RL Adventure

PER and NoisyNet

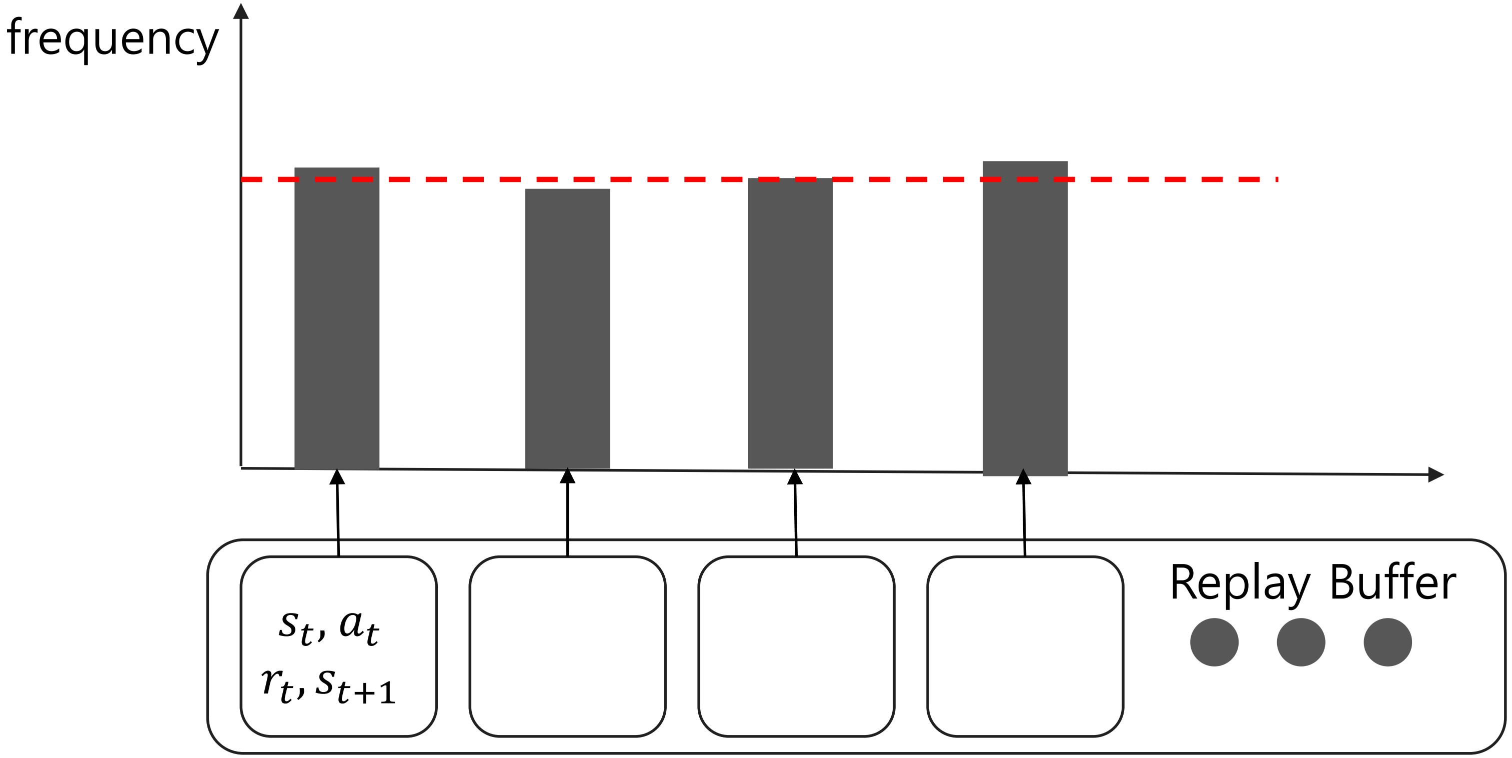
양홍선

PYCON KOREA 2018

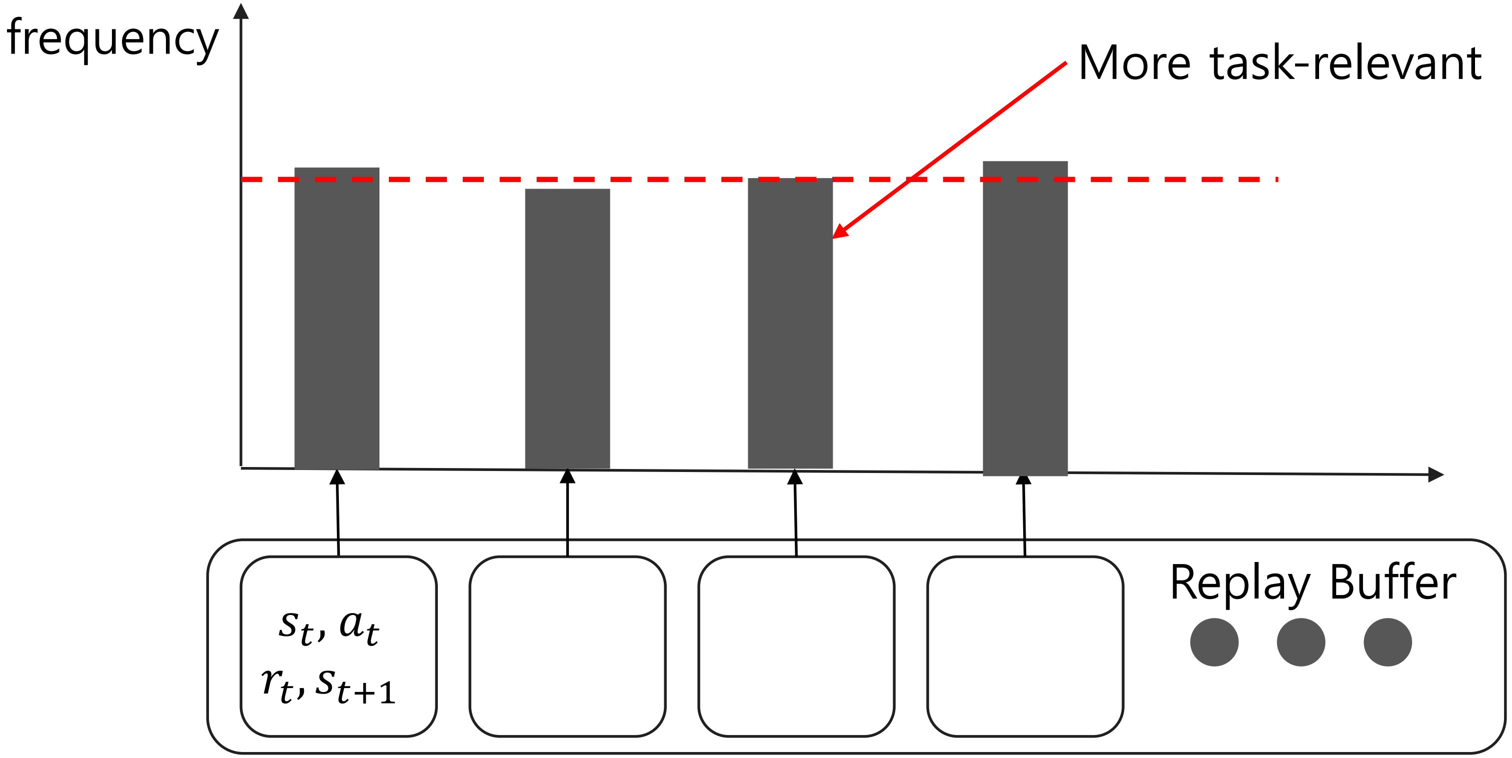
PER

Prioritized **E**xperience **R**eplay

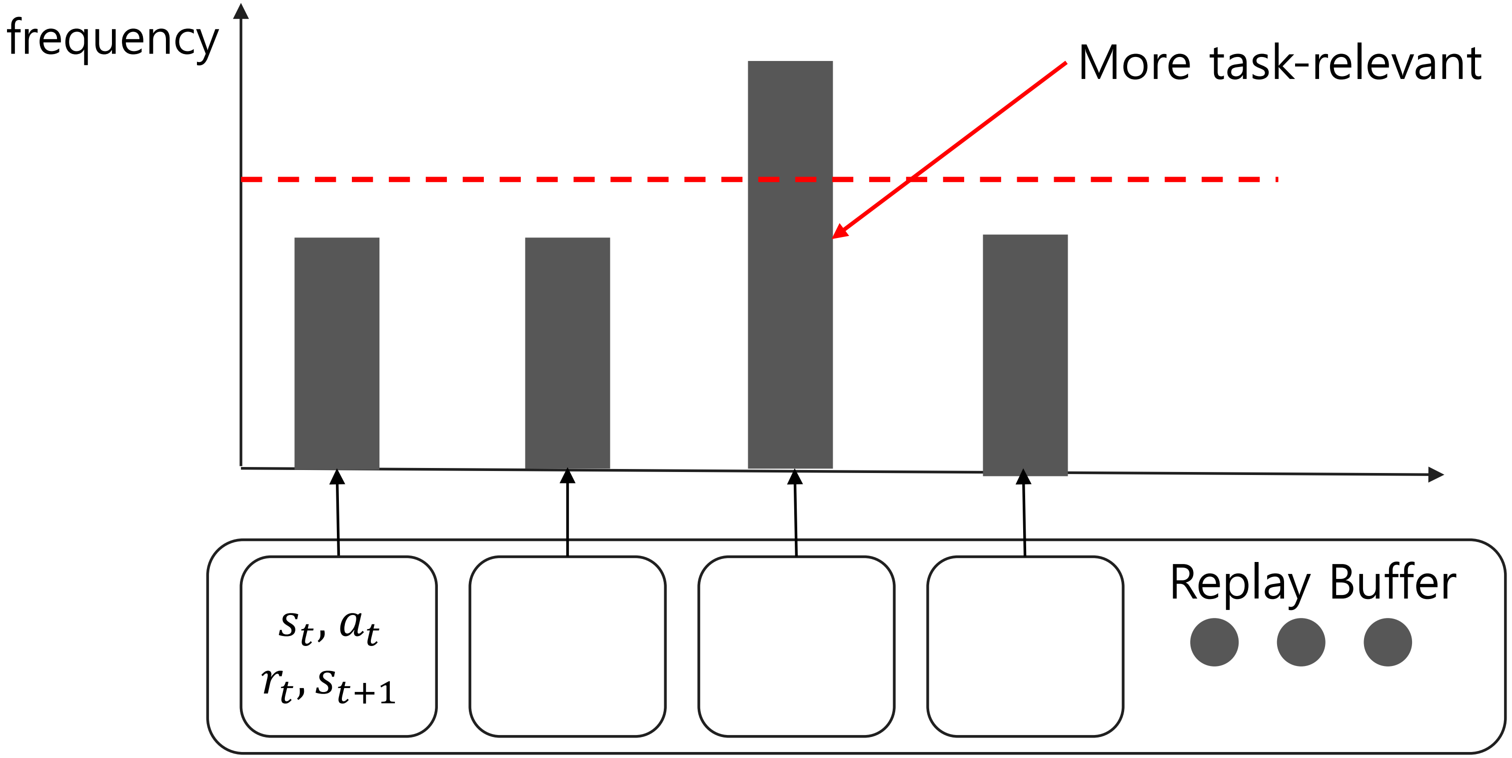
Replay Memory



Replay Memory



Replay Memory



Design of Replay Memory

Which experiences to store

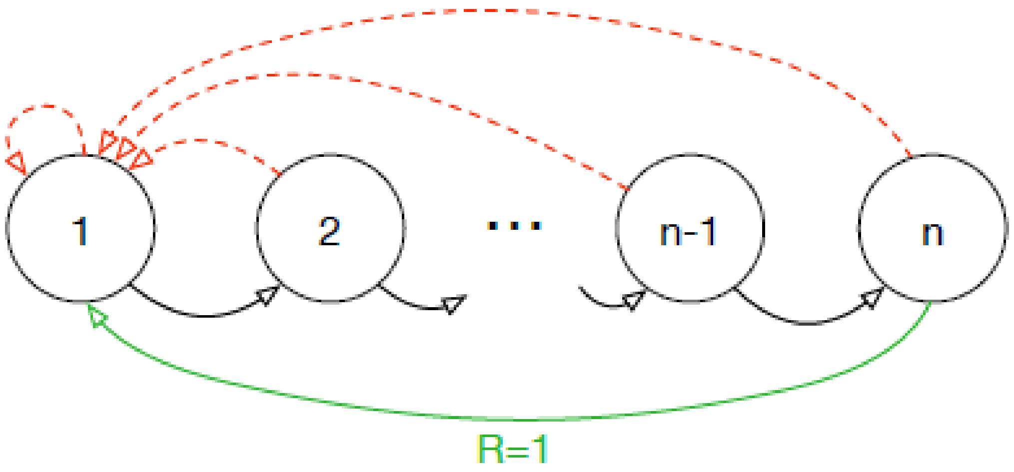
Which experiences to replay

Design of Replay Memory

Which experiences to store

Which experiences to replay

A Motivating Example



Two actions: ‘right(→→)’ and ‘wrong(→)’

The environment requires an exponential number of random steps until the first non-zero reward

