QR-DQN(Quantile Regression DQN)

Paper Link: https://arxiv.org/pdf/1710.10044.pdf

Key Features

- QR-DQN is a distributional RL algorithm, which is a variant of C51. It applies quantile regression to find a suitable support.
- QR-DQN is an excellent algorithm in two aspects
  - First, QR-DQN does not require a projection process, because it calculates target distribution according to the number of supports fixed in advance by fixing proportions. This simplifies the computational process.
  - Second, C51 is an algorithm that could not guarantee convergence because it could not satisfy gamma-contraction by using cross-entropy as a distance metric, but QR-DQN is an algorithm that guarantees convergence by using Wasserstein metric that satisfies gamma-contraction as a distance metric.
- Additionally, QR-DQN enables more stable learning. Because Huber loss function is introduced into the quantile regression loss function.

Background

limitations of C51 and

There are two main drawbacks of C51.

The first factor is the projection process included complex computation. It is an essential but rather complex process to correct support in the process of calculating target distribution. In contrast, QR-DQN uses a technique that fixes probabilities and estimates support designated as quantile.

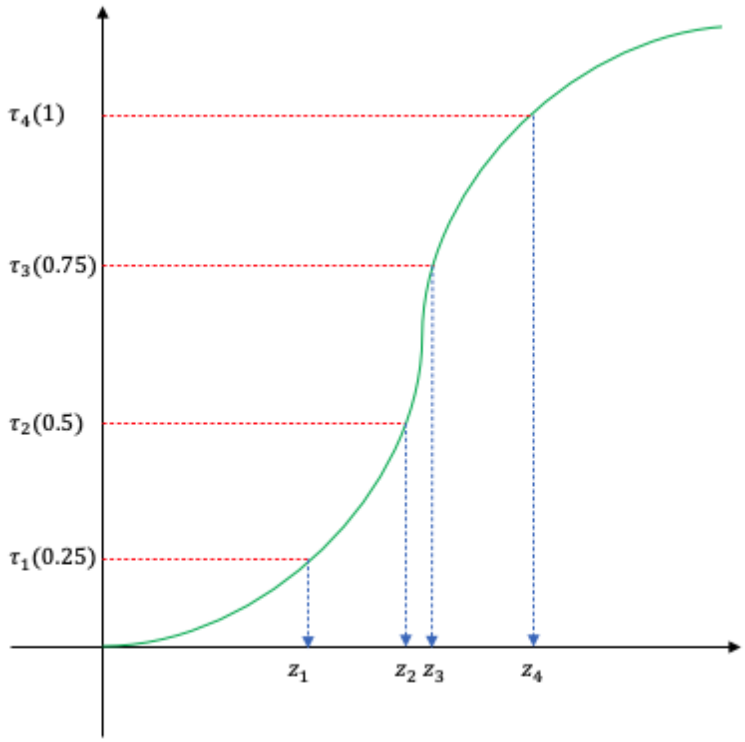
The second factor is that cross-entropy, which is used as a distance metric in C51, cannot guarantee convergence because it does not satisfy the gamma-contraction condition. To solve this problem, QR-DQN uses Wasserstein distance as metric, because Wasserstein distance is proved its property of convergence according to satisfy gamma-contraction.

Method

Quantile regression

If C51 conducts learning to estimate the optimal probability of each support, QR-DQN learns to find the optimal value of support based on the pre-selected probability. At that time, quantile is used as selected probability commonly, so this technique is called quantile regression.

First, quantile regression is a method applied based on CDF(Cumulative Distribution Function), so you need to know about CDF. CDF is literally a function in which probabilities for a supports are accumulated and it may be represented as a graph as follows.



Here, C51 compute τ by estimating support z as a domain, but quantile regression estimates support z based on proportion τ. This is expressed by equation as follows.

$$C51 \quad \tau_i = F_Z(z_i) \tag{1}$$

$$QR \quad z_i = F_Z^{-1}(\tau_i) \tag{2}$$

Then, why quantile regression method is used instead of the method used on C51? It is related with Wasserstein metric.

