upper bound on the mutual information

Variational Information Bottleneck

- Upper bound on $I(X,Z)$ can be obtained via the **KL divergence**

$$I(X,Z) \leq \int p(\mathbf{x}) E(\mathbf{z}|\mathbf{x}) \log \frac{E(\mathbf{z}|\mathbf{x})}{r(\mathbf{z})} \, d\mathbf{x} \, d\mathbf{z} = \underline{\mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})} \left[\mathrm{KL}\left[E(\mathbf{z}|\mathbf{x}) || r(\mathbf{z})\right]\right]}$$

- The regularized objective function using KL divergence as upper bound

$$\begin{aligned} \tilde{J}(q,E) = \min_{q,E} \quad & \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p(\mathbf{x},\mathbf{y})} \left[\mathbb{E}_{\mathbf{z}\sim E(\mathbf{z}|\mathbf{x})} \left[-\log q(\mathbf{y}|\mathbf{z})\right]\right] \\ \text{s.t.} \quad & \mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})} \left[\mathrm{KL}\left[E(\mathbf{z}|\mathbf{x}) || r(\mathbf{z})\right]\right] \leq I_c \end{aligned}$$

- To solve this problem, the constraint can be subsumed into the objective with a **coefficient** β

$$\min_{q,E} \quad \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p(\mathbf{x},\mathbf{y})} \left[\mathbb{E}_{\mathbf{z}\sim E(\mathbf{z}|\mathbf{x})} \left[-\log q(\mathbf{y}|\mathbf{z})\right]\right] + \beta \left(\mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})} \left[\mathrm{KL}\left[E(\mathbf{z}|\mathbf{x}) || r(\mathbf{z})\right]\right] - I_c\right)$$

Variational Discriminator Bottleneck

- Standard GAN framework

$$\max_G \min_D \quad \mathbb{E}_{\mathbf{x}\sim p^*(\mathbf{x})}\left[-\log\left(D(\mathbf{x})\right)\right] + \mathbb{E}_{\mathbf{x}\sim G(\mathbf{x})}\left[-\log\left(1 - D(\mathbf{x})\right)\right]$$

  - $p^*(x)$: samples from the target distribution

- **The regularized objective function for the discriminator using** variational information bottleneck

$$\begin{aligned} J(D,E) = \min_{D,E} \quad & \mathbb{E}_{x\sim p^*(\mathbf{x})}\left[\mathbb{E}_{\mathbf{z}\sim E(\mathbf{z}|\mathbf{x})}\left[-\log\left(D(\mathbf{z})\right)\right]\right] + \mathbb{E}_{\mathbf{x}\sim G(\mathbf{x})}\left[\mathbb{E}_{\mathbf{z}\sim E(\mathbf{z}|\mathbf{x})}\left[-\log\left(1 - D(\mathbf{z})\right)\right]\right] \\ \text{s.t.} \quad & \mathbb{E}_{\mathbf{x}\sim \tilde{p}(\mathbf{x})}\left[\mathrm{KL}\left[E(\mathbf{z}|\mathbf{x}) || r(\mathbf{z})\right]\right] \leq I_c \end{aligned}$$

- To optimize this objective, use a **Lagrange multiplier** β

$$\begin{aligned} J(D,E) = \min_{D,E} \max_{\beta \geq 0} \quad & \mathbb{E}_{\mathbf{x}\sim p^*(\mathbf{x})}\left[\mathbb{E}_{\mathbf{z}\sim E(\mathbf{z}|\mathbf{x})}\left[-\log\left(D(\mathbf{z})\right)\right]\right] + \mathbb{E}_{\mathbf{x}\sim G(\mathbf{x})}\left[\mathbb{E}_{\mathbf{z}\sim E(\mathbf{z}|\mathbf{x})}\left[-\log\left(1 - D(\mathbf{z})\right)\right]\right] \\ & + \beta\left(\mathbb{E}_{\mathbf{x}\sim \tilde{p}(\mathbf{x})}\left[\mathrm{KL}\left[E(\mathbf{z}|\mathbf{x}) || r(\mathbf{z})\right]\right] - I_c\right) \end{aligned}$$

Variational Discriminator Bottleneck

- Adaptively update β via dual gradient descent to enforce a constraint I_c on the mutual information

$$
\begin{aligned}
D, E &\leftarrow \arg\min_{D,E} \mathcal{L}(D, E, \beta) \\
\beta &\leftarrow \max\left(0,\ \beta + \alpha_\beta \left(\mathbb{E}_{\mathbf{x}\sim\tilde{p}(\mathbf{x})}\left[\mathrm{KL}\left[E(\mathbf{z}|\mathbf{x})||r(\mathbf{z})\right]\right] - I_c\right)\right)
\end{aligned}
$$

- $\mathcal{L}(D, E, \beta)$ is the Lagrangian

$$
\begin{aligned}
\mathcal{L}(D, E, \beta) = &\ \mathbb{E}_{\mathbf{x}\sim p^*(\mathbf{x})}\left[\mathbb{E}_{\mathbf{z}\sim E(\mathbf{z}|\mathbf{x})}\left[-\log\left(D(\mathbf{z})\right)\right]\right] + \mathbb{E}_{\mathbf{x}\sim G(\mathbf{x})}\left[\mathbb{E}_{\mathbf{z}\sim E(\mathbf{z}|\mathbf{x})}\left[-\log\left(1 - D(\mathbf{z})\right)\right]\right] \\
&+ \beta\left(\mathbb{E}_{\mathbf{x}\sim\tilde{p}(\mathbf{x})}\left[\mathrm{KL}\left[E(\mathbf{z}|\mathbf{x})||r(\mathbf{z})\right]\right] - I_c\right)
\end{aligned}
$$

