IQN(Implicit Quantile Network)

Paper Link: https://arxiv.org/pdf/1806.06923.pdf

Key Features

- IQN is a QR(Quantile Regression)-based distributional RL algorithm that estimates the distribution of return based on random sampling with respect to sample τ (quantile).
- IQN uses ϵ-greedy policy based on Risk-sensitive RL principle, so that it can ensure better performance.
- IQN is an algorithm improved by applying concepts of distributional RL and Risk-sensitive RL to DQN.

Background

Limitation of QR-DQN by using discrete quantile

QR-DQN approximately learns the return distribution using a predetermined quantile. In comparison with C51, QR-DQN reduces the complexity of parameter adjustment and eliminates the projection process. Additionally, QR-DQN uses Wasserstein metric as distance metric which satisfy gamma-contraction. So, QR-DQN can guarantee convergence. However, QR-DQN should fix these quantiles in advance. Approximation based on these fixed discrete quantiles has limitations in approximation performance for continuous distribution.

At first, approximation error can be controlled only by the number of specified quantiles. This control method has a limitation in approximate performance, because the diversity of samples used in learning is limited.

The second is an inflexible policy choice. Q-functions along the actions may has various identified distribution. When agent depends on specified policy, it is impossible to cope with these distributions approximately. Various policies may be required to appropriately respond to these distributions, but they do not exist in QR-DQN.

Method

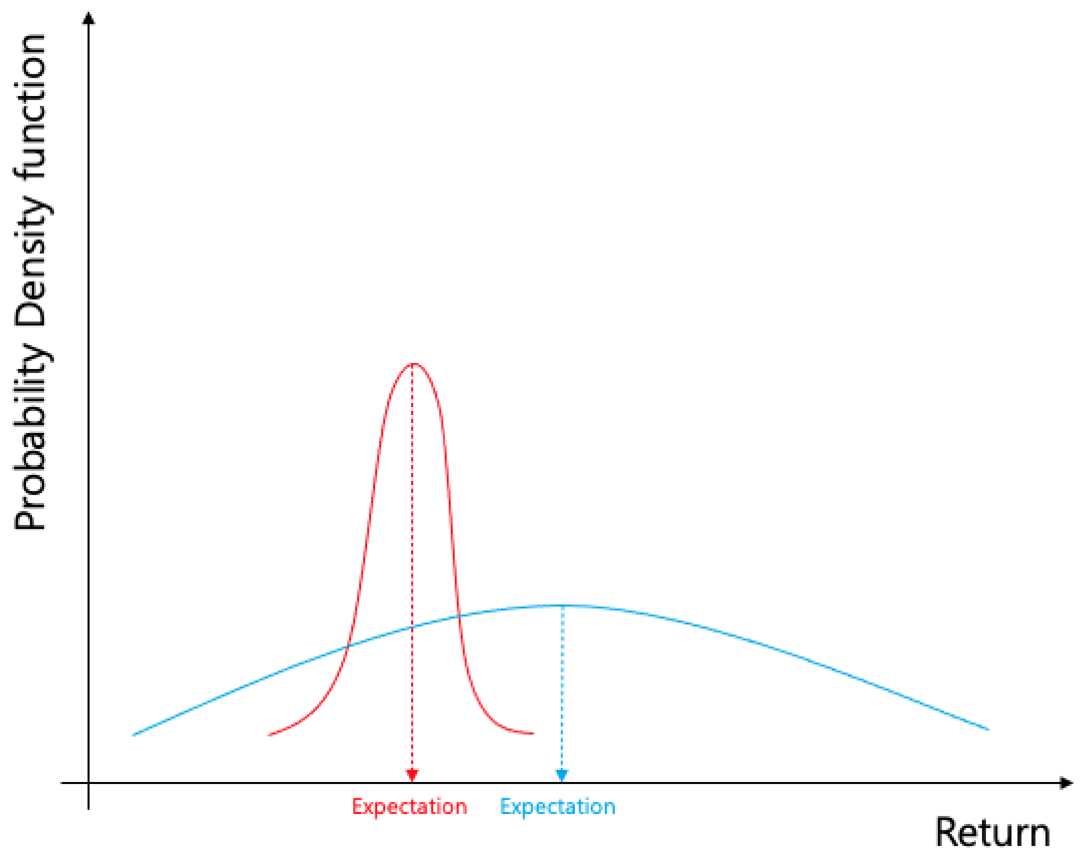
Control approximate error depending on network size and training steps

IQN learns return distribution based on randomly sampled probabilities τ each time. The method of sampling τ specifies the number of samples in advance and then agent extracts randomly in range of $U([0, 1])$ for each update step. Accordingly the approximation errors which exist chronically in QR-DQN algorithm can be removed effectively.

