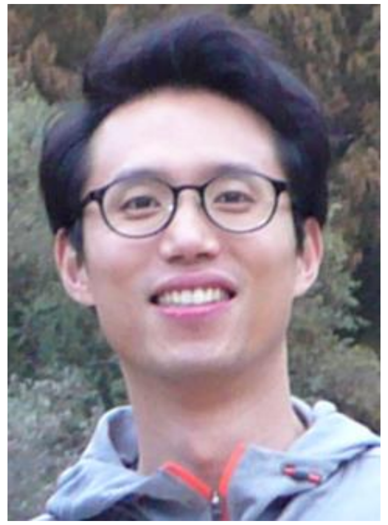
Pattern Recognition and Machine Intelligence

HANKUK UNIVERSITY OF FOREIGN STUDIES

REINFORCE ALGORITHM



Prof. Jae Young Choi

Pattern Recognition and Machine Intelligence Lab. (PMI)

Hankuk University of Foreign Studies

REINFORCE-THEORY

❖ Key Idea

- During learning, actions that resulted in good outcomes should become more probable
- These actions are positively reinforced
- Action probabilities are changed by following the policy gradient
- REINFORCE is known as policy gradient algorithm
- **Policy gradient** is used to modify the policy parameters to maximize the objective

REINFORCE-THEORY

❖ Objective

- Return

$$R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r_t'$$

$$\tau = s_0, a_0, r_0, \ldots, s_T, a_T, r_T$$

The return of a trajectory $R_t(\tau)$ is discounted sum of rewards from time step *t* to the end of a trajectory

REINFORCE-THEORY

❖ Objective

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t r_t\right]$$

The objective $J(\pi_\theta)$ is the expected return over all complete trajectories generated by an agent

Expectation is calculated over many trajectories sampled from a policy, $\tau \sim \pi_\theta$

REINFORCE-THEORY

❖ The Policy π_θ and Objective $J(\pi_\theta)$

- The policy provides a way for an agent to act
- The objective provides a target to maximize

❖ Solving the following problem

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

❖ To maximize the objective, we perform gradient ascent

$$\theta \leftarrow \theta + \alpha \underline{\nabla_\theta J(\pi_\theta)}$$

Compute gradient and use it to update the parameters

REINFORCE-THEORY

❖ The policy gradient $\nabla_\theta J(\pi_\theta)$

The return for trajectory $\tau \sim \pi_\theta$

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \boxed{R_t(\tau)} \nabla_\theta \log \underline{\pi_\theta(a_t \mid s_t)}\right]$$

The probability of the action taken by the agent at time step 't'

- The action is sampled from the policy $a_t \sim \pi_\theta(s_t)$
- The statement of policy gradient
  → Expected sum of the gradients of the log probabilities of the action a_t multiplied by the corresponding return $R_t(\tau)$

REINFORCE-THEORY

❖ Interpreting policy gradient

The return for trajectory $\tau \sim \pi_\theta$

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \boxed{R_t(\tau)} \nabla_\theta \log \underline{\pi_\theta(a_t \mid s_t)}\right]$$

The probability of the action taken by the agent at time step 't'

- If the return $R_t(\tau) > 0$, then the probability of the action $\pi_\theta(a_t \mid s_t)$ is increased.
- If the return $R_t(\tau) < 0$, then the probability of the action $\pi_\theta(a_t \mid s_t)$ is decreased.
- Over the course of many updates, the policy will learn to produce actions which result in high $R_t(\tau)$

REINFORCE-THEORY

❖ Policy Gradient Derivation

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

- We cannot differentiate $R(\tau) = \sum_{t=0}^{T} \gamma^t r_t$ with respect to θ
  → Rewards r_t are generated by an unknown reward function $\mathcal{R}(s_t, a_t, s_{t+1})$
- The only way is by changing the state and action distributions which, in turn, change the rewards received by an agent
  → Transform $\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$ into a form where we can take a gradient with respect to θ

