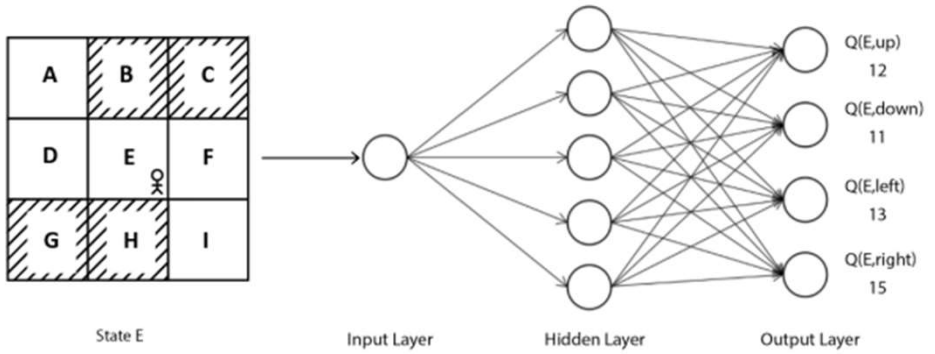
Policy Gradient Methods

Value-based RL

- In the previous chapters, we learned various methods to find the optimal policy in a given environment.
- The methods involved (iteratively) calculating Q-values of actions in each state.
- From the Q-values, we extract the optimal policy.
  - In each state, we choose an action with the highest Q value.
- These methods - using Q-values - are called "value-based" RL methods.

State	Action	Value
s_0	0	10
s_0	1	5
s_1	0	10
s_1	1	11



Disadvantages of Value-based RL Methods

- It is suitable for discrete environments
  - environments with discrete action space
- It cannot be directly applied to continuous environments
  - environments with continuous action space

- Suppose we are training an agent to drive a car and say we have one continuous action in our action space: speed.
- The speed ranges from 0km/h to 150km/h.
- In this case, how can we compute the Q value of all possible state-action pairs?

Disadvantages of Value-based RL Methods

- One way to use a Q table or a Q network is to discretize the continuous actions.
  - action 1: 0km/h to 10km/h
  - action 2: 10km/h to 20km/h
  - and so on.

- Discretization can work, but is not desirable depending on the task.
  - Granularity of action is degraded
  - may end up with huge number of actions

- Many real-world problems have continuous action space
  - self-driving cars, robots, etc.

Policy-based RL Methods

- There is another approach of reinforcement learning called policy-based RL methods.
- In policy-based RL, we do not need to compute the Q function.
- Instead, the optimal policy is computed directly.

- Policy-based methods have several advantages over value-based methods
  - One is that they support continuous action spaces.