Proposed Algorithm

- Generative adversarial imitation learning

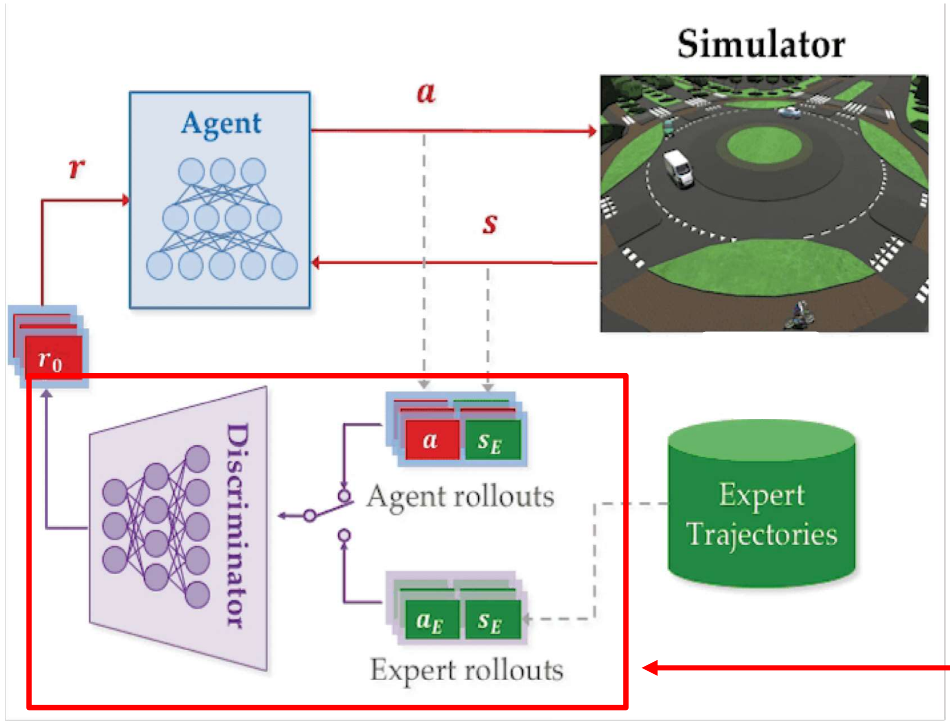
$$\max_{\pi} \min_{D} \quad \mathbb{E}_{\mathbf{s}\sim\pi^*(\mathbf{s})}\left[-\log\left(D(\mathbf{s})\right)\right] + \mathbb{E}_{\mathbf{s}\sim\pi(\mathbf{s})}\left[-\log\left(1 - D(\mathbf{s})\right)\right]$$

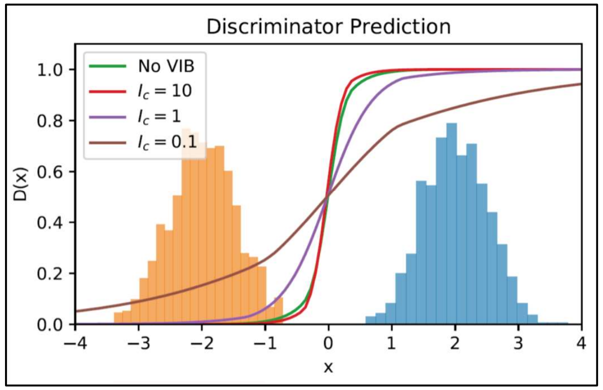
- Variational adversarial imitation learning

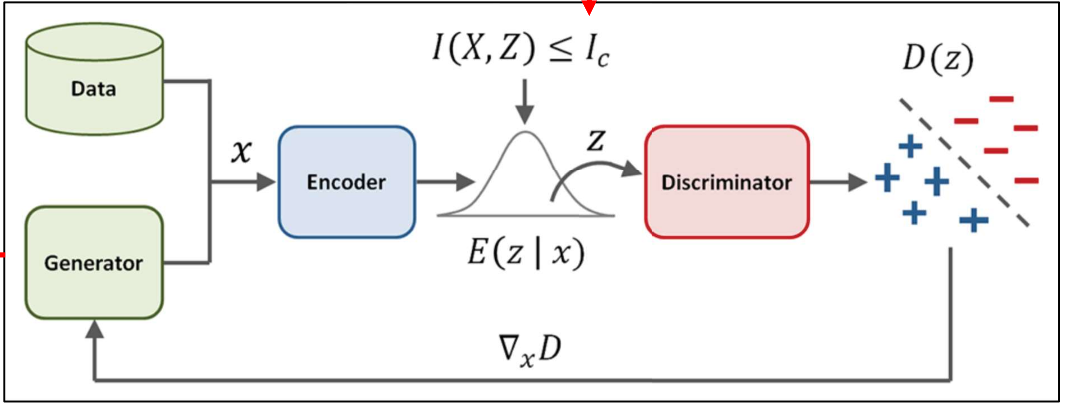
$$\min_{D,E} \max_{\beta\geq 0} \; \mathbb{E}_{\mathbf{s}\sim\pi^*(\mathbf{s})}\left[\mathbb{E}_{\mathbf{z}\sim E(\mathbf{z}|\mathbf{s})}\left[-\log\left(D(\mathbf{z})\right)\right]\right] + \mathbb{E}_{\mathbf{s}\sim\pi(\mathbf{s})}\left[\mathbb{E}_{\mathbf{z}\sim E(\mathbf{z}|\mathbf{s})}\left[-\log\left(1 - D(\mathbf{z})\right)\right]\right]$$
$$+ \beta\left(\mathbb{E}_{\mathbf{s}\sim\tilde{\pi}(\mathbf{s})}\left[\mathrm{KL}\left[E(\mathbf{z}|\mathbf{s})||r(\mathbf{z})\right]\right] - I_c\right).$$

  - $\tilde{\pi} = \frac{1}{2}\pi^* + \frac{1}{2}\pi$ represents a mixture of the expert's policy and the agent's policy