REINFORCE-THEORY

❖ Policy Gradient Derivation

$$\nabla_\theta \overbrace{\mathbb{E}_{x \sim \underbrace{p(x|\theta)}_{\text{Parameterized probability distribution}}} [\underbrace{f(x)}_{\text{Function}}]}^{\text{Expectation of function f(x)}}$$

$$= \nabla_\theta \int dx\; f(x) p(x \,|\, \theta) \qquad (\textit{definition of expectation})$$

$$= \int dx\; \nabla_\theta \big(p(x \,|\, \theta) f(x)\big) \qquad (\textit{bring in } \nabla_\theta)$$

$$= \int dx\; \big(f(x) \nabla_\theta p(x \,|\, \theta) + p(x \,|\, \theta) \nabla_\theta f(x)\big) \qquad (\textit{chain-rule})$$

$$= \int dx\; f(x) \nabla_\theta p(x \,|\, \theta) \qquad (\nabla_\theta f(x) = 0)$$

$$= \int dx\; f(x) p(x \,|\, \theta) \frac{\nabla_\theta p(x \,|\, \theta)}{p(x \,|\, \theta)} \qquad \left(\textit{multiply } \frac{p(x \,|\, \theta)}{p(x \,|\, \theta)}\right)$$

$$= \int dx\; f(x) p(x \,|\, \theta) \nabla_\theta \log p(x \,|\, \theta) \qquad \boxed{\nabla_\theta \log p(x \,|\, \theta) = \frac{\nabla_\theta p(x \,|\, \theta)}{p(x \,|\, \theta)}}$$

$$= \mathbb{E}_x [f(x) \nabla_\theta \log p(x \,|\, \theta)] \qquad (\textit{definition of expectation})$$

- f(x) is a black-box function which cannot be integrated
- To deal with it, we convert the equation into an expectation
  → It can be estimated through sampling

REINFORCE-THEORY

❖ Policy Gradient Derivation

$$\nabla_\theta \mathbb{E}_{x \sim p(x|\theta)}[f(x)] = \mathbb{E}_x[f(x)\nabla_\theta \log p(x \mid \theta)]$$

Gradient of an expectation
= Expectation of the gradient of log probability multiplied by the original function

REINFORCE-THEORY

❖ Policy Gradient Derivation

By replacing $x = \tau, f(x) = R(\tau), p(x \mid \theta) = p(\tau \mid \theta)$

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

❖ Policy Gradient Derivation

By replacing $x = \tau, f(x) = R(\tau), p(x \mid \theta) = p(\tau \mid \theta)$

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

The term $p(\tau \mid \theta)$ should be related to the policy $\pi_\theta(a_t \mid s_t)$ (we can control over)

It needs to be expanded further in the next slides

REINFORCE-THEORY

❖ Policy Gradient Derivation

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

$$p(\tau \mid \theta) = \prod_{t \geq 0} p(s_{t+1} \mid s_t, a_t)\pi_\theta(a_t \mid s_t)$$

$$\log p(\tau \mid \theta) = \log \prod_{t \geq 0} p(s_{t+1} \mid s_t, a_t)\pi_\theta(a_t \mid s_t)$$

$$\log p(\tau \mid \theta) = \sum_{t \geq 0} \left(\log p(s_{t+1} \mid s_t, a_t) + \log \pi_\theta(a_t \mid s_t)\right)$$

$$\nabla_\theta \log p(\tau \mid \theta) = \nabla_\theta \sum_{t \geq 0} \left(\boxed{\log p(s_{t+1} \mid s_t, a_t)} + \log \pi_\theta(a_t \mid s_t)\right)$$

$\boxed{\log p(s_{t+1} \mid s_t, a_t) \text{ is independent of } \theta}$, its gradient is zero

$$\nabla_\theta \log p(\tau \mid \theta) = \nabla_\theta \sum_{t \geq 0} \log \pi_\theta(a_t \mid s_t)$$

REINFORCE-THEORY

❖ Policy Gradient Derivation

- Final formation

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

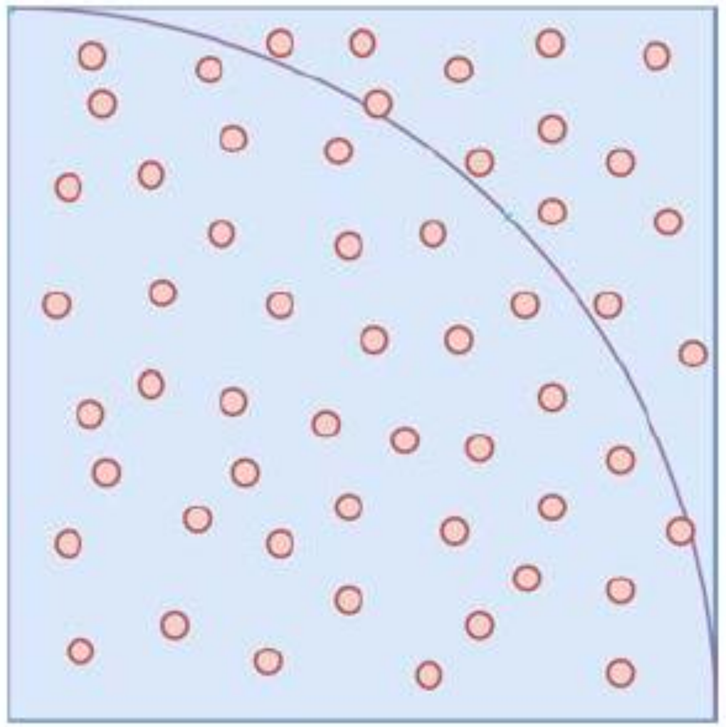
Expressing the probability $p(\tau \mid \theta)$ in terms of $\pi_\theta(a_t \mid s_t)$