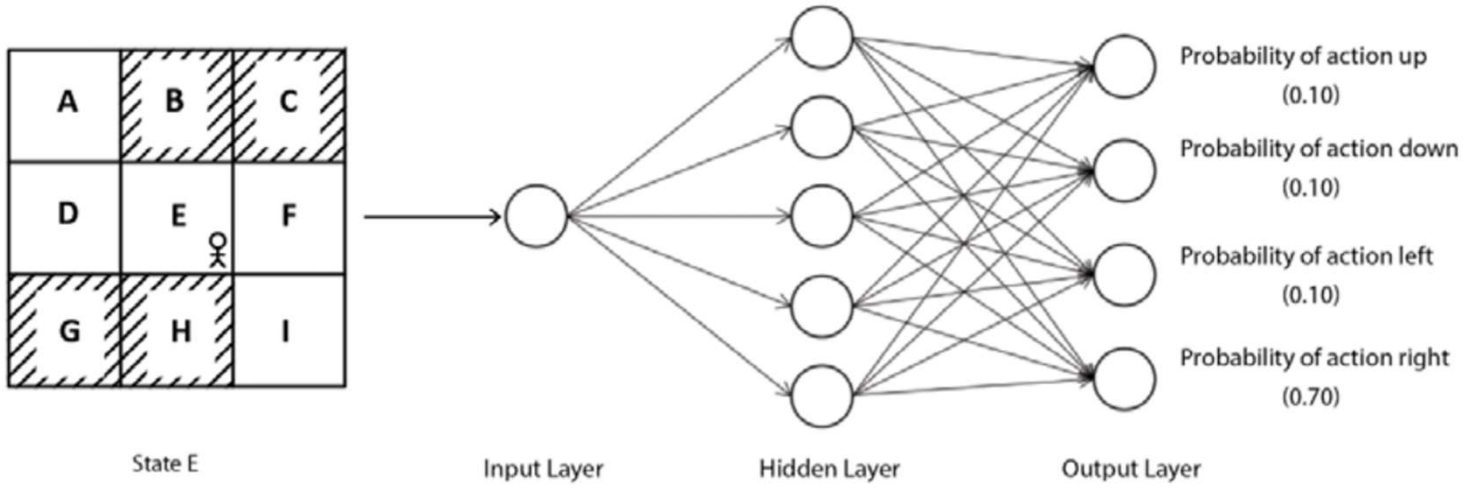
Policy Gradient: Intuition

- A policy-based method: does not compute Q function
- Directly parameterizes the policy using some parameter θ

- In a policy-based method, we use a stochastic policy.
  - In a stochastic policy, actions in a state is chosen based on a probability distribution.
  - Using a stochastic policy is the means for exploration in a policy-based method.
  - According to a stochastic policy π, the probability of choosing action a in a state s is $\pi(a|s)$.
  - Furthermore, we parameterized our policy, and denote the policy as $\pi_\theta(a|s)$.
    - The policy is parameterized by θ.

Policy Gradient: Intuition

- What does it mean to parameterize the policy?
- It means we assume there is some function that maps the state to the probability of actions in that state.
- As for the function, we can use a neural network called policy network as a function approximator.
- Parameters of the policy network is θ, which is trained using the gradient descent algorithm.



DQN vs. Policy Gradient

- In DQN, we feed the state as an input to the network, and it returns the Q values of all possible actions in that state.
- We select an action that has the maximum Q value.

- In the Policy Gradient method, we feed the state as an input to the network, and it returns the probability distribution over an action space.
- We select an action selects an action using the probability distribution.

Learning in the Policy Gradient Method

- We play an episode and store the states, actions, and rewards. This becomes our training data.

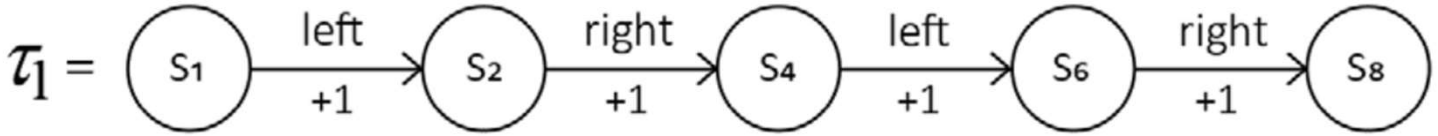
- If we get a positive return, we increase the probability of all the actions that we took in each state until the end of the episode.

- If we get a negative return, we decrease the probability of all the actions that we took in each state until the end of the episode.

Learning in the Policy Gradient Method: Example

- We have eight states s_1 to s_8, and our goal is to reach state s_8.
- The action space only consists of two actions: *left* and *right*.
- We have a policy network, which returns the probability of each actions when we feed the state.

