DQN Extensions

한양대학교 최솔비

Prioritized replay buffer

- basic DQN 은 에피소드 안의 transition 들 간의 상관관계를 깨기 위해 replay buffer를 사용해왔음.
    - stochastic gradient descent 방법을 사용하기 위해서는 학습데이터가 i.i.d 특성을 지녀야함.
    - buffer 로부터 랜덤하게 샘플링 하여 학습 배치 구성.
- Prioritized replay buffer: 기존의 방식대로 buffer에서 uniform distribute 로 random하게 샘플링 하는 것이 아닌,
- training loss 에 따라 replay buffer sample 들에 우선순위를 부여하고, 이 우선순위 비율로 샘플링한다.
- 수렴속도를 향상시킬 수 있다.

Prioritized replay buffer

> "Train more on data that surprises you."

- 'unusual'(우선순위가 높은) sample 학습 vs rest of the buffer 학습
    - balance 를 잘 맞춰야 함
    - 너무 적은 데이터 집합만 계속 학습시키면 i.i.d 특성을 잃게 되며, overfit 될 수 있음.

Prioritized replay buffer

- 다음 식에 의해 우선순위가 계산된다: $P(i) = \frac{{p_i}^\alpha}{\sum_k {p_k}^\alpha}$
- $P(i)$는 확률
- p_i: 버퍼의 i번째 샘플의 우선순위
- α: priority에 주기로 결정한 가중치.
  - α가 0이면 uniform 하게 샘플링한다.
  - 논문에는 0.6에서 시작해서 튜닝해감.
- Bellman update를 할때의 loss 값에 우선순위를 비례하게 만든다.
- 새로 버퍼에 추가된 샘플은 최대 우선순위를 준다.(곧 샘플링 되도록)

Prioritized replay buffer

- 이 방법을 사용하면 일부 샘플들을 다른 샘플에 비해 더 자주 뽑기 때문에 SGD 를 이용하기 위해 값을 보충해주어야 한다. (needs to be compensated for)
- sample weight 을 사용한다. 각각의 sample loss 에 곱하면 됨.
- $w_i = (N \cdot P(i))^{-\beta}$
- N : 버퍼안의 sample 개수
- β : 0과 1사이.
  - With $\beta = 1$, the bias introduced by the sampling is fully compensated.
  - 0과 1 사이에서 시작해서 1로 증가시켜 나가는 것이 좋다.

Implementation

- 새로운 replay buffer 가 필요함.
  - 우선순위를 tracking
  - 우선순위에 따라 batch 샘플링
  - weight 을 계산
  - loss 값 나온 후에 priority 갱신
- loss function 수정
  - loss value 를 replay buffer 에 전달해야함.

Implementation

- priority 계산 때문에 sampling 이 $O(1)$ 시간에 불가능하게 됨.
- 버퍼를 단순 리스트를 사용한다면, batch 샘플링 마다 모든 우선순위를 계산해야하기 때문에 $O(N)$ 시간이 걸림.
- segment tree 등을 사용한다면 $O(\log N)$ 시간안에 샘플링 할수 있음.

```
PRIO_REPLAY_ALPHA = 0.6
BETA_START = 0.4
BETA_FRAMES = 100000
# 베타값이 0.4 에서 시작해서 100000프레임 동안 1.0으로 바뀌어 나감.

class PrioReplayBuffer:
        def __init__(self, exp_source, buf_size,
                                    prob_alpha = 0.6):
                self.exp_source_iter = iter(exp_source)
                self.prob_alpha = prob_alpha
                self.capacity = buf_size
                self.pos = 0
                self.buffer = []
                self.priorities = np.zeros((buf_size, ),
                                    dtype = np.float32)
```

- circular buffer 를 사용함.
- Numpy array에 priority 를 저장.