$\nabla_\theta J(\pi_\theta)$ can be estimated easily using a policy network π_θ with gradient computation

REINFORCE-THEORY

❖ Implementation of policy gradient

- The REINFORCE algorithm numerically estimates the policy gradient using Monte Carlo sampling
- Use random sampling to generate data used to approximate a function
  → "approximation with random sampling"



$$\frac{\text{area of circle}}{\text{area of square}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

$$\pi = \frac{\text{No.of samples in a circle}}{\text{No.of total samples}}$$

By iteratively sampling more points and updating the ratio, our estimation will get closer to the precise value

Monte Carlo sampling used to estimate π

REINFORCE-THEORY

❖ Implementation of policy gradient

$$\nabla_\theta J(\pi_\theta) = \boxed{\mathbb{E}_{\tau \sim \pi_\theta}} \left[\sum_{t=0}^{T} R_t(\tau) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right]$$

Expectation means that more trajectories τs are sampled using a policy and averaged, it approaches the actual policy gradient $\nabla_\theta J(\pi_\theta)$

- Monte Carlo estimate over sampled trajectories

$$\nabla_\theta J(\pi_\theta) \approx \sum_{t=0}^{T} R_t(\tau) \nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

UDERSTANDING REINFORCE

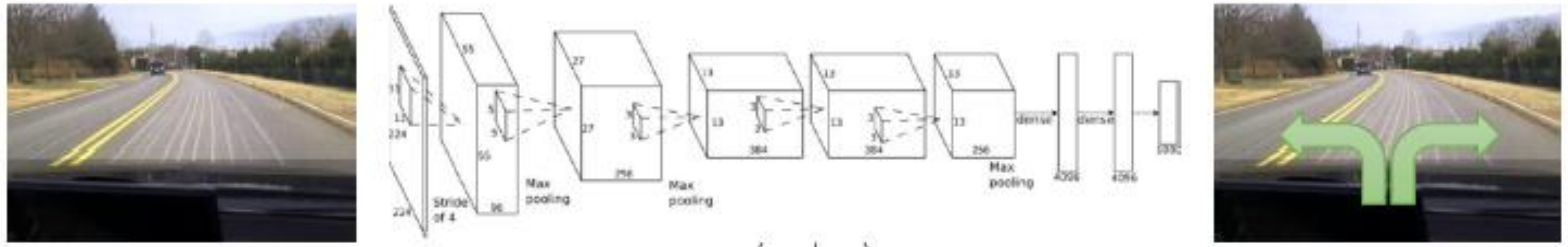
❖ Evaluating the policy gradient

$$\text{recall: } J(\theta) = E_{\tau \sim p_\theta(\tau)}\left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t)\right] \approx \frac{1}{N}\sum_i \sum_t r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$$

$$\nabla_\theta J(\theta) = E_{\tau \sim p_\theta(\tau)}\left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t)\right)\right]$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})\right)$$

what is this?



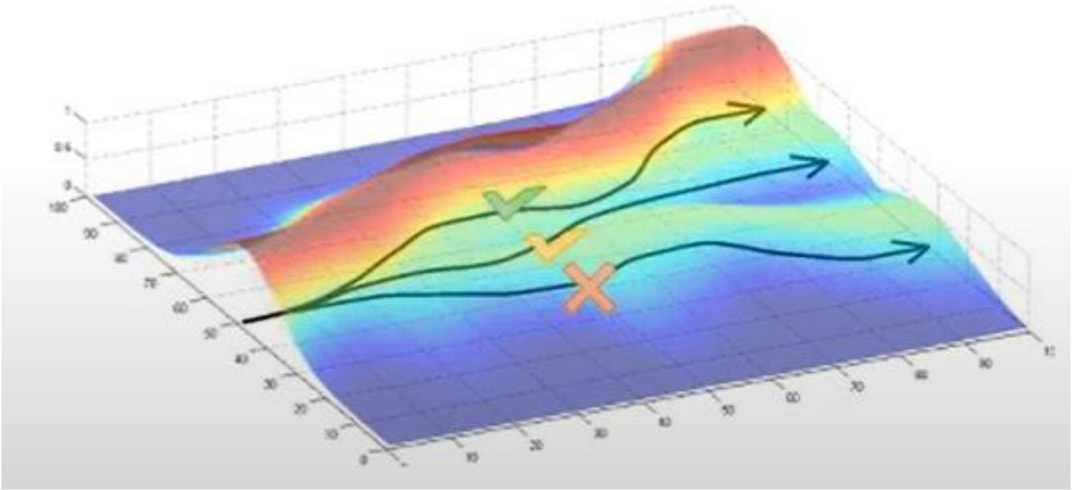
$\mathbf{s}_t$ $\qquad$ $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ $\qquad$ $\mathbf{a}_t$