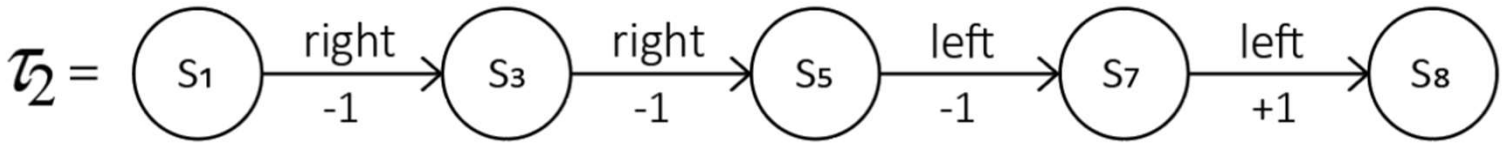
- Suppose we select an action in each state based on the probability distribution returned by the policy network.



- The return of this trajectory is $R(\tau_1) = 1 + 1 + 1 + 1 = 4$.
- Since we got a positive return, we increase the probabilities of all the actions that we took in each state until the end of the episode.
- The probabilities of $(s_1, left), (s_2, right), (s_4, left), (s_6, right)$ are increased.

Learning in the Policy Gradient Method: Example

- We generate another trajectory τ_2, where we also select actions based on the probability distribution given by the policy network.



- The return of this trajectory is $R(\tau_2) = -1 - 1 - 1 + 1 = -2$.
- Since we got a negative return, we decrease the probabilities of all the actions that we took in each state until the end of the episode.
- The probabilities of $(s_1, right), (s_3, right), (s_5, left), (s_7, left)$ are decreased.

Learning in the Policy Gradient Method

- How exactly do we increase and decrease probabilities?
- This is done using backpropagation, an algorithm used to update the parameters of the policy network.
- The updates are done in such a way that the actions yielding high return will get high probabilities, and actions yielding low return will get low probabilities.

Details of the Policy Gradient Method

- The goal of the policy gradient method is to find the optimal parameter θ of the neural network, so that the network returns the "correct" probability distribution over the action space.

- The objective of our network is to assign high probabilities to actions that maximize the expected return of the trajectory.
- The objective function can be written as:
  - $J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[R(\tau)]$
  - τ: trajectory
  - $\tau \sim \pi_\theta(\tau)$: we are sampling the trajectory based on the policy π given by the network parameterized by θ.
  - $R(\tau)$: return of the trajectory τ.
- We want to maximize the objective function.

Details of the Policy Gradient Method

- To minimize the objective function, we calculate the gradients of the objective function and update the parameter using gradient descent.

- This time, we need to maximize the objective function. So we use gradient ascent.

- $\theta = \theta + \alpha \nabla_\theta J(\theta)$

  - $\nabla_\theta J(\theta)$: gradients of the the objective function

Details of the Policy Gradient Method

- The gradient $\nabla_\theta J(\theta)$ can be derived as:
  - $\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t|s_t) R(\tau)]$

- So the gradient ascent becomes:
  - $\theta = \theta + \alpha \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t|s_t) R(\tau)$
  - $\log \pi_\theta (a_t|s_t)$: log probability of taking an action a given the state s at time t.
  - $R(\tau)$: return of the trajectory.

- We can see from the equation that when $R(\tau)$ is positive, we are increasing the probability of action a_t in state s_t. When $R(\tau)$ is negative, we are decreasing the probability of action a_t in state s_t.

Details of the Policy Gradient Method

- Since we are updating θ using each step in the episode, we can say the following:

$$\theta = \theta + \alpha \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t|s_t) R(\tau)$$

- Until now, we have omitted the expection from the equation:
  - $\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [\nabla_\theta \log \pi_\theta (a_t|s_t) R(\tau)]$

- For the expectation, we are going to follow the Monte Carlo approach and change the expectation to average over N trajectories.

$$\theta = \theta + \alpha \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t|s_t) R(\tau)$$

Details of the Policy Gradient Method

- To update the parameter, we first collect data from N trajectories following the policy π_θ.
- Then, we compute the gradient as:

$$\theta = \theta + \alpha \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t | s_t) R(\tau) \right]$$

- Then, we update the parameter as:

$$\theta = \theta + \alpha \nabla_\theta J(\theta)$$

- We repeat this step for many iterations to find the optimal policy.