ϵ-greedy policy adoption based on Risk-sensitive RL

IQN chooses policy based on the Q-function which is applied Risk-sensitive RL concept. Then, what is Risk-sensitive RL?

Risk-sensitive RL is one of the reinforcement learning method that is learned by changing the criteria for action selection based on the coefficient β that determines the preference for risk. Then, what is Risk in RL? Look at the graph as follows.



The expected return of the red probability distribution is lower than it of the blue probability distribution, but the variability is low due to the small variance. On the other hand, blue has a high expected return but a high variance, so there is a high variability in receiving the expected return.

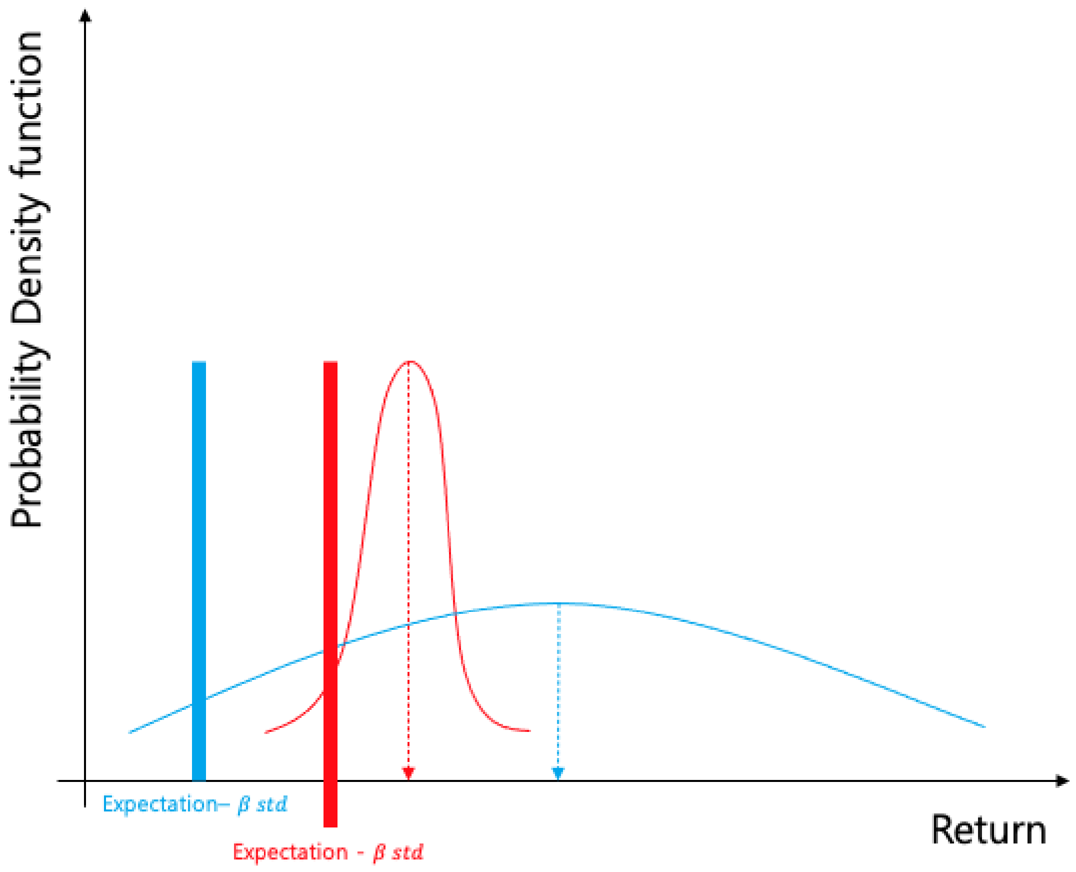
Risk, which is variability expressed as variance, is a measure of uncertainty. Risk-sensitive RL uses these risk to calculate uncertainty, not just expected return.