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

def populate(self, count):
    max_prio = self.priorities.max() if self.buffer
                                        else 1.0
    for _ in range (count):
        # count 만큼 샘플을 얻어 낸 후 circular buffer에 저장.
        sample = next (self.exp_source_iter)
        if len(self.buffer) < self.capacity:
            self.buffer.append(sample)
        else :
            self.buffer[self.pos] = sample

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

- pull the given amount of transitions from the `ExperienceSource` object and store them in the buffer.

```
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.probs_alpha
    probs /= probs.sum()
        # 우선순위 가중치를 준 확률로 batch 선택(샘플링).
    indices = np.random.choice(len(self.buffer),
                               batch_size, p = probs)
    samples = [self.buffer[idx] for idx in indices]
        # weight 계산
    total = len(self.buffer)
    weights = (total * probs[indices]) ** (-beta)
    weights /= weights.max()
    # 샘플링한 batch(학습데이터) 와 indices, weight리턴.
    return samples, indices, weights
```

- priority 를 probabilities 로 바꿈.
- indices 는 샘플링한 데이터의 priority 를 갱신하는 데 필요하다.

```
def calc_loss(batch, batch_weights, net, tgt_net,
                          gamma, device ="cpu"):
        states, actions, rewards, dones, next_states =
                          common.unpack_batch(batch)
        ...
        batch_weights_v = torch.tensor(batch_weights).to(

        ...

        expected_state_action_values = next_state_values.
                          detach() * gamma + rewards_v
        # weight의 MSE loss를 연산한다.
        losses_v = batch_weights_v *
(state_action_values - expected_state_action_values) ** 2

        return losses_v.mean(), losses_v + 1e-5
```

- 모든 각각의 샘플의 loss value 를 갖고 있게 됨. -> replay buffer 로 넘겨서 우선순위를 갱신하는데에 사용.
- 1e-5 를 더하는 이유는 loss value 가 0이 되어 zero priority로 만드는 것을 막기 위해서 이다.

main 함수

```
...
frame_idx = 0
beta = BETA_START

with common.RewardTracker(writer, param['stop_reward'])
                                    as reward_tracker:
  while True:
    frame_idx += 1
    buffer.populate(1)
    epsilon_tracker.frame(frame_idx)
    beta = min(1.0, BETA_START + frame_idx *
                (1.0 - BETA_START) / (BETA_FRAMES))
...
```

- beta 값을 linear하게 증가시킨다.(priority replay buffer weights' adjustment)

main 함수

```
...
  optimizer.zero_grad()
  batch, batch_indices, batch_weights = buffer.
                sample(params['batch_size'], beta)
  loss_v, sample_prios_v = calc_loss(batch,
                batch_weights, net, tgt_net.target_model,
                params['gamma'], device = device)
  loss_v.backward()
  optimizer.step()
  buffer.update_priorities(batch_indices,
                sample_prios_v.data.cpu(), numpy())
...
```

- `sample_prios_v` 는 배치의 각각의 샘플에 대한 individual loss value 를 담고 있는 tensor 이다. replay buffer 에서 우선순위를 갱신하는 데에 사용한다.