VAIL



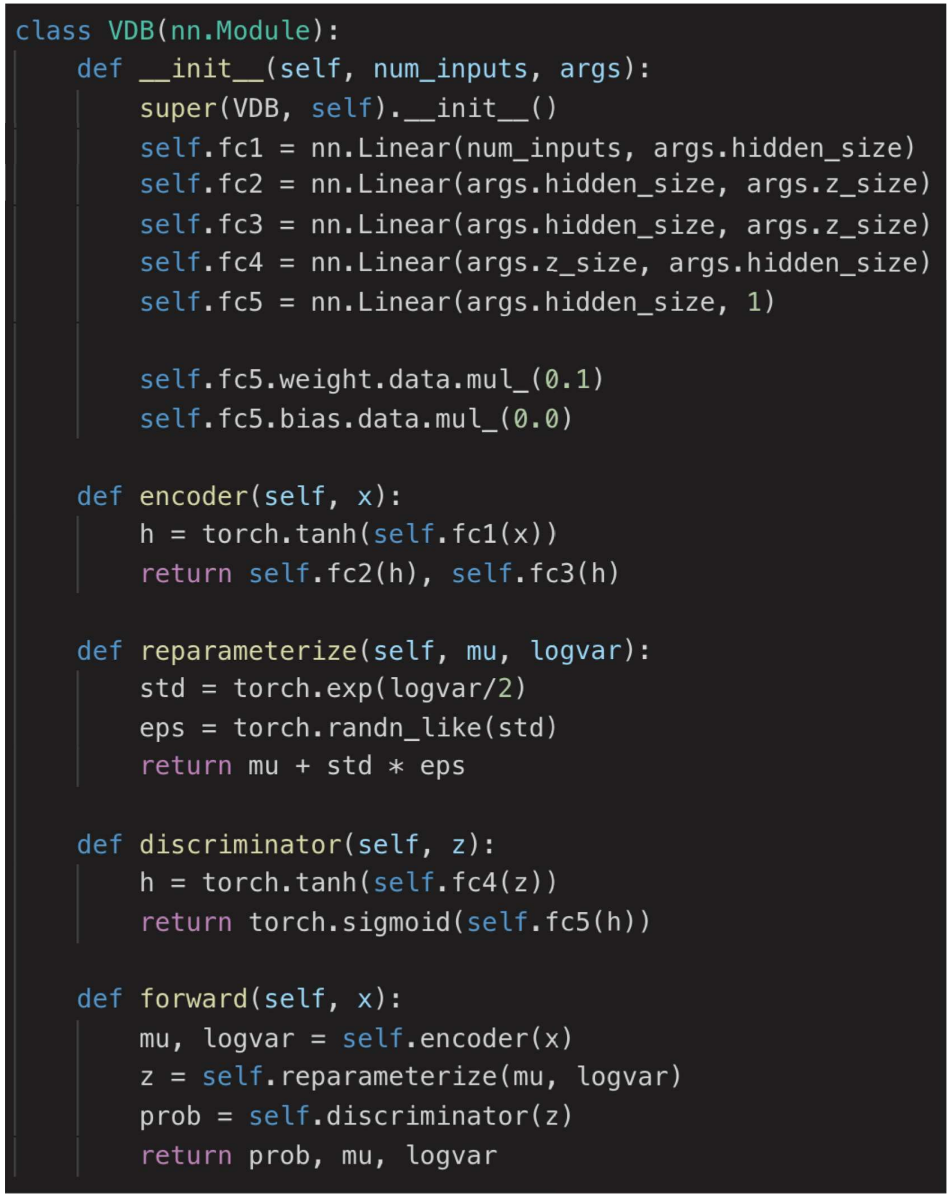




source : Learning from Demonstration in the Wild
image source : http://bitly.kr/Bcup4

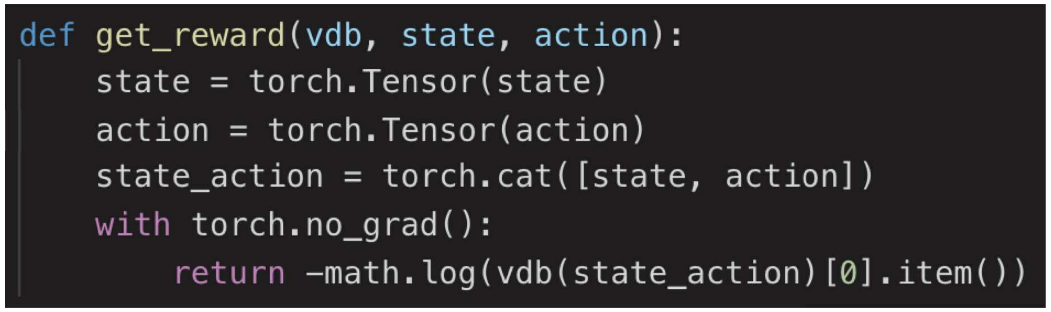
VAIL의 구현이야기

- Variational Discriminator Bottleneck
  - Num_inputs : 14, hidden_size : 100
  - Z_size : 4



```python
class VDB(nn.Module):
    def __init__(self, num_inputs, args):
        super(VDB, self).__init__()
        self.fc1 = nn.Linear(num_inputs, args.hidden_size)
        self.fc2 = nn.Linear(args.hidden_size, args.z_size)
        self.fc3 = nn.Linear(args.hidden_size, args.z_size)
        self.fc4 = nn.Linear(args.z_size, args.hidden_size)
        self.fc5 = nn.Linear(args.hidden_size, 1)

        self.fc5.weight.data.mul_(0.1)
        self.fc5.bias.data.mul_(0.0)

    def encoder(self, x):
        h = torch.tanh(self.fc1(x))
        return self.fc2(h), self.fc3(h)

    def reparameterize(self, mu, logvar):
        std = torch.exp(logvar/2)
        eps = torch.randn_like(std)
        return mu + std * eps

    def discriminator(self, z):
        h = torch.tanh(self.fc4(z))
        return torch.sigmoid(self.fc5(h))

    def forward(self, x):
        mu, logvar = self.encoder(x)
        z = self.reparameterize(mu, logvar)
        prob = self.discriminator(z)
        return prob, mu, logvar
```