Risk-sensitive RL makes agent select policies with specific tendency for these risk in the learning process. There are two types of the tendency.

The first is the Risk-averse policy. Its action selection equation is as follows.

$$a^* = argmaxE(R(s, a_i)) - \beta std(R(s, a_i))$$

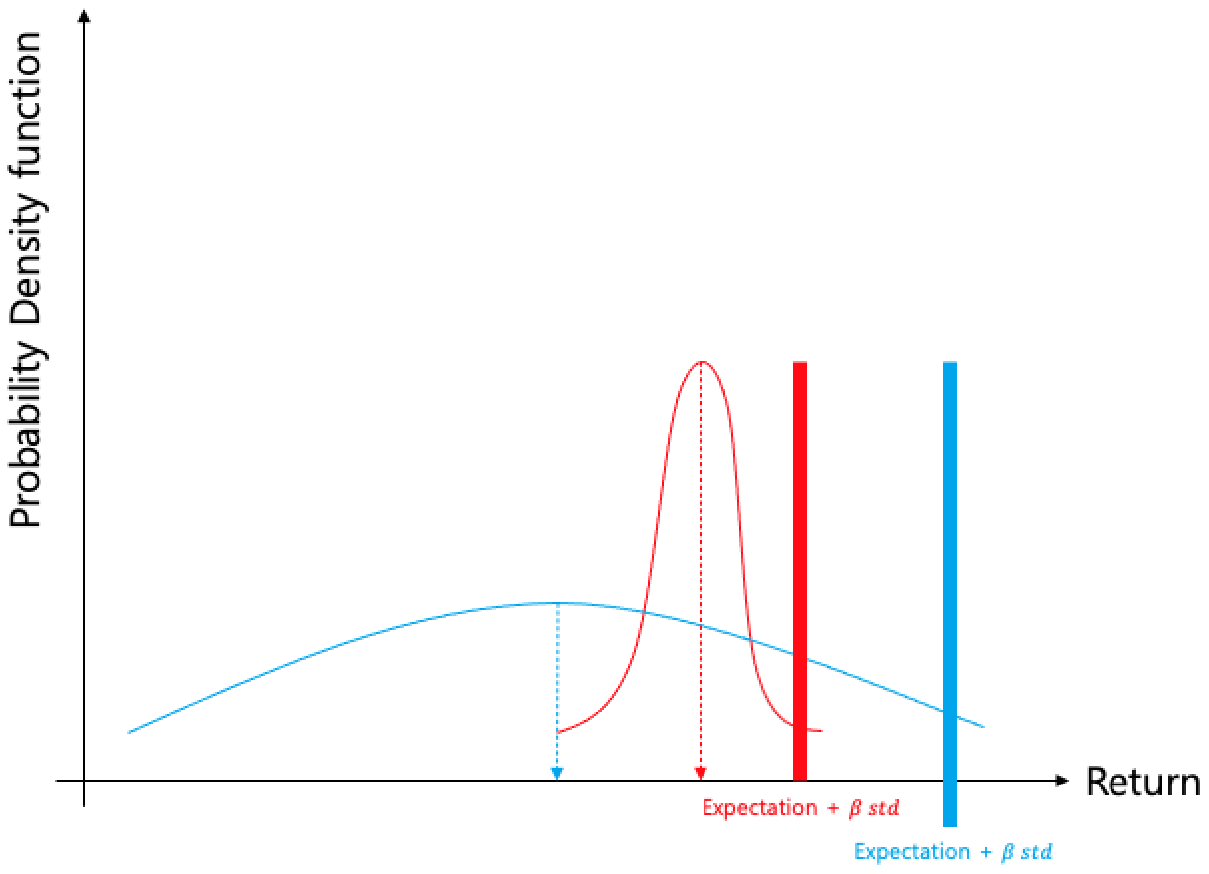
In the equation, the risk term $std(R(s, a_i))$ acts as a penalty for the expected return. Therefore, The final expected return for the blue graph is lower than the red one and then the result makes agent selects action for reds.