Wasserstein metric

Unlike the cross-entropy used in C51, Wasserstein metric satisfies gamma-contraction. The equation of Wasserstein is as follows,

$$W_p(U,Y) = (\int_0^1 |F_Y^{-1}(\omega) - F_U^{-1}|^p d\omega)^{\frac{1}{p}}$$

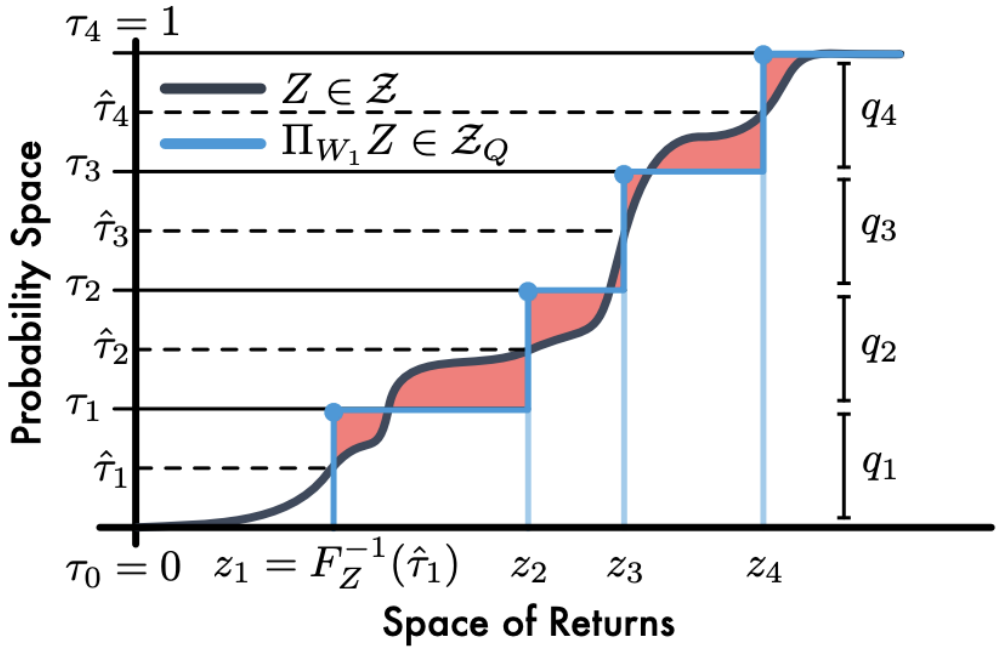
where U and Y are probability distribution and F is cumulative distribution. In the paper, 1-Wasserstein is used.

Unique minimizer

Wasserstein metric introduced earlier used to calculate the difference between target distribution and predicted distribution based on the reference probability called quantile. It should be noted that the probability chosen in this process is the median value of quantile called ‘Unique minimizer’, not just quantile. Refer the equation as follows.

$$unique\ minimizer\ \hat{\tau}_i = \frac{2(i-1)+1}{2N},\ (i = 1, 2, ..., N)$$

According to the next figure, it is easy to understand why these unique minimizers are used instead of quantile.



In the figure, the red area is the difference between the two distributions which is calculated based on unique minimizer $\hat{\tau}$. If it is calculated based on a general quantile τ other than unique minimizer $\hat{\tau}$, the territory of the red section increases. This means that when learning based on quantile τ proceeds, the difference is larger than when learning based on unique minimizer $\hat{\tau}$. Therefore, learning based on unique minimizer $\hat{\tau}$ makes it possible to closer estimate the target distribution.

Quantile regression loss applied Huber loss

At learning by quantile regression, Network makes effort to reduce the difference between target distribution and predicted distribution. Simultaneously the network learns to output supports of a size proportional to quantile.

Reducing the loss with target distribution is a common goal for most loss functions. Then why should network learn to output supports proportional to quantiles? Because the basic premise of quantile regression is that it is an operation based on CDF.

CDF takes the monotonous form of increase. If this assumption is not established, quantile regression learning based on CDF is also not available. Therefore, the equation of quantile regression(QR) is as follows.

$$QR = \sum_{i=1}^{N} \mathbb{E}_j[\rho_{\hat{\tau}_i}(\tau\theta_j - \theta_i(x,a))] \tag{3}$$

$$\tau\theta_j \leftarrow r + \gamma\theta_j(x^{'}, a^{*}) \tag{4}$$

$$N: \ Number\ of\ quantiles \tag{5}$$

$$\rho_t(u) = \begin{cases} u(r-1) & if\ u < 0 \\ u(\tau) & if\ u \geq 0 \end{cases} \tag{6}$$

To calculate quantile regression loss, the following three processes must be performed.