Deriving the Policy Gradient

- Derivation of the following equation

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau) \right]$$

- We start from the definition of $J(\theta)$,

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[R(\tau)]$$

- When X is a continuous random variable whose probability density function (pdf) is given as $p(x)$, the expection of f(X) can be defined as:

$$\mathbb{E}_{x \sim p(x)}[f(X)] = \int_x p(x) f(x) dx$$

- Thus, from the definition of $J(\theta)$,

$$J(\theta) = \int \pi_\theta(\tau) R(\tau) d\tau$$

Deriving the Policy Gradient

- Now we apply the derivative,

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) R(\tau) d\tau$$

- multiply and divide by $\pi_\theta(\tau)$,

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) \frac{\pi_\theta(\tau)}{\pi_\theta(\tau)} R(\tau) d\tau$$

- and rearrange the equation.

$$\nabla_\theta J(\theta) = \int \pi_\theta(\tau) \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} R(\tau) d\tau$$

Deriving the Policy Gradient

- We use the log derivative trick which is:

$$\frac{d}{dx} f(\log x) = \frac{f'(x)}{f(x)}$$

- Using the trick, we get

$$\nabla_\theta J(\theta) = \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau)\, R(\tau) d\tau$$

- If we write this in the expection form:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [\nabla_\theta \log \pi_\theta(\tau)\, R(\tau)]$$

Deriving the Policy Gradient

- How to compute $\nabla_\theta \log \pi_\theta(\tau)$

- The probability distribution of trajectory can be given as:

$$\pi_\theta(\tau) = p(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

  - $p(s_0)$ is the initial state distribution.

- Taking the log on both sides, we can write:

$$\log \pi_\theta(\tau) = \log \left[p(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) \right]$$

Deriving the Policy Gradient

- Log of a product is equal to sum of the logs, thus:

$$\log \pi_\theta(\tau) = \log p(s_0) + \log \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

- Using the same rule, we can further write

$$\log \pi_\theta(\tau) = \log p(s_0) + \sum_{t=0}^{T-1} [\log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t)]$$

- Now we apply the derivative to the equation.

$$\nabla_\theta \log \pi_\theta(\tau) = \nabla_\theta \left[\log p(s_0) + \sum_{t=0}^{T-1} [\log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t)] \right]$$

Deriving the Policy Gradient

- When taking derivative, the terms $\log p(s_0)$ and $\log p(s_{t+1}|s_t, a_t)$ are not related to θ, so they become zero.

$$\nabla_\theta \log \pi_\theta(\tau) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

- Using this equation, we get the following:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau) \right]$$

The Policy Gradient Algorithm

- This algorithm is called REINFORCE, or Monte Carlo policy gradient.

1. Initialize the network parameter θ with random values
2. Generate N trajectories $\{\tau^i\}_{i=1}^{N}$ following the policy π_θ
3. Compute the return of the trajectory $R(\tau)$
4. Compute the gradients

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)\right]$$

5. Update the network parameter as $\theta = \theta + \alpha\nabla_\theta J(\theta)$
6. Repeat *steps* 2 to 5 for several iterations

Policy Gradient on Continuous Action Space

- We saw that in the policy gradient method, the policy network returns a probability distribution of actions in a state.
- Then, we select an action based on the probability distribution.
- However, this applies to a discrete action space.

- If the action space is continuous, we can have the policy network return the mean and the variance of the action as output.
- We can generate a Gaussian distribution using the mean and the variance, and select an action by sampling from the distribution.

Variance Reduction Methods

- The REINFORCE method is an on-policy method, because we improve the same policy with which we are generating episodes in every iteration.

- One of the major issue in the method is the the gradient, $\nabla_\theta J(\theta)$, will have high variance in each update.
  - The return varies greatly in each episode.