The second is the Risk-seeking policy. Its action selection equation is as follows.

$$a^* = argmaxE(R(s, a_i)) + \beta std(R(s, a_i))$$

The difference from Risk-averse policy is the sign of β. While Risk-averse policy uses risk as a penalty, Risk-seeking policy adopt the risk as an advantage. It can make agent aim for high return with high risk. Accordingly, agent selects the action whose distribution has high variance with respect to reward expectation as blue graph.


Probability Density function
Return
Expectation + β std
Expectation + β std

β is used as a predefined value within the range $U([0,1])$ range. So, how this β is determined? The paper presents four methods as follows.

- CPW(Cumulative Probability Weighting) parameterization
- Wang
- POW(POWer formula)

- CVaR(Conditional Value-at-Risk)

For more information about these methods, please refer to the paper. For reference, the most commonly used method in RL is CVaR. It has the same effect as adjusting β by adjusting the sample range $U([0,1])$ to $U([0,\eta])$ for sample τ by specified point η.

Apply Risk-sensitive RL on IQN

IQN uses the Q-function that a predetermined β is applied. Under the this Q-function(1), policy can also be expressed as equation(2).

$$Q_\beta(x,a) = \mathop{\mathbb{E}}_{\tau\sim U([0,1])}[Z_{\beta(\tau)}(x,a)] \tag{1}$$

$$\pi_\beta = \mathop{argmax}_{a\in\mathcal{A}} Q_\beta(x,a) \tag{2}$$

For two samples Q-function(τ), target Q-function(τ')$\sim U$([0,1]), and policy π_β, the sampled TD(Temporal Difference) error at step t is as follows.

$$\delta_t^{\tau,\tau'} = r_t + \gamma Z_{\tau'}(x_{t+1}, \pi_\beta(x_{t+1})) - Z_\tau(x_t, a_t)$$

Then, the IQN loss function is given as follows. It is Quantile Huber loss which used at QR-DQN loss calculation.

$$\mathcal{L}(x_t, a_t, r_t, x_{t+1}) = \frac{1}{N'}\sum_{i=1}^{N}\sum_{j=1}^{N'}\rho_{\tau_i}^k(\delta_t^{\tau_i,\tau_j'})$$

where N and N' is respectively number of IID(identically independent distribution) samples $\tau_i, \tau'_j \sim U([0,1])$ used to estimate the loss. Finally, sample-based Risk-sensitive policy can be obtained as follows.

$$\tilde{\pi}_\beta(x) = \underset{a \in \mathcal{A}}{argmax} \frac{1}{K} \sum_{k=1}^{K} Z_{\beta(\tilde{\tau}_k)}(x,a), \quad \tilde{\tau}_k \sim \beta(\cdot)$$

Implementation of IQN

The basic network structure of IQN is the same as DQN. The only difference is to add embedding function to embed samples τ.

$$DQN = Q(x,a) \approx f(\psi(x))_a \tag{3}$$

$$IQN = Z_\tau(x,a) \approx f(\psi(x) \odot \phi(\tau))_a \tag{4}$$

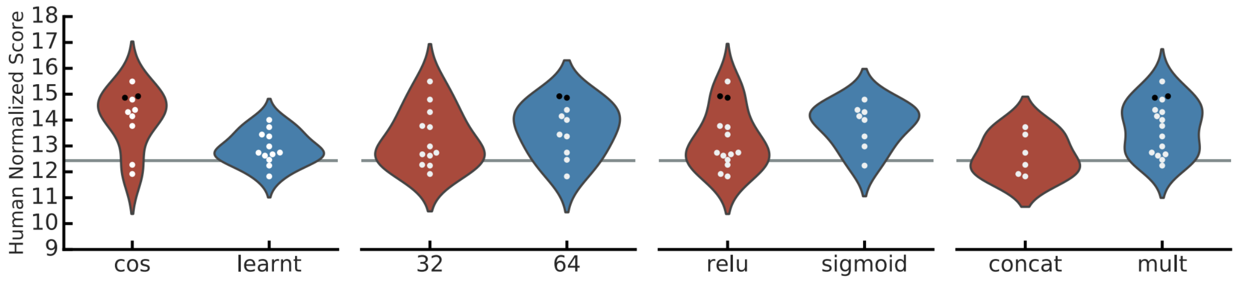
IQN multiplies the convolution function output(ψ) and the embedded function(ϕ) output by the element-wise method. After that, fully connected layer(f) operate to output Q-function(Z)

Embedding function(ψ)

The embedding function used for IQN is commonly ‘N cosine basis function’ as follows.