The most relevant transitions are hidden in a mass of highly redundant failure cases

How?

Prioritizing with **TD-Error**

A transition's TD error δ
→how 'surprising' or unexpected the transition is

Weakness

A low TD-Error on first visit may not be replayed for a long time

The PER with TD-Error is sensitive to noise spikes

Greedy prioritization focuses on a small subset of the experience

Stochastic Sampling!

Stochastic Prioritization

Proportional prioritization

- $p_i = |\delta_i| + \epsilon$
- $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$
    - $p_i > 0$: the priority of transition i
    - α: determines how much prioritization is used
- Sum-tree

Stochastic Prioritization

Rank-based prioritization

- $p_i = \frac{1}{\mathrm{rank}(i)}$
  - $\mathrm{rank}(i)$ is the rank of transition i when the replay memory is sorted according to $|\delta_i|$
  - More robust
- Binary heap

Annealing the Bias

- Importance-Sampling (IS) weights
  - $w_i = \left(\frac{1}{N} \frac{1}{P(i)}\right)^{\beta}$
    - Normalize: $\frac{1}{\max_i w_i}$
  - $\Delta \leftarrow \Delta + \boxed{w_i \cdot \delta_i} \cdot \nabla_{\theta} Q(S_{i-1}, A_{i-1})$