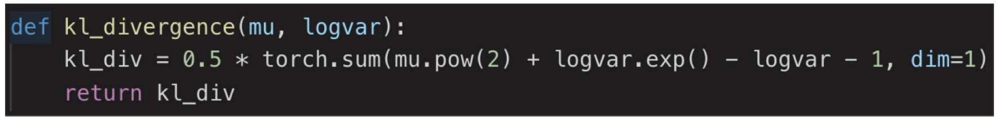
- Variational Discriminator Bottleneck
    - Num_inputs : 14, hidden_size : 100
    - Z_size : 4
- Get reward
    - $-\log(D_{w_{i+1}}(s,a))$



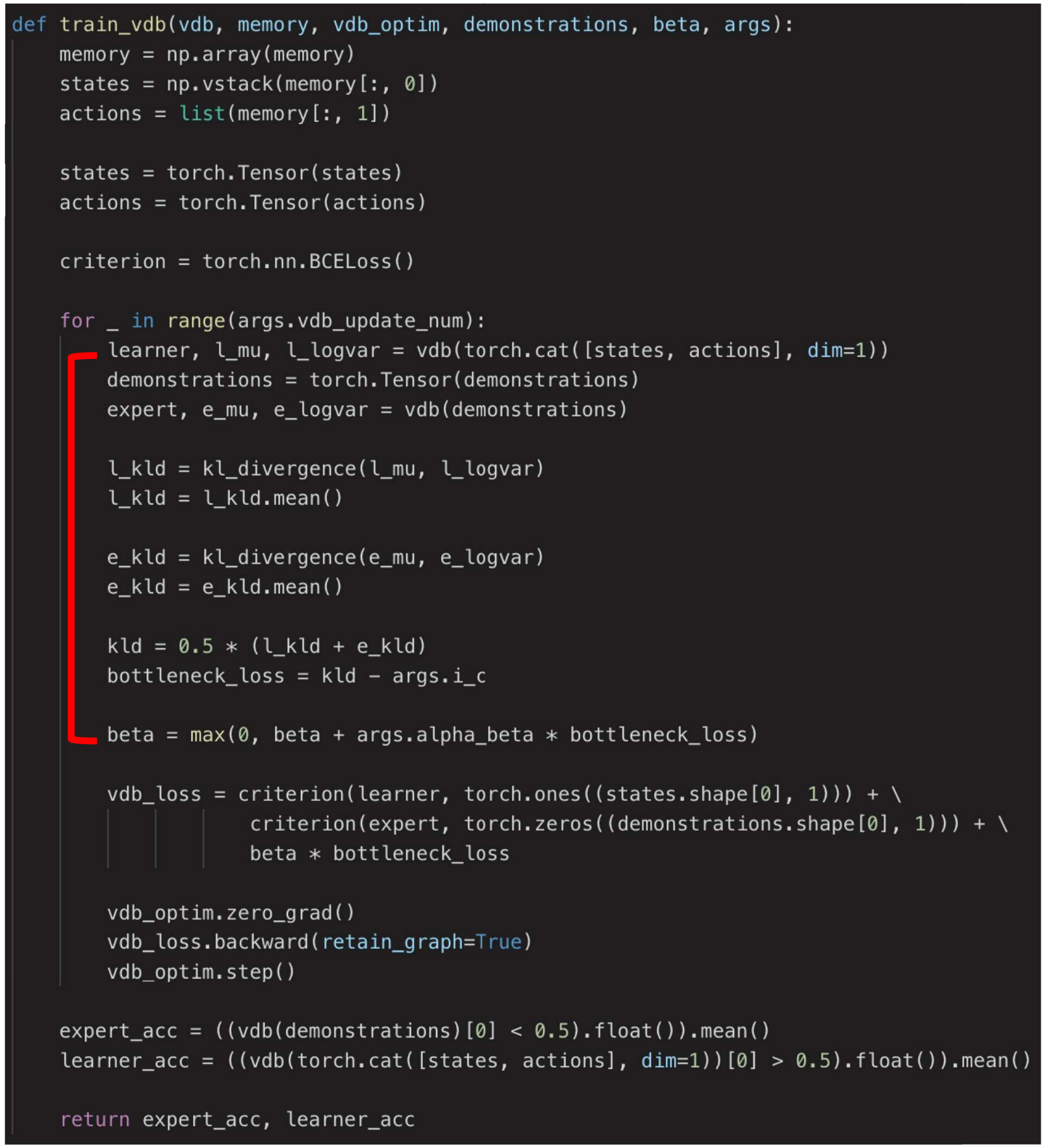
```python
def get_reward(vdb, state, action):
    state = torch.Tensor(state)
    action = torch.Tensor(action)
    state_action = torch.cat([state, action])
    with torch.no_grad():
        return -math.log(vdb(state_action)[0].item())
```

- Variational Discriminator Bottleneck
  - Num_inputs : 14, hidden_size : 100
  - Z_size : 4
- Get reward
  - $-\log(D_{w_{i+1}}(s,a))$
- Adaptively update β

$$\beta \leftarrow \max\left(0,\ \beta + \alpha_\beta \left(\mathbb{E}_{\mathbf{x}\sim\tilde{p}(\mathbf{x})}\left[\mathrm{KL}\left[E(\mathbf{z}|\mathbf{x})||r(\mathbf{z})\right]\right] - I_c\right)\right)$$



```python
def kl_divergence(mu, logvar):
    kl_div = 0.5 * torch.sum(mu.pow(2) + logvar.exp() - logvar - 1, dim=1)
    return kl_div
```



```python
def train_vdb(vdb, memory, vdb_optim, demonstrations, beta, args):
    memory = np.array(memory)
    states = np.vstack(memory[:, 0])
    actions = list(memory[:, 1])

    states = torch.Tensor(states)
    actions = torch.Tensor(actions)

    criterion = torch.nn.BCELoss()

    for _ in range(args.vdb_update_num):
        learner, l_mu, l_logvar = vdb(torch.cat([states, actions], dim=1))
        demonstrations = torch.Tensor(demonstrations)
        expert, e_mu, e_logvar = vdb(demonstrations)

        l_kld = kl_divergence(l_mu, l_logvar)
        l_kld = l_kld.mean()

        e_kld = kl_divergence(e_mu, e_logvar)
        e_kld = e_kld.mean()

        kld = 0.5 * (l_kld + e_kld)
        bottleneck_loss = kld - args.i_c

        beta = max(0, beta + args.alpha_beta * bottleneck_loss)

        vdb_loss = criterion(learner, torch.ones((states.shape[0], 1))) + \
                    criterion(expert, torch.zeros((demonstrations.shape[0], 1))) + \
                    beta * bottleneck_loss

        vdb_optim.zero_grad()
        vdb_loss.backward(retain_graph=True)
        vdb_optim.step()

    expert_acc = ((vdb(demonstrations)[0] < 0.5).float()).mean()
    learner_acc = ((vdb(torch.cat([states, actions], dim=1))[0] > 0.5).float()).mean()

    return expert_acc, learner_acc
```

