Notes on Distributional Reinforcement Learning

Shijie Huang

January 3, 2021

1 Introduction

With a passion on distriubtional RL and to better study the algorithms, I implemented the main distributional RL algorithms (Categorical, Quantile and Expectile) [1]. This summary documents my understanding and thoughts, and hopefully it will be helpful to those who intend to understand and implement the algorithms.

While I have tried my best to use plain language and simplified process charts to illustrate the algorithms, reading the original paper is essential to fully understand the rationale behind them (especially the proofs). This summary is not mathematically sound and accurate.

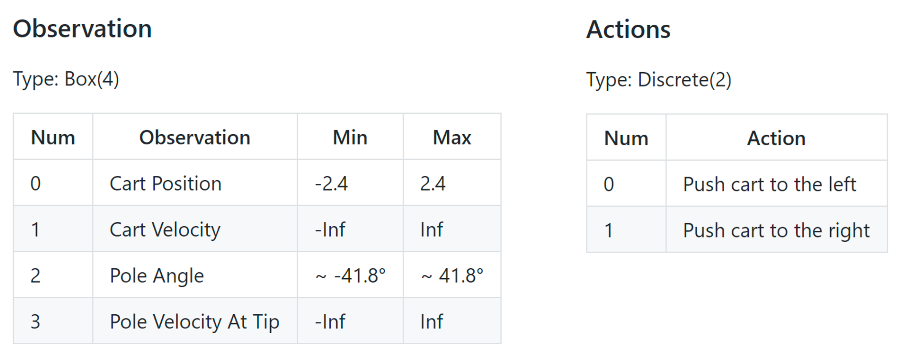
When I implement the algorithms, I think the hardest part is to get the shape of arrays/matrix correct, particularly when outcomes are processed in batches. Although implementations may vary from one to another, it is recommended that you draw a diagram with expected outcome shape attached to each process. In the diagrams I draw below, I included the outcome shape at each step to help understanding. I did not include the batches because it created unnecessary complications that would not help understand the algorithms.

This summary does not touch on the issue of state representation/neural network structure. I only pay attention to the RL part because choices of deep learning structure do not really contribute to the understanding of core distributional RL algorithms.

2 Environment

I use Cartpole (v0) as the environment. For more information, please visit the github wiki page. Each state (observation) is a (1, 4) vector with continuous values, and each action is a discrete scalar, either 0 or 1.

[1] see `https://github.com/hsjharvey/Reinforcement-Learning`



Observation

Type: Box(4)

Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -41.8°	~ 41.8°
3	Pole Velocity At Tip	-Inf	Inf

Actions

Type: Discrete(2)

Num	Action
0	Push cart to the left
1	Push cart to the right

3 Distributional RL

Figure 1 illustrates a typical process of a DQN. In a nutshell, the idea of distributional version is not sophisticated: given a (state, action), the network does not output action value directly as a single scalar value, but rather a distribution. In the figure, I highlighted the key steps where distributional RL are different from standard DQN.

The main structure of the distributional version remains the same, e.g. two networks, TD updates, etc. The main challenges are the following:

1. What distribution(s) should we choose?
2. How do we apply the concept of value distribution in the general RL framework?

The first question is more general than reinforcement learning itself. There are two general ways to approach it: (1) assume some known functional form with meta-parameters (e.g. Gaussian, student-t, etc.) (2) approximation via statistical properties. The pros and cons of each approach are a topic of another field, but it is not too hard to see that in a neural network setting, approximation is the default method. So the question pins down to the following: how do we approximate/characterise a distribution?

A distribution has certain statistics to characterise itself. In plain language, if we know some statistical properties of a distribution, we can roughly know what this distribution ”looks like”. A typical example is histogram: a probability distribution can be characterised by equal-sized bins with different heights (probabilities or frequencies). Thus, if we know those characteristics, we will have a distribution. I will unpack different approximation statistics in the following sections.

The second question is more involved. In the context of RL, replacing the scalar action value with a distribution raises more questions. For instance, how do we ”choose a distribution” to act upon? In standard DQN, we can follow a specific policy, and compare two scalar values regardless how they are computed. However, in distributional RL, it is not obvious how to ”compare distributions”.

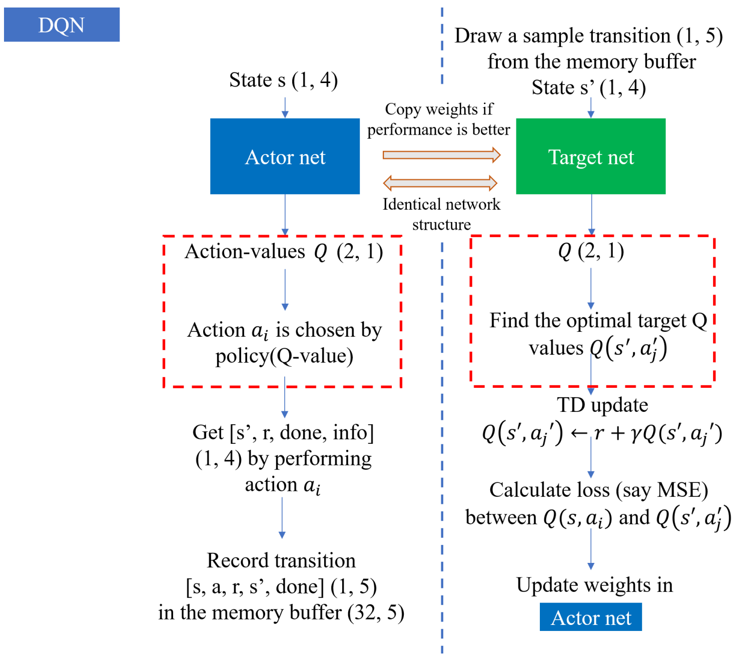

DQN
Draw a sample transition (1, 5) from the memory buffer
State s (1, 4)
State s' (1, 4)
Copy weights if performance is better
Actor net
Target net
Identical network structure
Action-values Q (2, 1)
Q (2, 1)
Action a_i is chosen by policy(Q-value)
Find the optimal target Q values $Q(s', a_j')$
TD update
$Q(s', a_j') \leftarrow r + \gamma Q(s', a_j')$
Get [s', r, done, info] (1, 4) by performing action a_i
Calculate loss (say MSE) between $Q(s, a_i)$ and $Q(s', a_j')$
Record transition [s, a, r, s', done] (1, 5) in the memory buffer (32, 5)
Update weights in
Actor net