- When the variance of gradient is high, it takes a lot of time to converge.

Variance Reduction Methods

- Methods to reduce the variance
  - policy gradient with reward-to-go (causality)
  - policy gradient with baseline

Policy Gradient with Reward-To-Go

- In the original method, the policy gradient is computed as:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta\,(a_t|s_t)R(\tau)\right]$$

- We make a small change in the equation.

- Instead of using the full return of the trajectory $R(\tau)$, we use the reward-to-go R_t, which is the return of the trajectory starting from state s_t.

Policy Gradient with Reward-To-Go

- Let us consider one episode, in which the trajectory was as follows.

- The return of the trajectory is $R(\tau) = +1 - 1 + 1 + 1 = 2$.

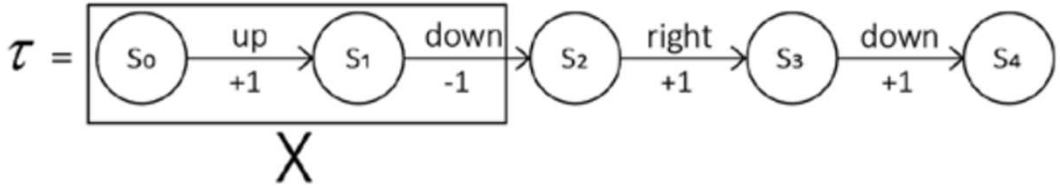
- Now, we compute the gradient as :
  - $\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta(\text{up}|s_0) R(\tau) + \nabla_\theta \log \pi_\theta(\text{down}|s_1) R(\tau) + \nabla_\theta \log \pi_\theta(\text{right}|s_2) R(\tau) + \nabla_\theta \log \pi_\theta(\text{down}|s_3) R(\tau)$

- In every step of the episode, we are multiplying the log probability of the action by the return of the full trajectory $R(\tau)$.

Policy Gradient with Reward-To-Go

- Suppose we want to know how good the action right is in the state s_2.

- If the action *right* in the state s_2 is a good action, we increase its probability. If not, then we decrease its probability.
- To find out whether the action right in the state s_2 is good, we do not need to consider the states and actions that came before s_2.



- Thus, we use the return starting from s_2, which we call reward-to-go.
  - $\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta(\text{up}|s_0)R_0 + \nabla_\theta \log \pi_\theta(\text{down}|s_1)R_1 + \nabla_\theta \log \pi_\theta(\text{right}|s_2)R_2 + \nabla_\theta \log \pi_\theta(\text{down}|s_3)R_3$

$$R_t = \sum_{t'=0}^{T-1} r(s_{t'}, a_{t'})$$

Policy Gradient with Reward-To-Go

- In policy gradient with reward-to-go, the gradient is expressed as:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta\,(a_t|s_t)R_t\right]$$

$$R_t = \sum_{t'=0}^{T-1} r(s_{t'}, a_{t'})$$

Policy Gradient with Reward-To-Go

- The algorithm

1. Initialize the network parameter θ with random values
2. Generate N number of trajectories $\{\tau^i\}_{i=1}^{N}$ following the policy π_θ
3. Compute the return (reward-to-go) R_t
4. Compute the gradients:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\, R_t\right]$$

5. Update the network parameter as $\theta = \theta + \alpha\nabla_\theta J(\theta)$
6. Repeat *steps* 2 to 5 for several iterations

Policy Gradient with Baseline

- We start from the gradient in policy gradient with reward-to-go.

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta\,(a_t|s_t)R_t\right]$$

- We introduce a new function b called a baseline function, and subtract the baseline from the reward.

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta\,(a_t|s_t)(R_t - b)\right]$$

Policy Gradient with Baseline

- The purpose of the baseline is to reduce the variance in the return.
- Thus, if the baseline b is a value that can give us the expected return from the state the agent is in, then subtracing b in every step will reduce the variance in return.

