C51

Paper Link: https://arxiv.org/pdf/1707.06887.pdf

Key Features

- Distributional RL is a new reinforcement learning mechanism that learns Q-value as a distribution, unlike the conventional RL algorithms learn Q-value as a scalar.
- C51 is one of the algorithms of the distribution RL. To calculate the expected distribution of the value, C51 uses the distribution version of the Bellman equation.
- Although learning a distribution that takes into account uncertainty increases the stability of learning, C51 is a theoretically incomplete algorithm by using cross-entropy as a loss function. Because, cross-entropy is not mathematically proven convergent on gamma-contraction which basically must be satisfied for training distributional RL.

Background

Distributional RL: learn uncertainty with respect to reward

In conventional RL, agent applies the optimal action for the current state to the environment and is rewarded as a result. However, in the real world, even if it is the same state-action pair, the reward may differ from time to time. Distributional RL is a method suggested to learn this uncertainty.

Method

Distributional version of Bellman equation

Comparison between the Bellman equation for conventional RL and the Bellman equation for distributional RL are shown as follows.

$$conventional\ RL \quad Q(x,a) = \mathbb{E}R(x,a) + \gamma \mathbb{E}Q(X',A') \tag{1}$$

$$distributional\ RL \quad Z(x,a) \overset{D}{=} R(x,a) + \gamma Z(X',A') \tag{2}$$

Z is distribution of reward. Unlike conventional RL, distributional RL uses a value distribution to estimate expected reward. This method can learn the uncertainty of the reward obtained from the same pair of state-action sample.

To learn distribution can make learning stable because the distributional Bellman operator used on its calculation preserves multimodality in value distributions and approximating the full distribution mitigates the effects of learning from a non-stationary policy.

Replace scalar value to value distribution

Distributional RL network replaces update target from scalar(value) to vector(value distribution). Therefore, the output of distributional RL network also should be a vector that consists of value distribution with respect to actions. The each value distribution is based on a discrete distribution, because the discrete distribution has the advantages of being highly expressive and computationally friendly.

The value distribution for each action consists of probabilities for a specific value called support(or atom) together. To implement the supports, the following three hyperparameters should be determined: the number of supports, the maximum value of support, and the minimum value of support.

Difference with DQN

C51 is an algorithm that combines the idea of distribution RL with DQN. C51 has some differences compared to the original DQN.

First, C51 calculates the expected value of the discrete probability distribution for the Q-value as follows.

$$Q(x,a) = \sum_{i=0}^{N} z_i p_i(x,a)$$

where, N is the number of supports. After calculation, agent selects action which has maximum Q-value.

Second, C51 calculates the difference between the update target distribution and the predicted value distribution through cross-entropy.

$$Loss = -\sum_{i=0}^{N} m_i log p_i(x,a)$$

where, N is the number of supports, m is target distribution and p is probability of support. N is a hyperparameter and p is the output of the network. Then, how to get target distribution m? It will be described in the next section.