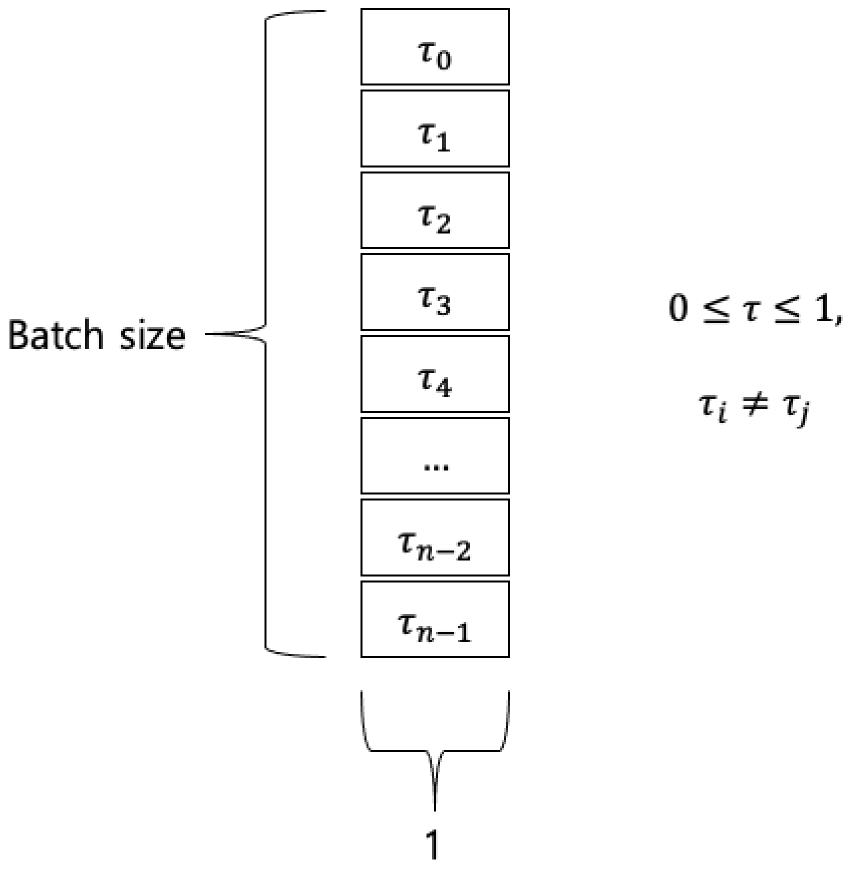
$$\phi_j(\tau) = ReLU(\sum_{i=0}^{n-1} cos(\pi i \tau) w_{ij} + b_j)$$

The reason why this function is selected commonly is that it has the best performance among several options.