Proportional prioritization (without sum-tree)

```
class NaivePrioritizedBuffer(object):
    def __init__(self, capacity, prob_alpha=0.6):


    def push(self, state, action, reward, next_state, done):


    def sample(self, batch_size, beta=0.4):


    def update_priorities(self, batch_indices, batch_priorities):


    def __len__(self):

```

```python
class NaivePrioritizedBuffer(object):
    def __init__(self, capacity, prob_al

    def push(self, state, action, reward,

    def sample(self, batch_size, beta=0.4

    def update_priorities(self, batch_ind

    def __len__(self):
```

```python
    def __init__(self, capacity, prob_alpha=0.6):
        self.prob_alpha = prob_alpha
        self.capacity   = capacity
        self.buffer     = []
        self.pos        = 0
        self.priorities = np.zeros((capacity,), dtype=np.float32)
```

```python
class NaivePrioritizedBuffer(object):
    def __init__(self, capacity, prob_al

    def push(self, state, action, reward,

    def sample(self, batch_size, beta=0.4

    def update_priorities(self, batch_inc

    def __len__(self):
```

$$P(i) = \frac{p_i^{\alpha}}{\sum_k p_k^{\alpha}}$$

```python
    def __init__(self, capacity, prob_alpha=0.6):
        self.prob_alpha = prob_alpha
        self.capacity   = capacity
        self.buffer     = []
        self.pos        = 0
        self.priorities = np.zeros((capacity,), dtype=np.float32)
```

```python
class NaivePrioritizedBuffer(object):
    def __init__(self, capacity, prob_alp

    def push(self, state, action, reward,

    def sample(self, batch_size, beta=0.4

    def update_priorities(self, batch_ind

    def __len__(self):
```

```python
    def push(self, state, action, reward, next_state, done):
        assert state.ndim == next_state.ndim
        state      = np.expand_dims(state, 0)
        next_state = np.expand_dims(next_state, 0)

        max_prio = self.priorities.max() if self.buffer else 1.0

        if len(self.buffer) < self.capacity:
            self.buffer.append((state, action, reward, next_state, done))
        else:
            self.buffer[self.pos] = (state, action, reward, next_state, done)

        self.priorities[self.pos] = max_prio
        self.pos = (self.pos + 1) % self.capacity
```

```
class NaivePrioritizedBuffer(object):
    def __init__(self, capacity, prob_alp

    def push(self, state, action, reward

    def sample(self, batch_size, beta=0.4

    def update_priorities(self, batch_ind

    def __len__(self):
```

```
def push(self, state, action, reward, next_state, done):
    assert state.ndim == next_state.ndim
    state      = np.expand_dims(state, 0)
    next_state = np.expand_dims(next_state, 0)

    max_prio = self.priorities.max() if self.buffer else 1.0

    if len(self.buffer) < self.capacity:
        self.buffer.append((state, action, reward, next_state, done))
    else:
        self.buffer[self.pos] = (state, action, reward, next_state, done)

    self.priorities[self.pos] = max_prio
    self.pos = (self.pos + 1) % self.capacity
```

최소 한번은 replay

```
class NaivePrioritizedBuffer(object):
    def __init__(self, capacity, prob_alp

    def push(self, state, action, reward,

    def sample(self, batch_size, beta=0.4

    def update_priorities(self, batch_inc

    def __len__(self):
```

```
    def sample(self, batch_size, beta=0.4):
        if len(self.buffer) == self.capacity:
            prios = self.priorities
        else:
            prios = self.priorities[:self.pos]

        probs  = prios ** self.prob_alpha
        probs /= probs.sum()

        indices = np.random.choice(len(self.buffer), batch_size, p=probs)
        samples = [self.buffer[idx] for idx in indices]

        total    = len(self.buffer)
        weights  = (total * probs[indices]) ** (-beta)
        weights /= weights.max()
        weights  = np.array(weights, dtype=np.float32)

        batch       = list(zip(*samples))
        states      = np.concatenate(batch[0])
        actions     = batch[1]
        rewards     = batch[2]
        next_states = np.concatenate(batch[3])
        dones       = batch[4]

        return states, actions, rewards, next_states, dones, indices, weights
```

```
class NaivePrioritizedBuffer(object):
    def __init__(self, capacity, prob_alp

    def push(self, state, action, reward,

    def sample(self, batch_size, beta=0.4

    def update_priorities(self, batch_ind

    def __len__(self):
```

```
    def sample(self, batch_size, beta=0.4):
        if len(self.buffer) == self.capacity:
            prios = self.priorities
        else:
            prios = self.priorities[:self.pos]

        probs  = prios ** self.prob_alpha
        probs /= probs.sum()

        indices = np.random.choice(len(self.buffer), batch_size, p=probs)
        samples = [self.buffer[idx] for idx in indices]

        total    = len(self.buffer)
        weights  = (total * probs[indices]) ** (-beta)
        weights /= weights.max()
        weights  = np.array(weights, dtype=np.float32)

        batch       = list(zip(*samples))
        states      = np.concatenate(batch[0])
        actions     = batch[1]
        rewards     = batch[2]
        next_states = np.concatenate(batch[3])
        dones       = batch[4]

        return states, actions, rewards, next_states, dones, indices, weights
```