- A choice for the baseline function is:

$$b = \frac{1}{N}\sum_{i=1}^{N} R(\tau)$$

- We can use other functions as the baseline function as long as it does not depend on the network parameter θ.

Policy Gradient with Baseline

- One of the most popular function for baseline is the value function.
- The value function is the expected return an agent would obtain starting from the state following the policy π.
- Using the value function as the baseline, we can write:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta\,(a_t|s_t)(R_t - V(s_t))\right]$$

- For other baseline functions, we can also use the Q function or the advantage function.

Policy Gradient with Baseline

- Say we use the value function as our baseline function. How can we get the optimal value function?
- We approximate the value function using another neural network parameterized by ϕ. We call this the value network.

- To train the value network, we define the loss function as follows:

$$J(\phi) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\left(R_t - V_\phi(s_t)\right)^2$$

- Then, we minimize the error using gradient descent and update the network parameter as:

$$\phi = \phi + \alpha\nabla_\phi J(\phi)$$

Policy Gradient with Baseline

- The algorithm

1. Initialize the policy network parameter θ and value network parameter ϕ
2. Generate N number of trajectories $\{\tau^i\}_{i=1}^{N}$ following the policy π_θ
3. Compute the return (reward-to-go) R_t
4. Compute the policy gradient:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\ (R_t - V_\phi(s_t))\right]$$

5. Update the policy network parameter θ using gradient ascent as $\theta = \theta + \alpha\nabla_\theta J(\theta)$
6. Compute the MSE of the value network:

$$J(\phi) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1}\left(R_t - V_\phi(s_t)\right)^2$$

7. Compute gradients $\nabla_\phi J(\phi)$ and update the value network parameter ϕ using gradient descent as $\phi = \phi - \alpha\nabla_\phi J(\phi)$
8. Repeat *steps 2* to *7* for several iterations

Cart Pole Balancing with Policy Gradient

- We implement the policy gradient algorithm for the cart pole balancing task.
- The code includes all three versions:
  - policy gradient
  - policy gradient with reward-to-go
  - policy gradient with baseline
    - baseline: value function

Cart Pole Balancing with Policy Gradient

- library imports
  - namedtuple is data structure where we can record a value with its attribute name. It is used to record the (state, action, reward, logp, done) for all steps in a trajectory.
  - matplotlib.pyplot is used to draw a graph.

```
import torch
import gym
from collections import namedtuple
import matplotlib.pyplot as plt
```

Cart Pole Balancing with Policy Gradient

- PolicyNet: neural network used for policy gradient

```
class PolicyNet(torch.nn.Module):
    def __init__(self, input_size, output_size, hidden_layer_size=64):
        super(PolicyNet, self).__init__()
        self.fc1 = torch.nn.Linear(input_size, hidden_layer_size)
        self.fc2 = torch.nn.Linear(hidden_layer_size, output_size)
        self.softmax = torch.nn.Softmax(dim=0)

    def forward(self, x):
        x = torch.from_numpy(x).float()
        return self.softmax(self.fc2(torch.nn.functional.relu(self.fc1(x))))

    def get_action_and_logp(self, x):
        action_prob = self.forward(x)
        m = torch.distributions.Categorical(action_prob)
        action = m.sample()
        logp = m.log_prob(action)
        return action.item(), logp

    def act(self, x):
        action, _ = self.get_action_and_logp(x)
        return action
```

Cart Pole Balancing with Policy Gradient

- ValueNet: neural network used to calculate value function (baseline)
  - The output layer has one neuron, which indicates the value of a state given as input.

```
class ValueNet(torch.nn.Module):
    def __init__(self, input_size, hidden_layer_size=64):
        super(ValueNet, self).__init__()
        self.fc1 = torch.nn.Linear(input_size, hidden_layer_size)
        self.fc2 = torch.nn.Linear(hidden_layer_size, 1)

    def forward(self, x):
        x = torch.from_numpy(x).float()
        return self.fc2(torch.nn.functional.relu(self.fc1(x)))
```