1. Calculate the difference between target supports and predicted supports.

Target=[2,4,8] Prediction=[1,4,5]

Target Prediction Error($\tau\theta_j - \theta_i(x,a)$)

$$\begin{bmatrix} 2 & 2 & 2 \\ 4 & 4 & 4 \\ 8 & 8 & 8 \end{bmatrix} - \begin{bmatrix} 1 & 4 & 5 \\ 1 & 4 & 5 \\ 1 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 1 & -2 & -3 \\ 3 & 0 & -1 \\ 7 & 4 & 3 \end{bmatrix}$$

On above example, quantile τ is $[0.33,\ 0.66,\ 1]$ and unique minimizer u is $[0.165,\ 0.495,\ 0.83]$ since the number of quantiles are 3.

2. Multiply the positive and negative outcomes by τ or (τ-1), respectively.

After that, multiply proportion by the element-wise product on error computed by process 1 as follows.

Target=[2,4,8] Prediction=[1,4,5]

Target Prediction Error($\tau\theta_j - \theta_i(x,a)$)

$$\begin{bmatrix}2 & 2 & 2\\ 4 & 4 & 4\\ 8 & 8 & 8\end{bmatrix} - \begin{bmatrix}1 & 4 & 5\\ 1 & 4 & 5\\ 1 & 4 & 5\end{bmatrix} = \begin{bmatrix}1 & -2 & -3\\ 3 & 0 & -1\\ 7 & 4 & 3\end{bmatrix}$$

Unique minimizer=[0.165, 0.495, 0.83]

Error($\tau\theta_j - \theta_i(x,a)$) Unique minimizer $p\hat{\tau}_i(\tau\theta_j - \theta_i(x,a))$

$$\begin{bmatrix}1 & -2 & -3\\ 3 & 0 & -1\\ 7 & 4 & 3\end{bmatrix} - \begin{bmatrix}0.165 & 0.495-1 & 0.83-1\\ 0.165 & 0.495 & 0.83-1\\ 0.165 & 0.495 & 0.83\end{bmatrix} = \begin{bmatrix}0.165 & 1.01 & 0.51\\ 0.495 & 0 & 0.17\\ 1.155 & 1.98 & 2.49\end{bmatrix}$$

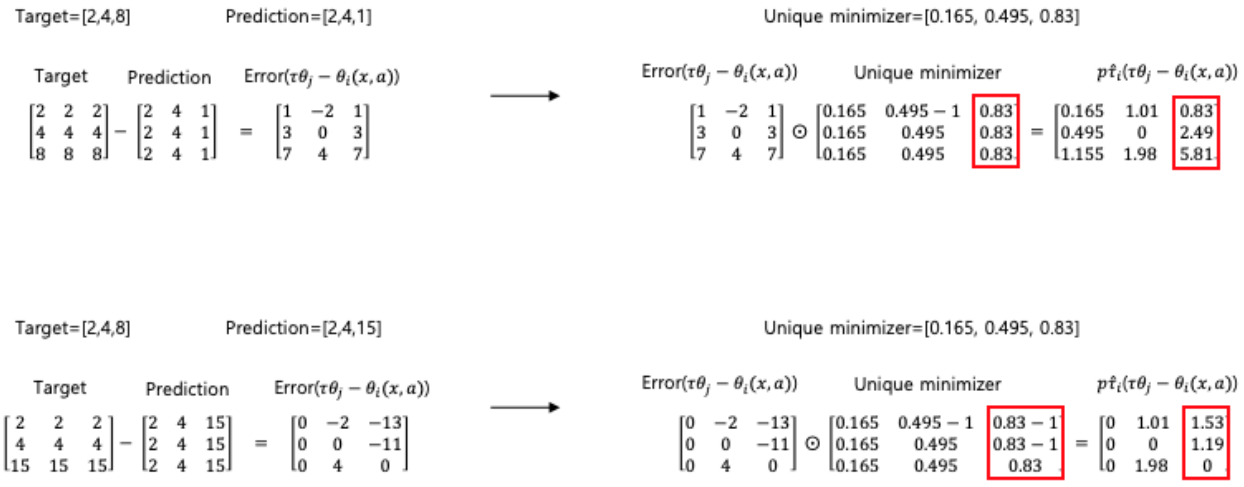
3. Calculate the quantile regression loss by averaging target values after summing predicted values.