UDERSTANDING REINFORCE

❖ What did we just do?

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t},\mathbf{a}_{i,t})\right)$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\underbrace{\nabla_\theta \log \pi_\theta(\tau_i)}_{\sum_{t=1}^{T}\nabla_\theta \log_\theta \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})} r(\tau_i)$$



good stuff is made more likely

bad stuff is made less likely

simply formalizes the notion of "trial and error"!

REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run it on the robot)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i,\mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

REINFORCE SUGGESTED READING

- Classic papers
  - Williams (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning: introduces REINFORCE algorithm
  - Baxter & Bartlett (2001). Infinite-horizon policy-gradient estimation: temporally decomposed policy gradient (not the first paper on this! see actor-critic section later)
  - Peters & Schaal (2008). Reinforcement learning of motor skills with policy gradients: very accessible overview of optimal baselines and natural gradient
- Deep reinforcement learning policy gradient papers
  - Levine & Koltun (2013). Guided policy search: deep RL with importance sampled policy gradient (unrelated to later discussion of guided policy search)
  - Schulman, L., Moritz, Jordan, Abbeel (2015). Trust region policy optimization: deep RL with natural policy gradient and adaptive step size
  - Schulman, Wolski, Dhariwal, Radford, Klimov (2017). Proximal policy optimization algorithms: deep RL with importance sampled policy gradient

REINFORCE-IMPLEMENTATION

Algorithm 2.1 REINFORCE algorithm

Initialize learning rate α

Initialize weights θ of a policy network π_θ

for $episode = 0, \ldots, MAX\_EPISODE$ **do**

  Sample a trajectory $\tau = s_0, a_0, r_0, \ldots, s_T, a_T, r_T$ → Use the policy network π_θ to generate a trajectory for an episode

  Set $\nabla_\theta J(\pi_\theta) = 0$

  for $t = 0, \ldots, T$ **do**

    $R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r'_t$ → For each time step 't' in the trajectory, compute the return $R_t(\tau)$

    $\nabla_\theta J(\pi_\theta) = \nabla_\theta J(\pi_\theta) + R_t(\tau) \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ → Use $R_t(\tau)$ to estimate the policy gradient and sum policy gradients for all time steps

  end for

  $\theta = \theta + \alpha \nabla_\theta J(\pi_\theta)$ → Update the policy network parameters θ

end for

Note: A trajectory is discarded after each parameter update – it cannot be reused
→ REINFORCE is an on-policy algorithm

REINFORCE-IMPLEMENTATION

❖ REINFORCE Training Loop

Code 2.6 REINFORCE implementation: training method

```
# slm_lab/agent/algorithm/reinforce.py

class Reinforce(Algorithm):
    ...

    @lab_api
    def sample(self):
        batch = self.body.memory.sample()
        batch = util.to_torch_batch(batch, self.net.device,
        ↪  self.body.memory.is_episodic)
        return batch

    @lab_api
    def train(self):
        ...
        clock = self.body.env.clock
        if self.to_train == 1:
            batch = self.sample()
            ...
            pdparams = self.calc_pdparam_batch(batch)
            advs = self.calc_ret_advs(batch)
            loss = self.calc_policy_loss(batch, pdparams, advs)
            self.net.train_step(loss, self.optim, self.lr_scheduler,
            ↪  clock=clock, global_net=self.global_net)
            # reset
            self.to_train = 0
            return loss.item()
        else:
            return np.nan
```

This code makes one parameter update to the policy network using a batch of collected trajectories

The trajectories are obtained from the agent's memory by calling sample

Computing the action probability distribution

Computing the returns used to calculate the policy loss

Updating the policy network parameters using the loss

REINFORCE-IMPLEMENTATION

❖ On-Policy Replay Memory

- REINFORCE is an on-policy algorithm
- Trajectories sampled by the algorithm should be discarded in a Memory class for learning, then deleted after each training step

- For implementation, the following API methods are contained

1. `reset` clears and resets the memory class variables.
2. `update` adds an experience to the memory.
3. `sample` samples a batch of data for training.