Cart Pole Balancing with Policy Gradient

- policyGradient: runs policy gradient algorithm while running episodes.
  - collect_trajectory runs a single episode and collects (state, action, reward, logp, done) in each step. logp is the log probability of selecting the action in the state.

```
def policyGradient(env, max_num_steps=1000, gamma=0.98, lr=0.01,
                   num_traj=10, num_iter=200):
    input_size = env.observation_space.shape[0]
    output_size = env.action_space.n
    Trajectory = namedtuple('Trajectory', 'states actions rewards dones logp')

    def collect_trajectory():
        state_list = []
        action_list = []
        reward_list = []
        dones_list = []
        logp_list = []
        state = env.reset()
        done = False
        steps = 0
        while not done and steps <= max_num_steps:
            action, logp = policy.get_action_and_logp(state)
            newstate, reward, done, _ = env.step(action)
            state_list.append(state)
            action_list.append(action)
            reward_list.append(reward)
            dones_list.append(done)
            logp_list.append(logp)
            steps += 1
            state = newstate

        traj = Trajectory(states=state_list, actions=action_list,
                          rewards=reward_list, logp=logp_list, dones=dones_list)
        return traj
```

Cart Pole Balancing with Policy Gradient

- policyGradient: runs policy gradient algorithm while running episodes.
  - calc_returns calculates (discounted) R_t for each step in the episode

```
def calc_returns(rewards):
    dis_rewards = [gamma**i * r for i, r in enumerate(rewards)]
    return [sum(dis_rewards[i:]) for i in range(len(dis_rewards))]
```

  - We prepare a policy network and a value network. Both needs training, so we use the Adam optimizer to train the networks.
    - Value network is only needed when we use the baseline

```
policy = PolicyNet(input_size, output_size)
policy_optimizer = torch.optim.Adam(policy.parameters(), lr=lr)

value = ValueNet(input_size)
value_optimizer = torch.optim.Adam(value.parameters(), lr=lr)
```

Cart Pole Balancing with Policy Gradient

- loss terms for the three versions of policy gradient

```
mean_return_list = []
for it in range(num_iter):
    traj_list = [collect_trajectory() for _ in range(num_traj)]
    returns = [calc_returns(traj.rewards) for traj in traj_list]

    #====================================#
    # policy gradient with base function #
    #====================================#
    #policy_loss_terms = [-1. * traj.logp[j] * (returns[i][j] - value(traj.states[j]))
    #                     for i, traj in enumerate(traj_list) for j in range(len(traj.actions))]

    #====================================#
    # policy gradient with reward-to-go  #
    #====================================#
    #policy_loss_terms = [-1. * traj.logp[j] * (torch.Tensor([returns[i][j]]))
    #                     for i, traj in enumerate(traj_list) for j in range(len(traj.actions))]

    #====================================#
    # policy gradient                    #
    #====================================#
    policy_loss_terms = [-1. * traj.logp[j] * (torch.Tensor([returns[i][0]]))
                         for i, traj in enumerate(traj_list) for j in range(len(traj.actions))]
```

Cart Pole Balancing with Policy Gradient

- Updating parameters of neural networks through backpropagation

```
    policy_loss = 1. / num_traj * torch.cat(policy_loss_terms).sum()
    policy_optimizer.zero_grad()
    policy_loss.backward()
    policy_optimizer.step()

    value_loss_terms = [1. / len(traj.actions) * (value(traj.states[j]) - returns[i][j])**2.
                        for i, traj in enumerate(traj_list) for j in range(len(traj.actions))]
    value_loss = 1. / num_traj * torch.cat(value_loss_terms).sum()
    value_optimizer.zero_grad()
    value_loss.backward()
    value_optimizer.step()
```