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

```python
class NaivePrioritizedBuffer(object):
    def __init__(self, capacity, prob_alp

    def push(self, state, action, reward,

    def sample(self, batch_size, beta=0.4

    def update_priorities(self, batch_ind

    def __len__(self):
```

```python
    def sample(self, batch_size, beta=0.4):
        if len(self.buffer) == self.capacity:
            prios = self.priorities
        else:
            prios = self.priorities[:self.pos]

        probs  = prios ** self.prob_alpha
        probs /= probs.sum()

        indices = np.random.choice(len(self.buffer), batch_size, p=probs)
        samples = [self.buffer[idx] for idx in indices]

        total    = len(self.buffer)
        weights  = (total * probs[indices]) ** (-beta)
        weights /= weights.max()
        weights  = np.array(weights, dtype=np.float32)

        batch       = list(zip(*samples))
        states      = np.concatenate(batch[0])
        actions     = batch[1]
        rewards     = batch[2]
        next_states = np.concatenate(batch[3])
        dones       = batch[4]

        return states, actions, rewards, next_states, dones, indices, weights
```

IS weights

```
class NaivePrioritizedBuffer(object):
    def __init__(self, capacity, prob_alp

    def push(self, state, action, reward,

    def sample(self, batch_size, beta=0.4

    def update_priorities(self, batch_ind

    def __len__(self):
```

```
    def update_priorities(self, batch_indices, batch_priorities):
        for idx, prio in zip(batch_indices, batch_priorities):
            self.priorities[idx] = prio

    def __len__(self):
        return len(self.buffer)
```

```python
class NaivePrioritizedBuffer(object):
    def __init__(self, capacity, prob_alp

    def push(self, state, action, reward,

    def sample(self, batch_size, beta=0.4

    def update_priorities(self, batch_ind

    def __len__(self):
```

TD-Error로 업데이트

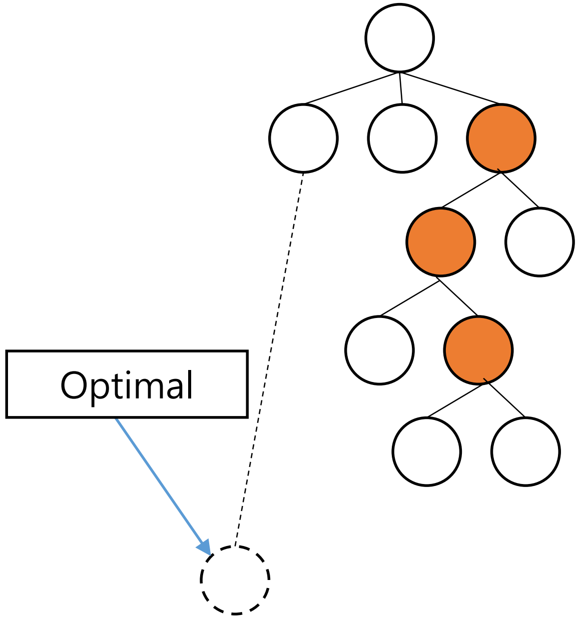
```python
    def update_priorities(self, batch_indices, batch_priorities):
        for idx, prio in zip(batch_indices, batch_priorities):
            self.priorities[idx] = prio

    def __len__(self):
        return len(self.buffer)
```

NoisyNet

Noisy Networks for Exploration

Exploitation

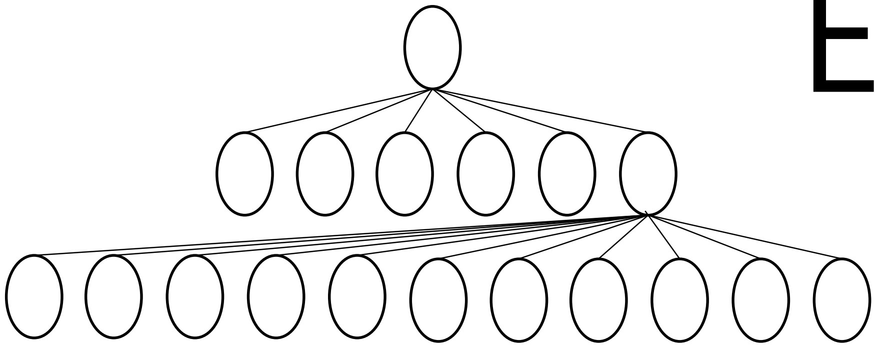
Exploration



High exploitation

Exploitation

Exploration



High exploration

Efficient

Exploration

Exploration methods

ϵ −greedy

일정 확률 (ϵ) 만큼 무작위로 행동

Entropy regularization

Loss에 추가하는 패널티로 한쪽으로 치우치지 않게 함

$$-\sum_{a} \pi(s,a) \log \pi(s,a)$$

$\epsilon - \text{greedy}$, Entropy regularization

$\epsilon - \text{greedy}$, Entropy regularization

$\Downarrow$

Random perturbations

$\epsilon - \text{greedy}$, Entropy regularization

⇩

Random perturbations

⇩

Hard to the large-scale behavioural patterns

NoisyNet!!