result

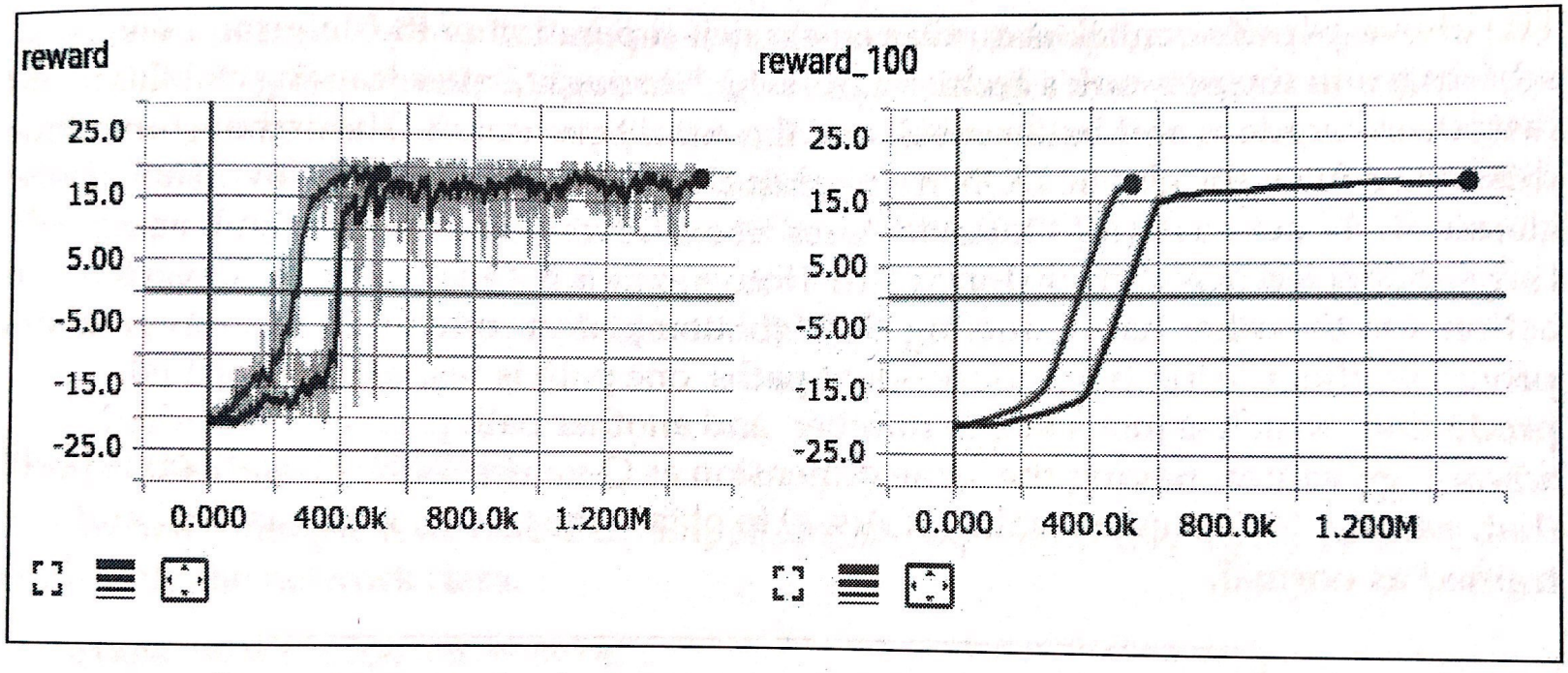


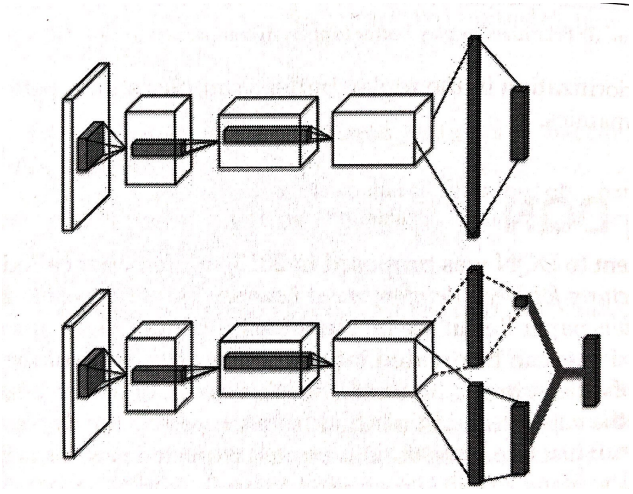
Figure 10: Prioritized replay buffer (upper) in comparison to basic DQN (lower)

- better convergence dynamics

Dueling DQN

- 네트워크가 추정하려고 하는 Q value 를 다음과 같이 나눈다.
- $Q(s,a) = V(s) + A(s,a)$
- $V(s)$: the value of the state
- $A(s,a)$: the advantage of the value in this state
  - how much extra reward some particular action from the state bring us.
- Value 와 Advantage 를 네트워크 아키텍쳐 상에서 분리함.
- 학습 안정성이 높아지며 수렴속도가 빨라진다.

Dueling DQN



- 일반적인 DQN 방식(위)에서는 network 의 output이 각각의 action에 대한 Q value 이다.
- dueling DQN(아래)에서는 두개의 독립된 path로 처리한다.
- state 에 대한 V(s) (single number) 와, 각각 action에 대한 A(s,a) (기존의 Q value dimension과 같음).
- 이후 V(s)와 A(s,a)를 합쳐서 Q(s,a)를 얻어 사용한다.