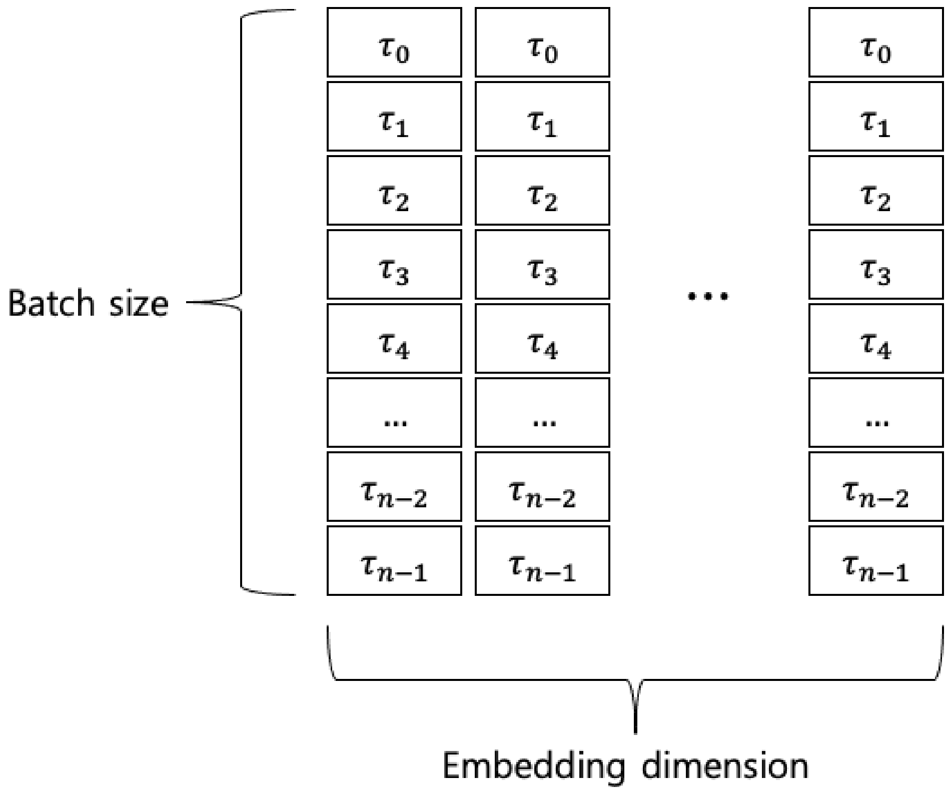


The calculation proceeds in the following procedure.

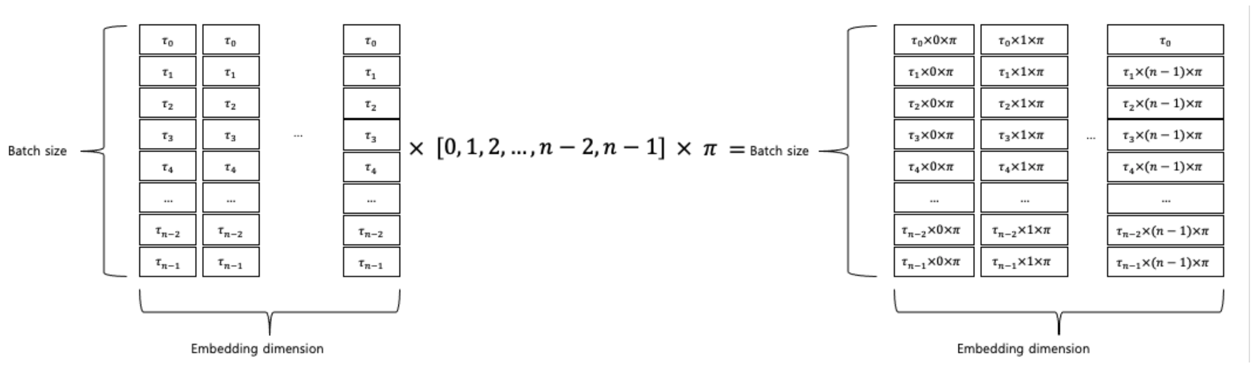
- Random sampling τ



- Broadcast to dimension of embedding(In paper, 64 is adopted)