NoisyNet learn perturbations of the network weights are used to drive exploration

$$\theta \coloneqq \mu + \Sigma \odot \epsilon$$

Learnable parameters

$$\theta \coloneqq \mu + \Sigma \odot \epsilon$$

Noise variables

$\zeta \coloneqq (\mu, \Sigma)$ Learnable parameters

$$\theta \coloneqq \mu + \Sigma \odot \epsilon$$

Noise variables

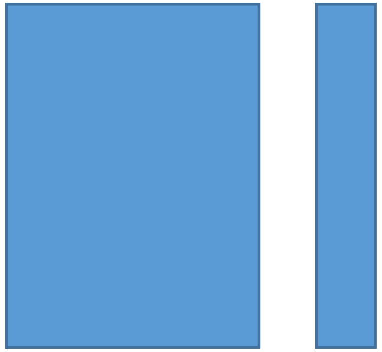
$$y = wx + b$$

$$y \coloneqq (\mu^w + \sigma^w \odot \epsilon^w)x + \mu^b + \sigma^b \odot \epsilon^b$$

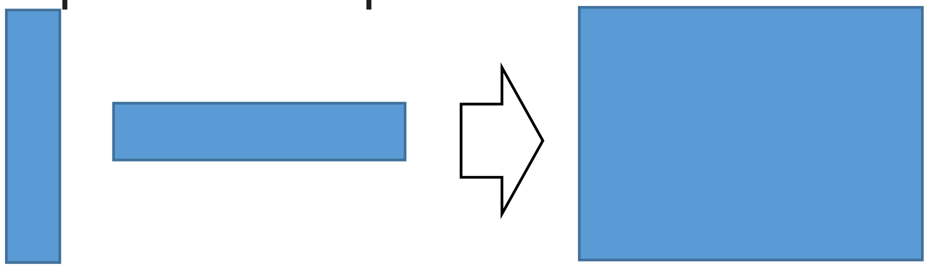
NoisyNet

- p inputs and q outputs

- Independent Gaussian noise
  - Using an independent Gaussian noise entry per weight
  - pq+q



- Factorised Gaussian noise
  - Using and independent noise per each output and input
  - p+q



Loss

$$L(\theta) = \mathbb{E}\left[\mathbb{E}_{(x,a,r,y)\sim D}[r + \gamma \max_{b\in A} Q(y,b;\theta^{-}\ ) - Q(x,a;\theta)]^2\right]$$

$$\bar{L}(\zeta) = \mathbb{E}\left[\mathbb{E}_{(x,a,r,y)\sim D}[r + \gamma \max_{b\in A} Q(y,b,\epsilon';\zeta^{-}) - Q(x,a,\epsilon;\zeta]^2\right]$$

Loss

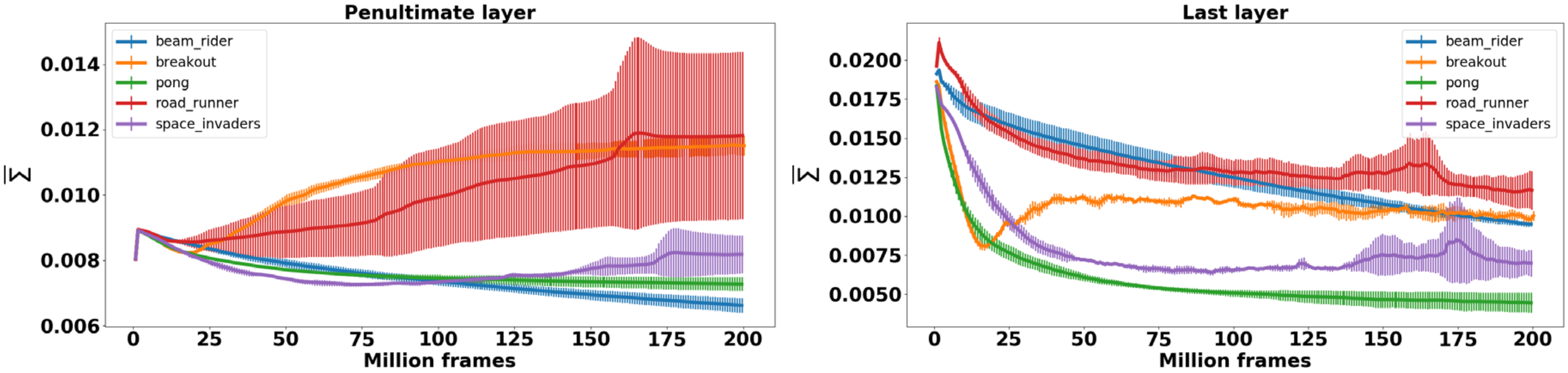
$$L(\theta) = \mathbb{E}\left[\mathbb{E}_{(x,a,r,y)\sim D}[r + \gamma \max_{b\in A} Q(y, b; \boxed{\theta^-}) - Q(x, a; \boxed{\theta)}]^2\right]$$