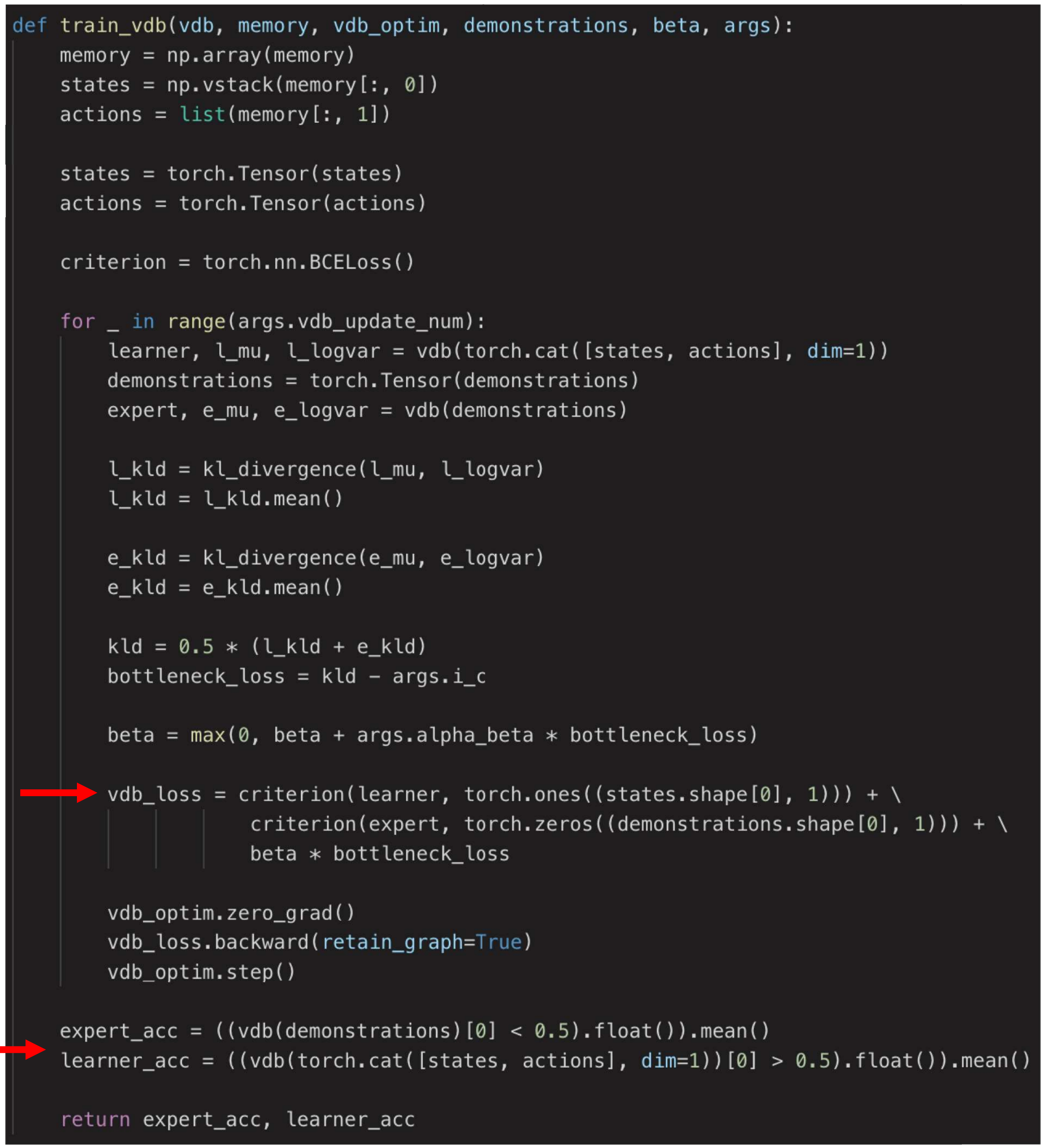
- Variational Discriminator Bottleneck
  - Num_inputs : 14, hidden_size : 100
  - Z_size : 4
- Get reward
  - $-\log(D_{w_{i+1}}(s,a))$
- Adaptively update β

$$\beta \leftarrow \max\left(0,\ \beta + \alpha_\beta \left(\mathbb{E}_{\mathbf{x}\sim\tilde{p}(\mathbf{x})}\left[\mathrm{KL}\left[E(\mathbf{z}|\mathbf{x})||r(\mathbf{z})\right]\right] - I_c\right)\right)$$