Dueling DQN

- 한 state 에 대한 advantage 값들의 평균을 0 이 되도록 만들어야 함.(?)
- $Q(s,a) = V(s) + A(s,a) - \frac{1}{N}\sum_k A(s,k)$

```
class DuelingDQN(nn.MOdule):
  def __init__(self, input_shape, n_actions):
    super(DuelingDQN, self).__init__()

    self.conv = nn.Sequential (
        nn.Conv2d(input_shape[0], 32,
                kernel_size = 8, stride=4),
        nn.ReLU(),
        nn.Conv2d(32, 64, kernel_size =4, stride =2),
        nn.ReLU(),
        nn.Conv2d(64, 64, kernel_size =3, stride =1),
        nn.ReLU()
        )
```

- 여기까지는 전과 같음.

```
conv_out_size = self._get_conv_out(input_shape)
self.fc_adv = nn.Sequential(
    nn.Linear(conv_out_size, 512),
    nn.ReLU(),
    nn.Linear(512, n_actions)
    )
self.fc_val = nn.Sequential(
    nn.Linear(conv_out_size, 512),
    nn.ReLU(),
    nn.linear(512, 1)
    )
```

- advantages를 위한 layer(output dimension은 액션 개수) 와 value 예측을 위한 layer(output dimension 은 1)로 나눠짐.

```
def _get_conv_out(self, shape):
    o = self.conv(torch.zeros(1, *shape))
    return int(np.prod(o.size()))

def forward(self, x):
    fx = x.float() / 256
    conv_out = self.conv(fx).view(fx.size()[0], -1)
    val = self.fc_val(conv_out)
    adv = self.fc_adv(conv_out)
    return val + adv - adv.mean()
```

- advantage값과 value 값을 더한후, advantage 의 평균을 빼서, 모든 action에 대한 advantage 의 합이 0이 될 수 있도록 한다.