REINFORCE-IMPLEMENTATION

❖ On-Policy Replay Memory : Memory Initialization and Reset

Code 2.7 OnPolicyReplay: reset

```
# slm_lab/agent/memory/onpolicy.py

class OnPolicyReplay(Memory):
    ...

    def __init__(self, memory_spec, body):
        super().__init__(memory_spec, body)
        # NOTE for OnPolicy replay, frequency = episode; for other classes
        ↪  below frequency = frames
        util.set_attr(self, self.body.agent.agent_spec['algorithm'],
        ↪  ['training_frequency'])
        # Don't want total experiences reset when memory is
        self.is_episodic = True
        self.size = 0  # total experiences stored
        self.seen_size = 0  # total experiences seen cumulatively
        # declare what data keys to store
        self.data_keys = ['states', 'actions', 'rewards', 'next_states',
        ↪  'dones']
        self.reset()

    @lab_api
    def reset(self):
        '''Resets the memory. Also used to initialize memory vars'''
        for k in self.data_keys:
            setattr(self, k, [])
        self.cur_epi_data = {k: [] for k in self.data_keys}
        self.most_recent = (None,) * len(self.data_keys)
        self.size = 0
```

- “reset” in Code 2.7 is used to clear the memory after each training step
- This is specific to an on-policy memory because trajectories cannot be reused for later training

Individual episodes are constructed by storing experiences in the current episode data dictionary “self.cur_epi_data”

❖ On-Policy Replay Memory : Memory Update

Code 2.8 OnPolicyReplay: add experience

```
# slm_lab/agent/memory/onpolicy.py

class OnPolicyReplay(Memory):
    ...

    @lab_api
    def update(self, state, action, reward, next_state, done):
        '''Interface method to update memory'''
        self.add_experience(state, action, reward, next_state, done)

    def add_experience(self, state, action, reward, next_state, done):
        '''Interface helper method for update() to add experience to memory'''
        self.most_recent = (state, action, reward, next_state, done)
        for idx, k in enumerate(self.data_keys):
            self.cur_epi_data[k].append(self.most_recent[idx])
        # If episode ended, add to memory and clear cur_epi_data
        if util.epi_done(done):
            for k in self.data_keys:
                getattr(self, k).append(self.cur_epi_data[k])
            self.cur_epi_data = {k: [] for k in self.data_keys}
            # If agent has collected the desired number of episodes, it is ready to train
            # length is num of epis due to nested structure
            if len(self.states) == self.body.agent.algorithm.training_frequency:
                self.body.agent.algorithm.to_train = 1
        # Track memory size and num experiences
        self.size += 1
        self.seen_size += 1
```

Add the experience to the current episode

Check if the episode is finished. This is given by the “done” variable which will be ‘1’ if the episode is finished, ‘0’ otherwise

If the episode is finished, add the entire set of experiences for the episode to the main containers in the memory class

If the episode is finished, clear the current episode dictionary so that memory class is ready to store the next episode

If the desired number of episodes has been collected, set the ‘train flag’ to ‘1’ in the agent. This signals that the agent should train this time step

REINFORCE-IMPLEMENTATION

❖ On-Policy Replay Memory : Memory Sample

Code 2.9 OnPolicyReplay: sample

```
# slm_lab/agent/memory/onpolicy.py

class OnPolicyReplay(Memory):

    def sample(self):
        batch = {k: getattr(self, k) for k in self.data_keys}
        self.reset()
        return batch
```

Returning all of the completed episodes packaged into a batch dictionary

Reset the memory as the stored experiences will no longer be valid once the agent has completed a training step

Assignment-Implementation of REINFORCE

- Implementing Code 2.1 and Code 2.10 in the textbook
  - Analyzing and interpreting codes line by line
  - Draw graph in Figure 2.2 (textbook) and explaining the results
- Reproducing the results in Code 2.12, Code 2.13, and Figure 2.3
  - Draw graphs in Figure 2.3 (textbook) and discussing the results
- Reproducing the results in Code 2.14, Code 2.15, and Figure 2.4
  - Draw graphs in Figure 2.4 (textbook) and discussing the results

Assignment-Implementation of REINFORCE

- ❖ Please submit the report (word format)
- ❖ Please submit all the codes from your implementation including
  - Main training codes
  - Reproduction codes of experimental results
  - Execution codes