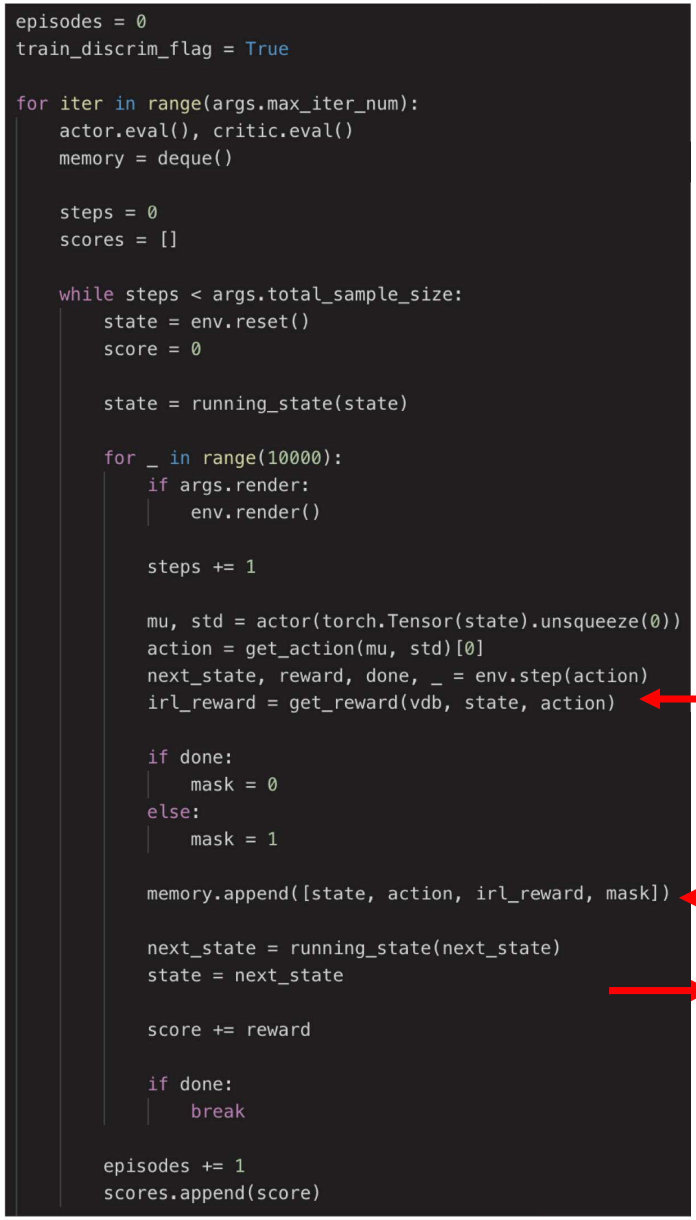
- Train VDB

$$\min_{D,E}\max_{\beta\geq 0}\ \mathbb{E}_{\mathbf{s}\sim\pi^*(\mathbf{s})}\left[\mathbb{E}_{\mathbf{z}\sim E(\mathbf{z}|\mathbf{s})}\left[-\log\left(D(\mathbf{z})\right)\right]\right] + \mathbb{E}_{\mathbf{s}\sim\pi(\mathbf{s})}\left[\mathbb{E}_{\mathbf{z}\sim E(\mathbf{z}|\mathbf{s})}\left[-\log\left(1-D(\mathbf{z})\right)\right]\right] + \beta\left(\mathbb{E}_{\mathbf{s}\sim\tilde{\pi}(\mathbf{s})}\left[\mathrm{KL}\left[E(\mathbf{z}|\mathbf{s})||r(\mathbf{z})\right]\right] - I_c\right)$$

  - Check accuracy of expert and learner

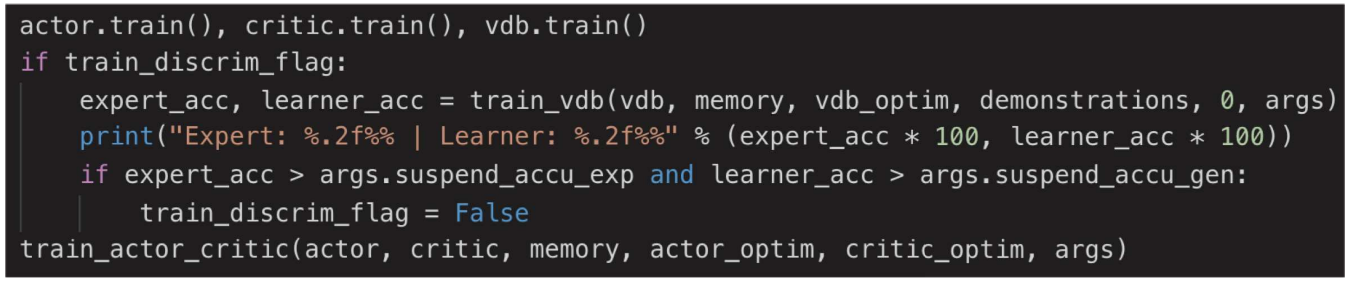


```python
def train_vdb(vdb, memory, vdb_optim, demonstrations, beta, args):
    memory = np.array(memory)
    states = np.vstack(memory[:, 0])
    actions = list(memory[:, 1])

    states = torch.Tensor(states)
    actions = torch.Tensor(actions)

    criterion = torch.nn.BCELoss()

    for _ in range(args.vdb_update_num):
        learner, l_mu, l_logvar = vdb(torch.cat([states, actions], dim=1))
        demonstrations = torch.Tensor(demonstrations)
        expert, e_mu, e_logvar = vdb(demonstrations)

        l_kld = kl_divergence(l_mu, l_logvar)
        l_kld = l_kld.mean()

        e_kld = kl_divergence(e_mu, e_logvar)
        e_kld = e_kld.mean()

        kld = 0.5 * (l_kld + e_kld)
        bottleneck_loss = kld - args.i_c

        beta = max(0, beta + args.alpha_beta * bottleneck_loss)

        vdb_loss = criterion(learner, torch.ones((states.shape[0], 1))) + \
                    criterion(expert, torch.zeros((demonstrations.shape[0], 1))) + \
                    beta * bottleneck_loss

        vdb_optim.zero_grad()
        vdb_loss.backward(retain_graph=True)
        vdb_optim.step()

    expert_acc = ((vdb(demonstrations)[0] < 0.5).float()).mean()
    learner_acc = ((vdb(torch.cat([states, actions], dim=1))[0] > 0.5).float()).mean()

    return expert_acc, learner_acc
```



```
episodes = 0
train_discrim_flag = True

for iter in range(args.max_iter_num):
    actor.eval(), critic.eval()
    memory = deque()

    steps = 0
    scores = []

    while steps < args.total_sample_size:
        state = env.reset()
        score = 0

        state = running_state(state)

        for _ in range(10000):
            if args.render:
                env.render()

            steps += 1

            mu, std = actor(torch.Tensor(state).unsqueeze(0))
            action = get_action(mu, std)[0]
            next_state, reward, done, _ = env.step(action)
            irl_reward = get_reward(vdb, state, action)

            if done:
                mask = 0
            else:
                mask = 1

            memory.append([state, action, irl_reward, mask])

            next_state = running_state(next_state)
            state = next_state

            score += reward

            if done:
                break

        episodes += 1
        scores.append(score)
```