result

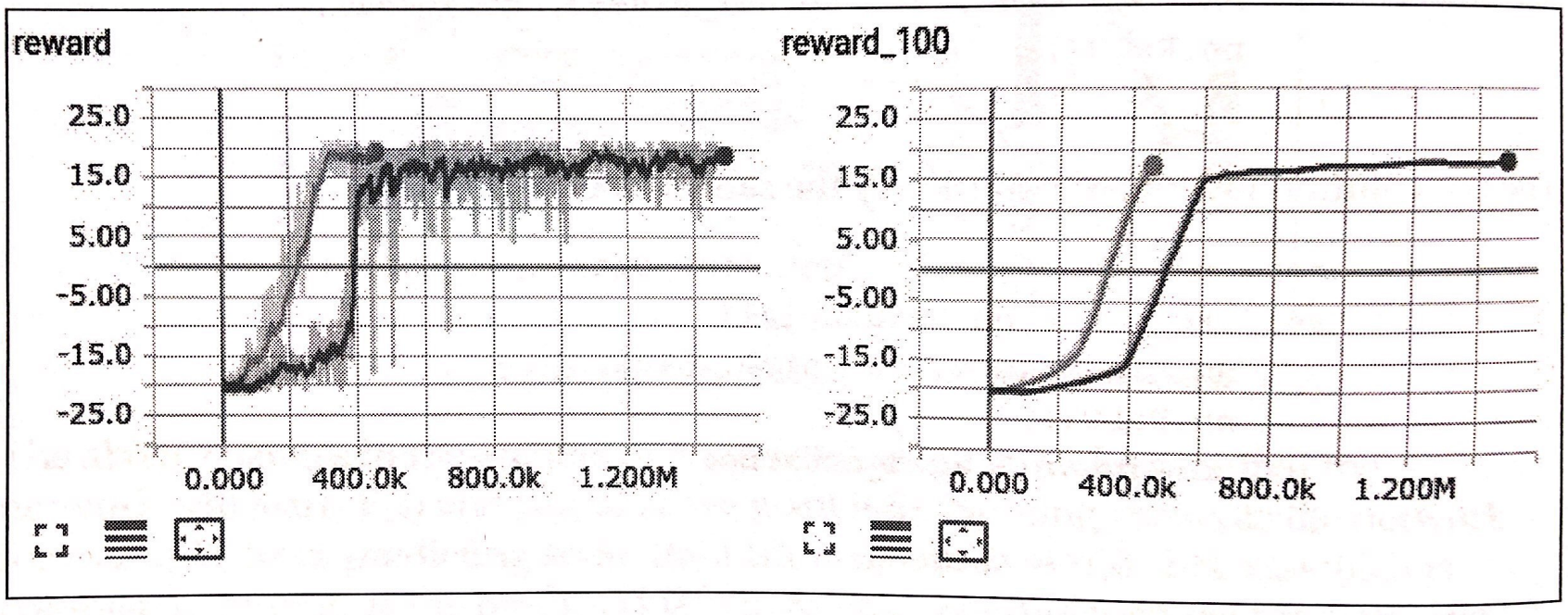


Figure 12: The convergence of a dueling architecture (light) in comparison to a basic DQN (dark)

- Pong 벤치마크에 대해 실행한 결과, dueling DQN 의 수렴이 더 빠르다.

Categorical DQN

- Q-value 를 Q-value probability distribution으로 바꾸자는 아이디어.
- Q-learning 과 value iteration method 는 value of actions 혹은 value of states등 simple numbers 로 예측한 total reward를 표현한다.
- 하나의 숫자로 모든 미래의 가능한 rewards 를 표현하는 것은 복잡한 환경에서의 미래예측과 맞지 않음(보통은 stocastic)
- 분산(variance)의 개념을 도입한 새로운 수학적 모델.
- $Z(x, a) = R(x, a) + \gamma Z(x', a')$
- Bellman equation과 비슷하지만, Z 와 R 이 숫자가 아닌 probability distribution이다.

implementation

- 구현시 loss function 에서 차이가 있게 된다: distributions간의 비교 (Kullback-Leibler divergence)
- distribution을 나타내기 위해 'parametric distribution'사용.
- values 의 range 를 나누어서 range 마다 value 가 들어가는 개수를 저장.
- 이 구간은 `N_ATOMS = 51` 개가 적절하다는 결론.