$$\bar{L}(\zeta) = \mathbb{E}\left[\mathbb{E}_{(x,a,r,y)\sim D}[r + \gamma \max_{b\in A} Q(y, b\boxed{, \epsilon'; \zeta^-)} - Q(x, a\boxed{, \epsilon; \zeta]}^2\right]$$

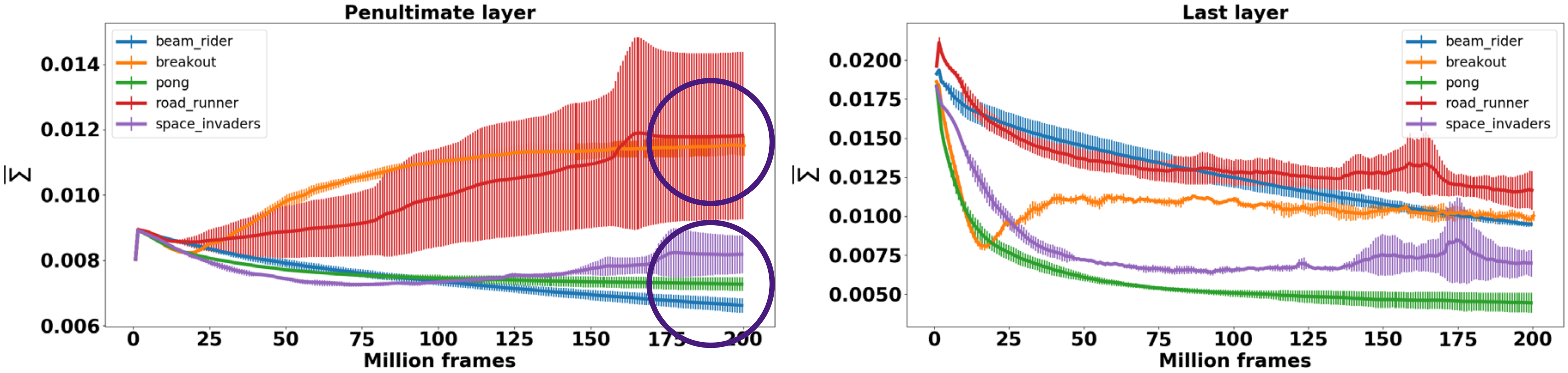
Initialisation of NoisyNet

- An unfactorized NoisyNet
  - $\mu_{i,j} \sim u\left[-\sqrt{\frac{3}{p}}, +\sqrt{\frac{3}{p}}\right]$
    - p: The number of inputs
  - $\sigma_{i,j} = 0.017$
- Factorised NosiyNet
  - $\mu_{i,j} \sim u\left[-\frac{1}{\sqrt{p}}, +\frac{1}{\sqrt{p}}\right]$
  - $\sigma_{i,j} = \frac{\sigma_0}{\sqrt{p}}$
    - $\sigma_0 = 0.5$

The learning curves of the average noise parameter $\bar{\Sigma}$



The learning curves of the average noise parameter $\bar{\Sigma}$



Factorised NosiyNet

```
class NoisyLinear(nn.Module):
    def __init__(self, in_features, out_features, std_init=0.4):


    def forward(self, x):


    def reset_parameters(self):


    def reset_noise(self):


    def _scale_noise(self, size):
```

```python
class NoisyLinear(nn.Module):
    def __init__(self, in_features

    def forward(self, x):

    def reset_parameters(self):

    def reset_noise(self):

    def _scale_noise(self, size):
```

```python
def __init__(self, in_features, out_features, std_init=0.5):
    super(NoisyLinear, self).__init__()

    self.in_features  = in_features
    self.out_features = out_features
    self.std_init     = std_init

    self.weight_mu    = nn.Parameter(torch.FloatTensor(out_features, in_features))
    self.weight_sigma = nn.Parameter(torch.FloatTensor(out_features, in_features))
    self.register_buffer('weight_epsilon', torch.FloatTensor(out_features, in_features)

    self.bias_mu    = nn.Parameter(torch.FloatTensor(out_features))
    self.bias_sigma = nn.Parameter(torch.FloatTensor(out_features))
    self.register_buffer('bias_epsilon', torch.FloatTensor(out_features))

    self.reset_parameters()
    self.reset_noise()
```

```python
class NoisyLinear(nn.Module):
    def __init__(self, in_features

    def forward(self, x):

    def reset_parameters(self):

    def reset_noise(self):

    def _scale_noise(self, size):
```

```python
def __init__(self, in_features, out_features, std_init=0.5):
    super(NoisyLinear, self).__init__()

    self.in_features  = in_features
    self.out_features = out_features
    self.std_init     = std_init

    self.weight_mu    = nn.Parameter(torch.FloatTensor(out_features, in_features))
    self.weight_sigma = nn.Parameter(torch.FloatTensor(out_features, in_features))
    self.register_buffer('weight_epsilon', torch.FloatTensor(out_features, in_features)

    self.bias_mu    = nn.Parameter(torch.FloatTensor(out_features))
    self.bias_sigma = nn.Parameter(torch.FloatTensor(out_features))
    self.register_buffer('bias_epsilon', torch.FloatTensor(out_features))

    self.reset_parameters()
    self.reset_noise()
```