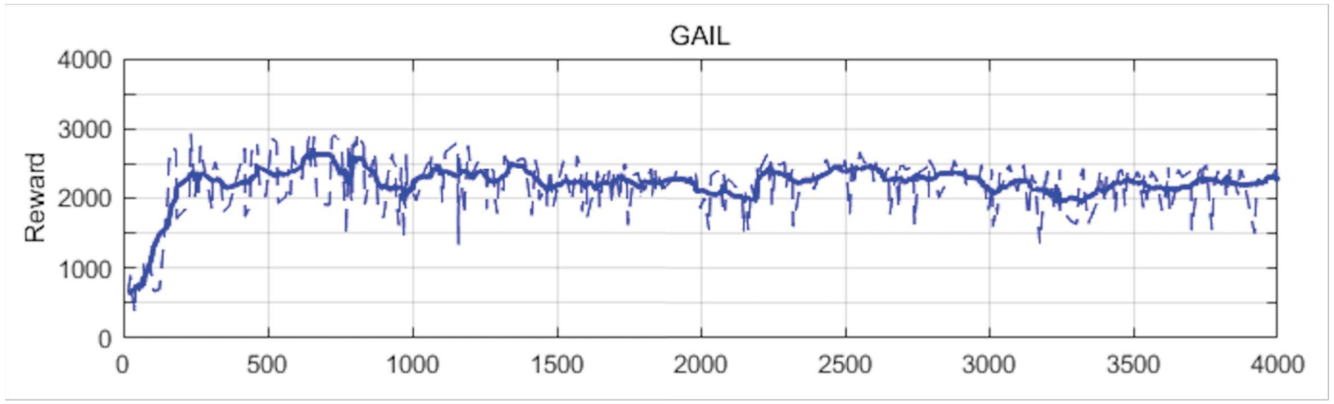
- IRL Step
  - 3 update number of vdb
- RL Step : PPO 사용
  - 10 update number of actor-critic
- Maximum iteration number : 4000
- Stop when accuracy of expert and learner from discriminator > 80%



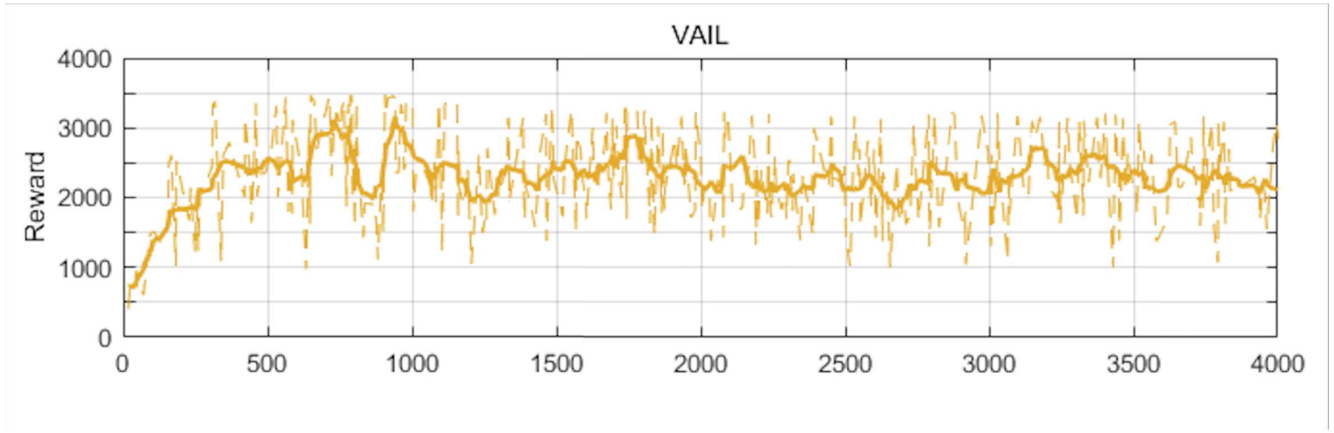
```
actor.train(), critic.train(), vdb.train()
if train_discrim_flag:
    expert_acc, learner_acc = train_vdb(vdb, memory, vdb_optim, demonstrations, 0, args)
    print("Expert: %.2f%% | Learner: %.2f%%" % (expert_acc * 100, learner_acc * 100))
    if expert_acc > args.suspend_accu_exp and learner_acc > args.suspend_accu_gen:
        train_discrim_flag = False
train_actor_critic(actor, critic, memory, actor_optim, critic_optim, args)
```