result

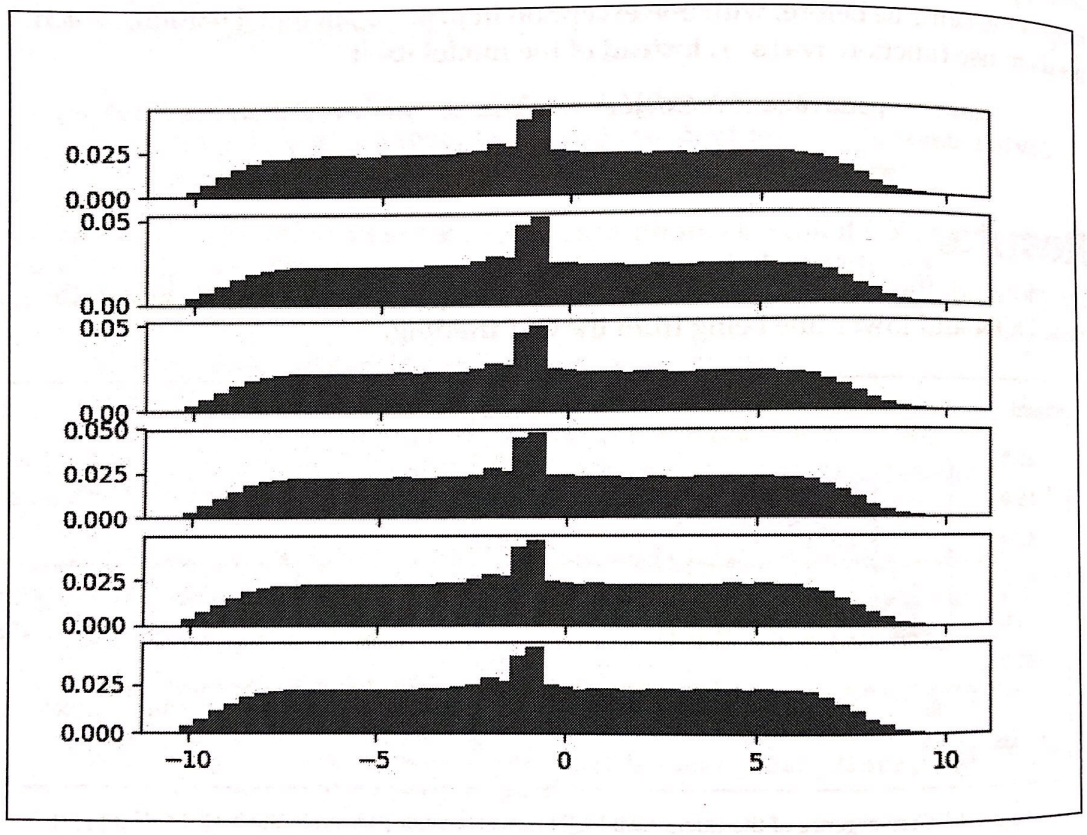


Figure 18: Probability distribution at the beginning of training

- 학습 시작시에는 수렴되지 않은 상태라 분포가 퍼져있고, 각각 action들에 대한 기대 reward 가 마이너스 값을 가리킨다.

result

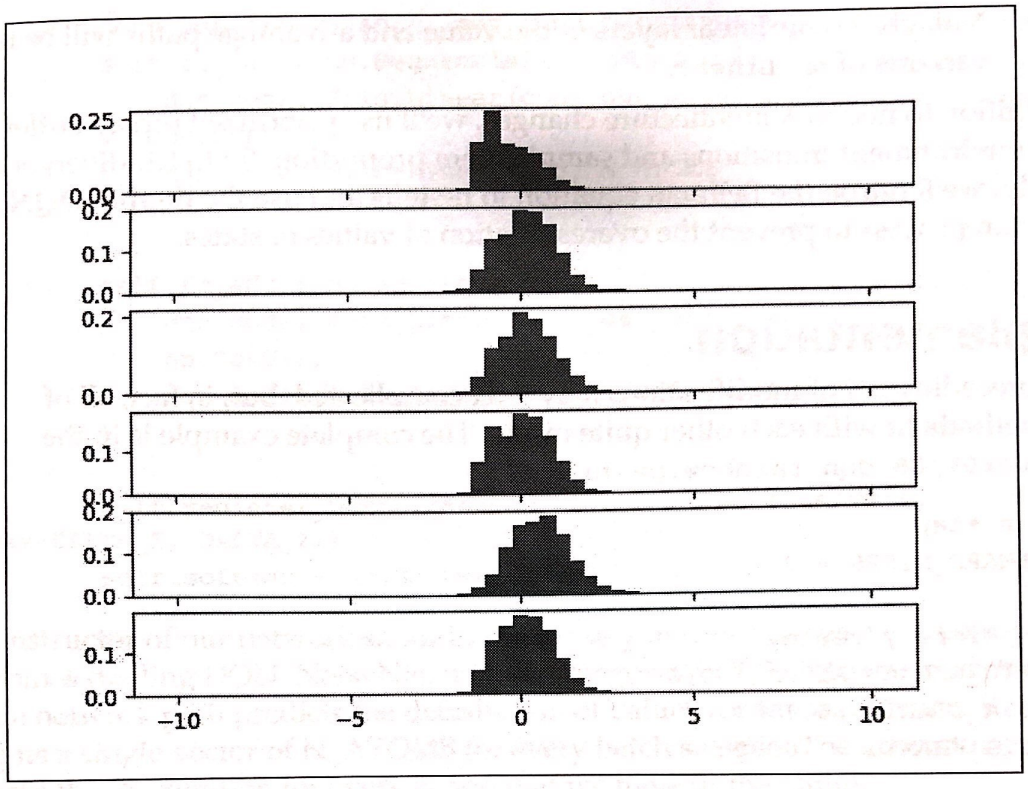


Figure 19: Probability distribution produced by the trained network

- 500k 의 학습 후 각각의 action들 이 다른 분포를 가지게 되었다.
- 첫번째 action은 Noop 인데, 아무것도 하지 않으면 질 확률이 높은 분포를 가지며,
- 5번째 action이 이길 확률이 높은 분포를 가지는 것을 볼 수 있다.

Combining everything

- Categorical DQN
- Dueling DQN
- NoisyNet
- Prioritized replay buffer
- unroll Bellman equation to n-steps
- double DQN action selection process

끝