Learnable parameters

```python
class NoisyLinear(nn.Module):
    def __init__(self, in_features,

    def forward(self, x):

    def reset_parameters(self):

    def reset_noise(self):

    def _scale_noise(self, size):
```

Factorised NosiyNet

$$\mu_{i,j} \sim u\left[-\frac{1}{\sqrt{p}}, +\frac{1}{\sqrt{p}}\right]$$

```python
def reset_parameters(self):
    mu_range = 1 / math.sqrt(self.weight_mu.size(1))

    self.weight_mu.data.uniform_(-mu_range, mu_range)
    self.weight_sigma.data.fill_(self.std_init / math.sqrt(self.weight_sigma.size(1)))

    self.bias_mu.data.uniform_(-mu_range, mu_range)
    self.bias_sigma.data.fill_(self.std_init / math.sqrt(self.bias_sigma.size(0)))

def reset_noise(self):
    epsilon_in  = self._scale_noise(self.in_features)
    epsilon_out = self._scale_noise(self.out_features)

    self.weight_epsilon.copy_(epsilon_out.ger(epsilon_in))
    self.bias_epsilon.copy_(self._scale_noise(self.out_features))

def _scale_noise(self, size):
    x = torch.randn(size)
    x = x.sign().mul(x.abs().sqrt())
    return x
```

```python
class NoisyLinear(nn.Module):
    def __init__(self, in_features,

    def forward(self, x):

    def reset_parameters(self):

    def reset_noise(self):

    def _scale_noise(self, size):
```

```python
def reset_parameters(self):
    mu_range = 1 / math.sqrt(self.weight_mu.size(1))

    self.weight_mu.data.uniform_(-mu_range, mu_range)
    self.weight_sigma.data.fill_(self.std_init / math.sqrt(self.weight_sigma.size(1)))

    self.bias_mu.data.uniform_(-mu_range, mu_range)
    self.bias_sigma.data.fill_(self.std_init / math.sqrt(self.bias_sigma.size(0)))

def reset_noise(self):
    epsilon_in  = self._scale_noise(self.in_features)
    epsilon_out = self._scale_noise(self.out_features)

    self.weight_epsilon.copy_(epsilon_out.ger(epsilon_in))
    self.bias_epsilon.copy_(self._scale_noise(self.out_features))

def _scale_noise(self, size):
    x = torch.randn(size)
    x = x.sign().mul(x.abs().sqrt())
    return x
```

Factorised

```python
class NoisyLinear(nn.Module):
    def __init__(self, in_features,

    def forward(self, x):

    def reset_parameters(self):

    def reset_noise(self):

    def _scale_noise(self, size):
```

```python
def forward(self, x):
    if self.training:
        weight = self.weight_mu + self.weight_sigma.mul(self.weight_epsilon)
        bias   = self.bias_mu   + self.bias_sigma.mul(self.bias_epsilon)
    else:
        weight = self.weight_mu
        bias   = self.bias_mu

    return F.linear(x, weight, bias)
```

```python
class NoisyLinear(nn.Module):
    def __init__(self, in_features,

    def forward(self, x):

    def reset_parameters(self):

    def reset_noise(self):

    def _scale_noise(self, size):
```

```python
def forward(self, x):
    if self.training:
        weight = self.weight_mu + self.weight_sigma.mul(self.weight_epsilon)
        bias   = self.bias_mu   + self.bias_sigma.mul(self.bias_epsilon)
    else:
        weight = self.weight_mu
        bias   = self.bias_mu

    return F.linear(x, weight, bias)
```

$$\theta := \mu + \sum \odot \epsilon$$

Code:	https://github.com/higgsfield/RL-Adventure
PER:	https://arxiv.org/abs/1511.05952
NoisyNet:	https://arxiv.org/abs/1706.10295

감사합니다

Q&A