VAIL Result

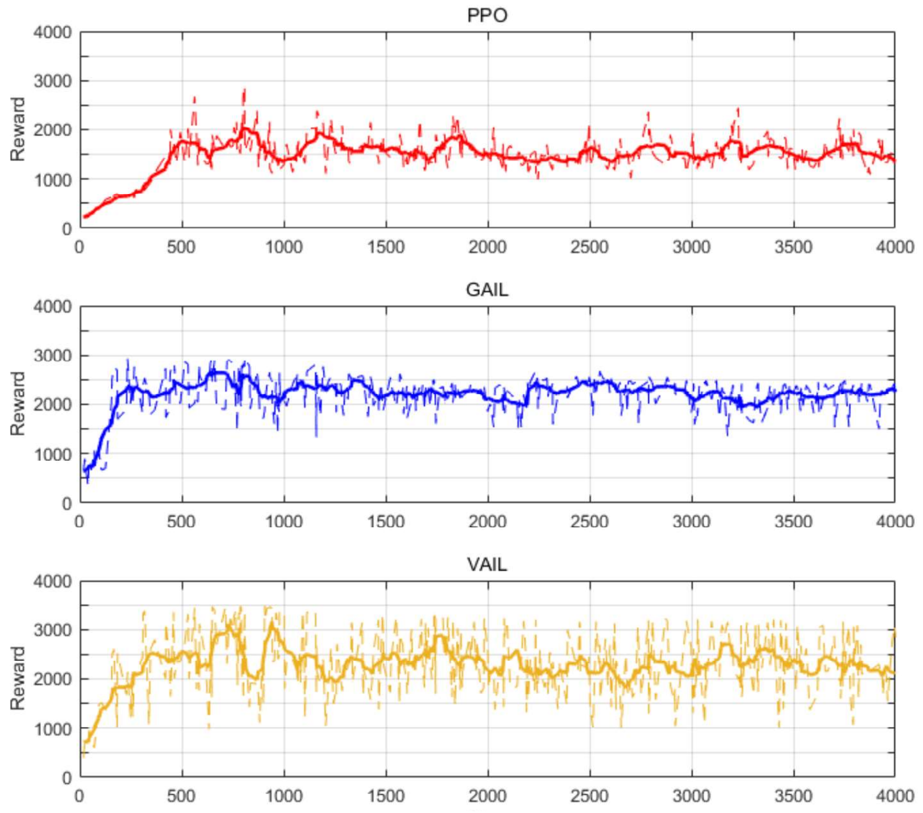
GAIL Learning curve

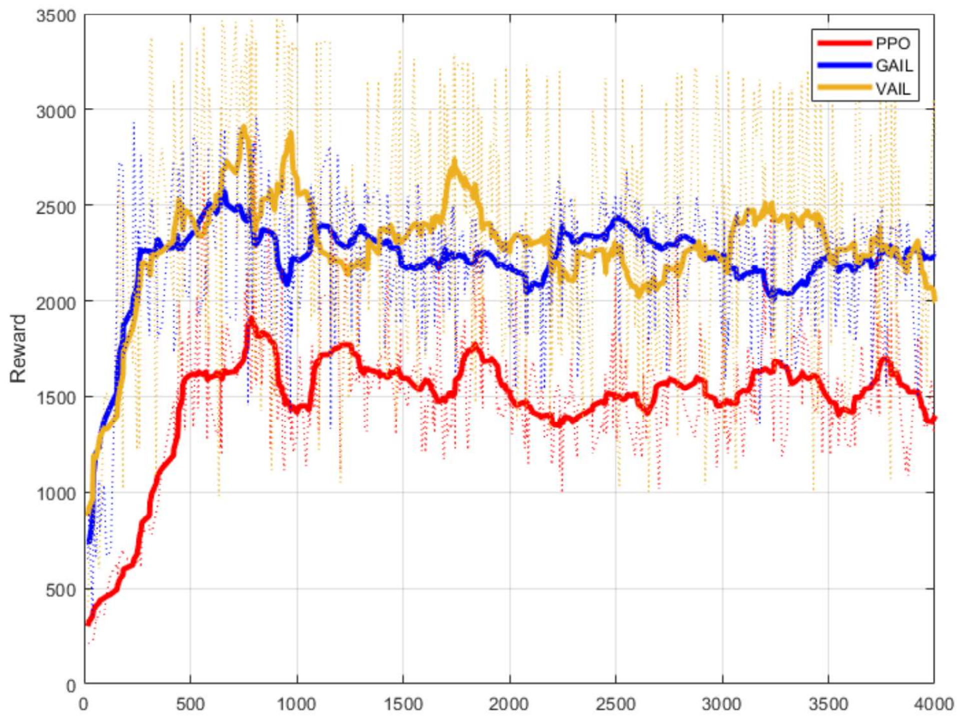


VAIL Learning curve



Total Result – PPO (to compare), GAIL, VAIL





느낀점

1) Model-free RL 쪽을 기본적으로 구현할 줄 알면 나중에 IRL 구현할 때도 훨씬 편하다.

2) Expert's demonstrations를 정말 잘 만들어야한다.

3) IRL은 결국 Optimization problem이기 때문에 최적화 기법을 공부하고 보면 편하다.

4) GAN은 정말 hyperparameter에 예민해서 조정을 많이 해야되기 때문에 힘들었다..
그래도 한번 잘 정하면 학습은 꽤나 잘 되는 것 같다.

이동민

프로젝트 매니저

윤승제

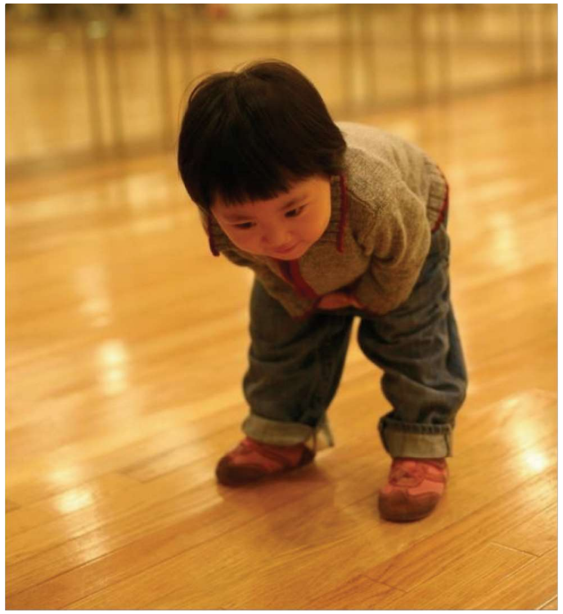
이승현

이건희

김준태

김예찬

4개월동안 모두 고생 많으셨습니다~!



Thanks for the precious advice~!!

최성준 박사님(Disney Research), 이경재님(SNU RLLab)

Thank you