$p\hat{\tau}_i(\tau\theta_j - \theta_i(x,a))$ Prediction summation Target average

$$\begin{bmatrix}0.165 & 1.01 & 0.51\\ 0.495 & 0 & 0.17\\ 1.155 & 1.98 & 2.49\end{bmatrix} \longrightarrow \begin{bmatrix}1.685\\ 0.665\\ 5.625\end{bmatrix} \longrightarrow [2.685]$$

Finally, quantile regression loss = 2.685 is derived.

There is one thing to note among the calculations shown earlier. It is that the structure of the quantile regression loss is designed to give more penalties to elements that break down the CDF's assumption, leading them to learn support proportional to quantile. This can be found in figure as follows.



At the above case, the rule of CDF are ignored. Because the specific quantile support prediction is less than the previous quantile support. Accordingly, the violation of the CDF's basic rule like this causes a larger loss value.

Quantile Huber loss

Quantile regression loss described above cannot be used for deep learning gradient operation as it is. Because there is a point that cannot be differentiated according to the discontinuity of the differential coefficient. Therefore, Huber loss is added to solve this problem by smoothing it. Equation of Huber loss is as follows.

$$\mathcal{L}_k(u) = \begin{cases} \frac{1}{2}u^2, & if\ |u| \le k \\ k(|u| - \frac{1}{2}k) & otherwise \end{cases}$$

Based on this, change the ρ as follows. The reason $(\tau - 1)$ changed to $(1 - \tau)$ is to make the loss positive.

$$\rho_\tau(u) = \begin{cases} u(\tau - 1) & if\ u < 0 \\ u(\tau) & if\ u \geq 0 \end{cases}$$

$$\downarrow$$

$$\rho_\tau(u) = \begin{cases} \mathcal{L}_k(u)(1 - \tau) & if\ u < 0 \\ \mathcal{L}_k(u)(\tau) & if\ u \geq 0 \end{cases}$$

Quantile Huber loss can be calculated by modifying the part of ρ as above and applying it to the quantile regression loss at the bottom.

$$\sum_{i=1}^{N} \mathbb{E}_j[\rho_{\hat{\tau}_i}^k(\tau\theta_j - \theta_i(x, a))]$$

Algorithm

Algorithm 1 Quantile Regression Q-Learning

Require: N, κ

input $x, a, r, x', \gamma \in [0, 1)$

Compute distributional Bellman target

$Q(x', a') := \sum_j q_j \theta_j(x', a')$

$a^* \leftarrow \arg\max_{a'} Q(x, a')$

$\mathcal{T}\theta_j \leftarrow r + \gamma\theta_j(x', a^*), \quad \forall j$

Compute quantile regression loss (Equation 10)

output $\sum_{i=1}^{N} \mathbb{E}_j \left[\rho_{\hat{\tau}_i}^{\kappa}(\mathcal{T}\theta_j - \theta_i(x, a))\right]$

Implementation on JORLDY

- QR-DQN 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 prediction supports matrix
```

```
# 2. Calculate target supports matrix
# 3. Calculate Quantile Huber loss and update network

def learn(self):
...

    # Get Theta Pred
    logit = self.network(state)
    logits, q_action = self.logits2Q(logit)
    action_eye = torch.eye(self.action_size, device=self.device)
    action_onehot = action_eye[action.long()]

    theta_pred = action_onehot @ logits

    with torch.no_grad():
        # Get Theta Target
        logit_next = self.network(next_state)
        _, q_next = self.logits2Q(logit_next)

        logit_target = self.target_network(next_state)
        logits_target, _ = self.logits2Q(logit_target)

        max_a = torch.argmax(q_next, axis=-1, keepdim=True)
        max_a_onehot = action_eye[max_a.long()]

        theta_target = reward + (1 - done) * self.gamma * torch.squeeze(
            max_a_onehot @ logits_target, 1
        )
        theta_target = torch.unsqueeze(theta_target, 2)

    error_loss = theta_target - theta_pred
    huber_loss = F.smooth_l1_loss(
        *torch.broadcast_tensors(theta_pred, theta_target), reduction="none"
    )

    # Get Loss
    loss = torch.where(error_loss < 0.0, self.inv_tau, self.tau) * huber_loss
    loss = torch.mean(torch.sum(loss, axis=2))

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

    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)
    q_action = torch.mean(_logits, dim=-1)
    return _logits, q_action
```