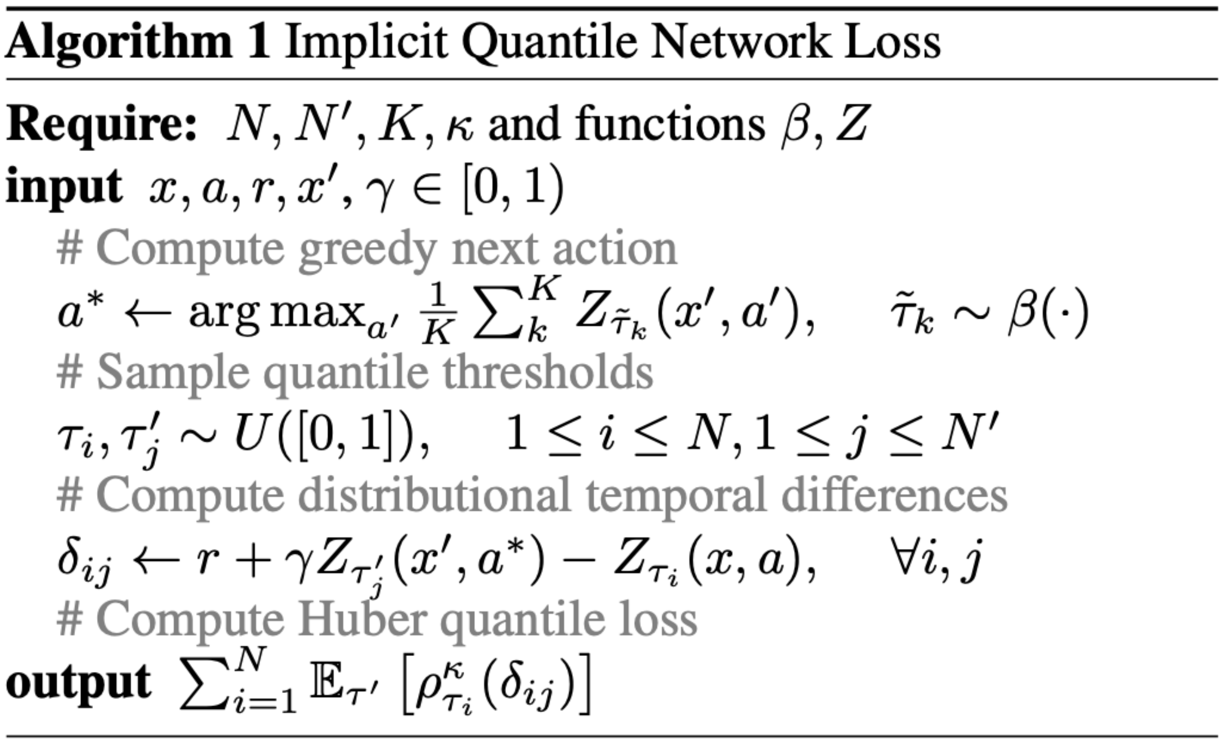
- Multiply i(0~n-1) and π



- Apply functions(Cosine, ReLU), weight w and bias b

$$ReLU(Cos(Matrix)) \times w + b$$

Algorithm



Algorithm 1 Implicit Quantile Network Loss

```
Require: $N, N', K, \kappa$ and functions $\beta, Z$
input $x, a, r, x', \gamma \in [0, 1)$
    # Compute greedy next action
    $a^* \leftarrow \arg\max_{a'} \frac{1}{K} \sum_k^K Z_{\tilde{\tau}_k}(x', a'), \quad \tilde{\tau}_k \sim \beta(\cdot)$
    # Sample quantile thresholds
    $\tau_i, \tau'_j \sim U([0, 1]), \quad 1 \leq i \leq N, 1 \leq j \leq N'$
    # Compute distributional temporal differences
    $\delta_{ij} \leftarrow r + \gamma Z_{\tau'_j}(x', a^*) - Z_{\tau_i}(x, a), \quad \forall i, j$
    # Compute Huber quantile loss
output $\sum_{i=1}^N \mathbb{E}_{\tau'} \left[\rho^\kappa_{\tau_i}(\delta_{ij})\right]$
```

Implementation on JORLDY

- IQN JORLDY Implementation

```
### act function ###
# Select action epsilon-greedy policy(random vs value distribuion)

@torch.no_grad()
def act(self, state, training=True):
    self.network.train(training)
    epsilon = self.epsilon if training else self.epsilon_eval
    sample_min = 0 if training else self.sample_min
    sample_max = 1 if training else self.sample_max
    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), sample_min, sample_max)
        _, q_action = self.logits2Q(logits)
        action = torch.argmax(q_action, -1, keepdim=True).cpu().numpy()
    return {"action": action}
```

```
### learn function ###
# 1. Calculate value distribution
# 2. Calculate target value distribution
# 3. Calculate loss(Quantile Huber)
# 4. Update network

def learn(self):
...

    # Get Theta Pred, Tau
    logit, tau = 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
    tau = torch.transpose(tau, 1, 2).contiguous()

    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, 1 - tau, 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()

...
```

```
### logidts2Q function ###
# Return transposed-logits and value distribution

def logits2Q(self, logits):
    _logits = torch.transpose(logits, 1, 2).contiguous()

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

References

Relevant papers

- A Comprehensive Survey on Safe Reinforcement Learning
  (J. Garcia, F. Fernandez, 2015)