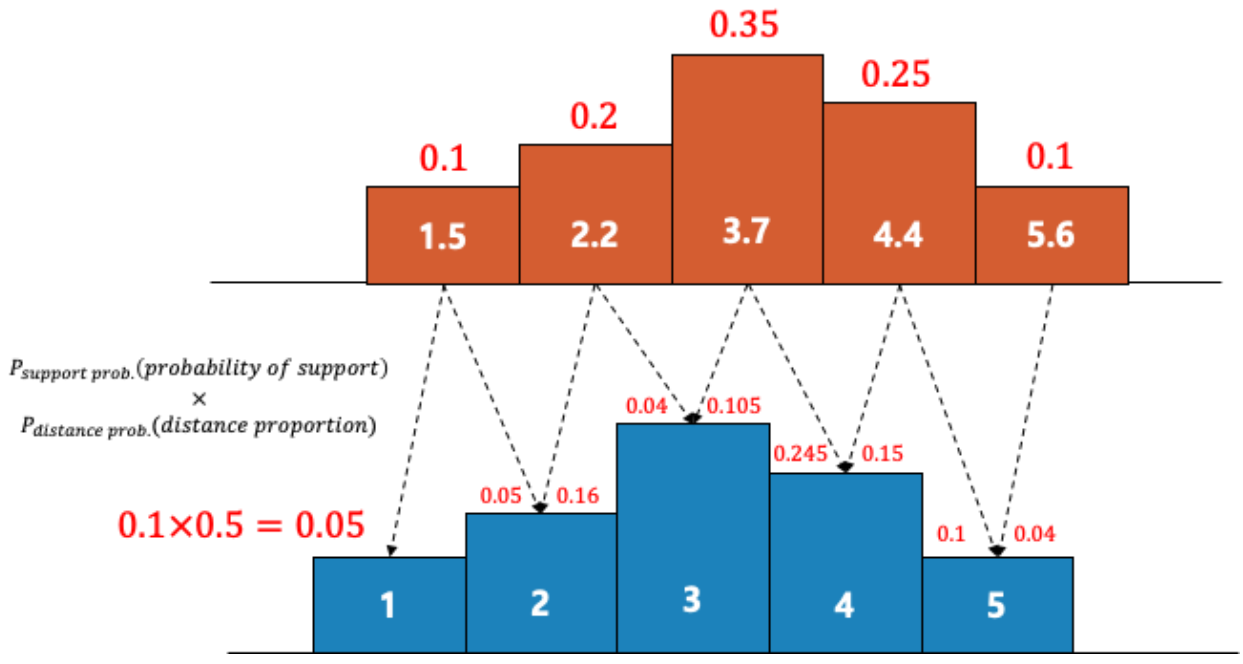
Target distribution calculation

Unlike calculating the scalar update target value of the conventional RL, calculating the update target distribution is a difficult task. First of all, the equation of target distribution is as follows.

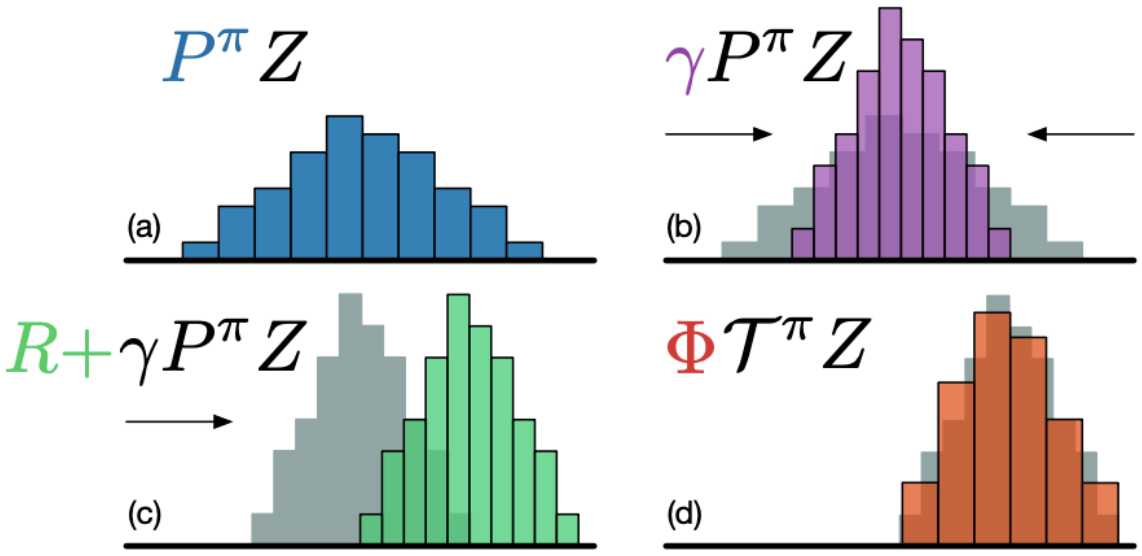
$$\hat{\mathrm{T}} z_i = [r_t + \gamma z_i]_{V_{min}}^{V_{max}}$$

where, $\hat{\mathrm{T}}$ is Bellman operator. V_{min} and V_{max} are lower-bound and upper-bound of support, respectively. If the support of the target distribution is calculated in the above equation, the predicted distribution and support will be different. Since cross-entropy between distributions with different supports cannot be calculated, the difference between target distribution and predicted distribution cannot be obtained in this way. To solve this, new distribution is projected on old distribution supports.

By projecting the new distribution on the original distribution, new support's probabilities are distributed respectively to the original's supports in proportion to the distance from original supports. The figure below expresses this content.



The figure below shows the entire process of calculating target distribution.



By obtaining target distribution as above, cross-entropy can be calculated and all the preparations needed for learning is finished.

Limitation of C51

For the convergence of distributional, the distance metric used between learning must satisfy gamma contraction. However, cross-entropy does not satisfy gamma contraction mathematically. Therefore, C51 has a limitation in that it does not guarantee convergence.