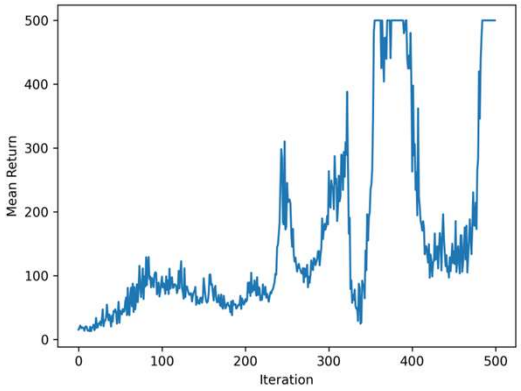
- Saving mean returns and printing intermediate results

```
        mean_return = 1. / num_traj * \
            sum([traj_returns[0] for traj_returns in returns])
        mean_return_list.append(mean_return)
        if it % 10 == 0:
            print('Iteration {}: Mean Return = {}'.format(it, mean_return))

    return policy, mean_return_list
```

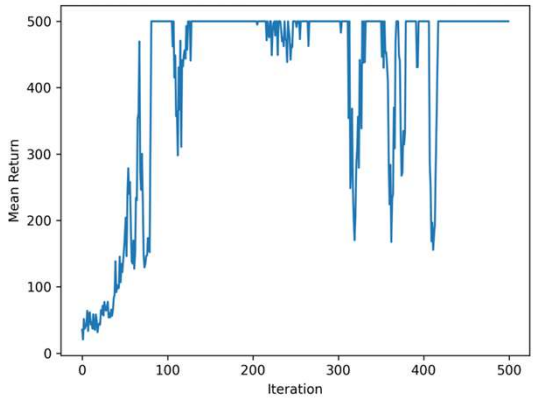
Cart Pole Balancing with Policy Gradient

- Running the policy gradient algorithm to solve the Cart Pole task
  - _max_episode_steps is maximum steps the environment allows in an episode.

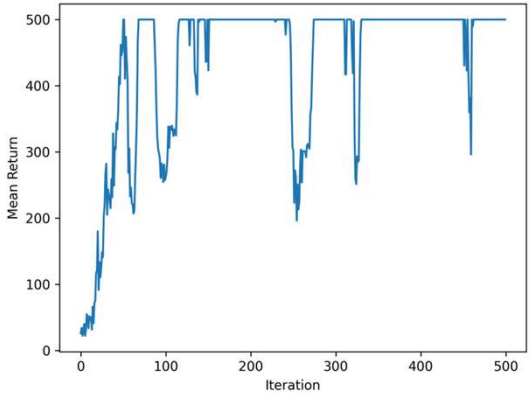
```
env = gym.make('CartPole-v1')
env._max_episode_steps=500
agent, mean_return_list = policyGradient(env, num_iter=500, max_num_steps=500, gamma=1.0, num_traj=5)

plt.plot(mean_return_list)
plt.xlabel('Iteration')
plt.ylabel('Mean Return')
plt.savefig('pg_returns.png', format='png', dpi=300)
```



policy gradient



policy gradient with reward-to-go



policy gradient with baseline

Cart Pole Balancing with Policy Gradient

- Summary
  - While DQN is a value-based method, policy gradient is a policy-based RL method.
  - In policy gradient, we do not calculate Q values. Instead, the neural network outpus probability distribution of actions in a state.
  - To train the policy network, we run episodes with stochastic policy. If the agent obtains a positive reward in an episode, probability of the actions taken is increased. If the agent obtains a negative reward, probability of the actions is decreased.
  - Since the original policy gradient method can lead to high variance of gradients, which can make the training take long time to converge.
  - We can use two improved algorithms: policy gradient with reward-to-go and policy gradient with baseline.
  - In policy gradient with baseline, we use another neural network to approximate value function, and use it as the baseline.

End of Class

Questions?

Email: jso1@sogang.ac.kr