Algorithm

Algorithm 1 Categorical Algorithm

input A transition $x_t, a_t, r_t, x_{t+1}, \gamma_t \in [0, 1]$

$Q(x_{t+1}, a) := \sum_i z_i p_i(x_{t+1}, a)$

$a^* \leftarrow \arg\max_a Q(x_{t+1}, a)$

$m_i = 0, \quad i \in 0, \ldots, N-1$

for $j \in 0, \ldots, N-1$ **do**

Compute the projection of $\hat{\mathcal{T}} z_j$ onto the support $\{z_i\}$

$\hat{\mathcal{T}} z_j \leftarrow [r_t + \gamma_t z_j]_{V_{\text{MIN}}}^{V_{\text{MAX}}}$

$b_j \leftarrow (\hat{\mathcal{T}} z_j - V_{\text{MIN}})/\Delta z \quad \# \, b_j \in [0, N-1]$

$l \leftarrow \lfloor b_j \rfloor, u \leftarrow \lceil b_j \rceil$

Distribute probability of $\hat{\mathcal{T}} z_j$

$m_l \leftarrow m_l + p_j(x_{t+1}, a^*)(u - b_j)$

$m_u \leftarrow m_u + p_j(x_{t+1}, a^*)(b_j - l)$

end for

output $-\sum_i m_i \log p_i(x_t, a_t)$ # Cross-entropy loss

Implementation on JORLDY

- C51 JORLDY Implementation

```
### act function ###
# act = epsilon-greedy policy-based action selection

@torch.no_grad()
def act(self, state, training=True):
    self.network.train(training)
    epsilon = self.epsilon if training else self.epsilon_eval

    if np.random.random() < epsilon:
```

```
        batch_size = (
            state[0].shape[0] if isinstance(state, list) else state.shape[0]
        )
        action = np.random.randint(0, self.action_size, size=(batch_size, 1))
    else:
        logits = self.network(self.as_tensor(state))
        _, q_action = self.logits2Q(logits)
        action = torch.argmax(q_action, -1, keepdim=True).cpu().numpy()
    return {"action": action}
```

```
### learn function  ###
# 1. Calculate target distribution and the project it on original support
# 2. Calculate cross-entropy loss
# 3. Update Q-network

def learn(self):
...

    logit = self.network(state)
    p_logit, q_action = self.logits2Q(logit)

    action_eye = torch.eye(self.action_size, device=self.device)
    action_onehot = action_eye[action.long()]

    p_action = torch.squeeze(action_onehot @ p_logit, 1)

    target_dist = torch.zeros(
        self.batch_size, self.num_support, device=self.device, requires_grad=False
    )
    with torch.no_grad():
        target_p_logit, target_q_action = self.logits2Q(
            self.target_network(next_state)
        )

        target_action = torch.argmax(target_q_action, -1, keepdim=True)
        target_action_onehot = action_eye[target_action.long()]
        target_p_action = torch.squeeze(target_action_onehot @ target_p_logit, 1)

        Tz = reward.expand(-1, self.num_support) + (1 - done) * self.gamma * self.z
        b = torch.clamp(Tz - self.v_min, 0, self.v_max - self.v_min) / self.delta_z
        l = torch.floor(b).long()
        u = torch.ceil(b).long()

        support_eye = torch.eye(self.num_support, device=self.device)
        l_support_onehot = support_eye[l]
        u_support_onehot = support_eye[u]

        l_support_binary = torch.unsqueeze(u - b, -1)
        u_support_binary = torch.unsqueeze(b - l, -1)
        target_p_action_binary = torch.unsqueeze(target_p_action, -1)
```

```
        lluu = (
            l_support_onehot * l_support_binary
            + u_support_onehot * u_support_binary
        )
        target_dist += done * torch.mean(
            l_support_onehot * u_support_onehot + lluu, 1
        )
        target_dist += (1 - done) * torch.sum(target_p_action_binary * lluu, 1)
        target_dist /= torch.clamp(
            torch.sum(target_dist, 1, keepdim=True), min=1e-8
        )

    max_Q = torch.max(q_action).item()
    max_logit = torch.max(logit).item()
    min_logit = torch.min(logit).item()

    loss = -(target_dist * torch.clamp(p_action, min=1e-8).log()).sum(-1).mean()
    self.optimizer.zero_grad(set_to_none=True)
    loss.backward()
    self.optimizer.step()

...
```

```
### logits2Q function ###
# logits2Q = Q-distribution expectation

def logits2Q(self, logits):
    _logits = logits.view(logits.shape[0], self.action_size, self.num_support)
    _logits_max = torch.max(_logits, -1, keepdim=True).values
    p_logit = torch.exp(F.log_softmax(_logits - _logits_max, dim=-1))

    z_action = self.z.expand(p_logit.shape[0], self.action_size, self.num_support)
    q_action = torch.sum(z_action * p_logit, dim=-1)

    return p_logit, q_action
```