Figure 1: Deep Q Learning.

Moreover, it is also not obvious that we can simply "update a distribution" so that it converges to the target distribution, just like what we do in standard DQN with a single scalar value. Although the actual implementation varies from algorithms to algorithms, it is important to have this first layer question in mind when we study the algorithms.

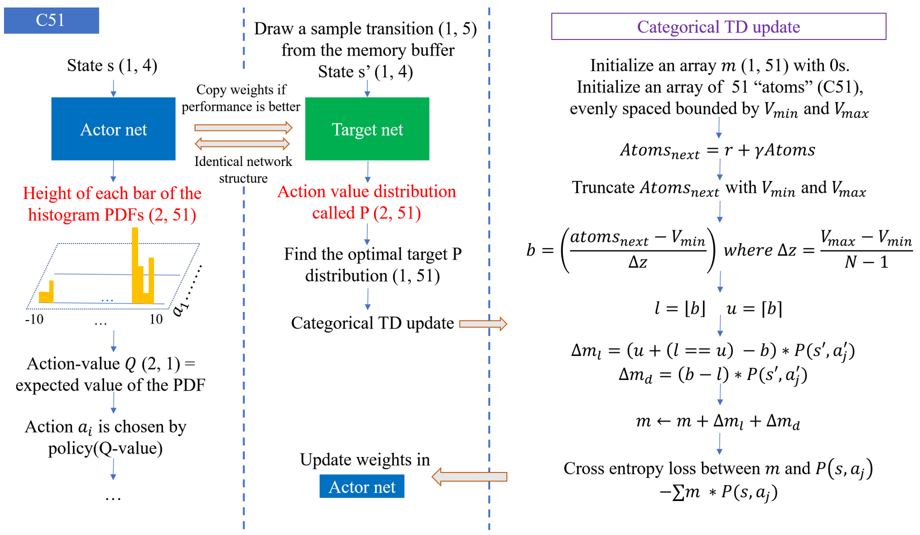
4 Categorical [1]

In categorical RL, the choice of distribution approximation is histogram. To characterise a histogram, we need two arrays: (1) bins (2) corresponding height (frequency). In categorical RL, we assume that the Q-values lie within a boundary, $[V_{min}, V_{max}]$; we also assume that there are $n = 50$ equal sized (ΔZ) bins. There are in total 51 points (called atoms) on the x-axis (hence the name C51). The corresponding frequencies are the output of neural networks.

In categorical RL, we compare actions by comparing the expected value of its PDF. Notice that this expected value is the mean action-value (Q-value).

To perform a TD update, we first shrink (γ) and move $(reward)$ the atoms (x-axis), then we re-distribute the frequencies back to where the original boundaries. This process is illustrated clearly in the original paper (see figure 3).

To see how it works in an atari game, one can visit `https://www.youtube.com/watch?v=vIz5P6s80qA`.



C51
State s (1, 4)
Actor net
Height of each bar of the histogram PDFs (2, 51)
-10
…
10
a_1
Action-value Q (2, 1) = expected value of the PDF
Action a_i is chosen by policy(Q-value)
…
Copy weights if performance is better
Identical network structure
Draw a sample transition (1, 5) from the memory buffer
State s' (1, 4)
Target net
Action value distribution called P (2, 51)
Find the optimal target P distribution (1, 51)
Categorical TD update
Update weights in Actor net
Categorical TD update
Initialize an array m (1, 51) with 0s.
Initialize an array of 51 "atoms" (C51), evenly spaced bounded by V_{min} and V_{max}
$Atoms_{next} = r + \gamma Atoms$
Truncate $Atoms_{next}$ with V_{min} and V_{max}
$b = \left(\frac{atoms_{next} - V_{min}}{\Delta z}\right)$ where $\Delta z = \frac{V_{max} - V_{min}}{N-1}$
$l = \lfloor b \rfloor \quad u = \lceil b \rceil$
$\Delta m_l = (u + (l == u) - b) * P(s', a_j')$
$\Delta m_d = (b - l) * P(s', a_j')$
$m \leftarrow m + \Delta m_l + \Delta m_d$
Cross entropy loss between m and $P(s, a_j)$
$-\sum m * P(s, a_j)$

Figure 2: Categorical DQN (C51)

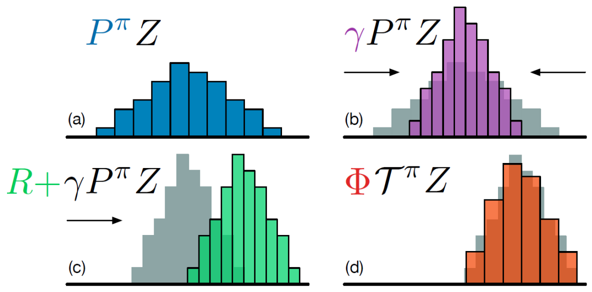


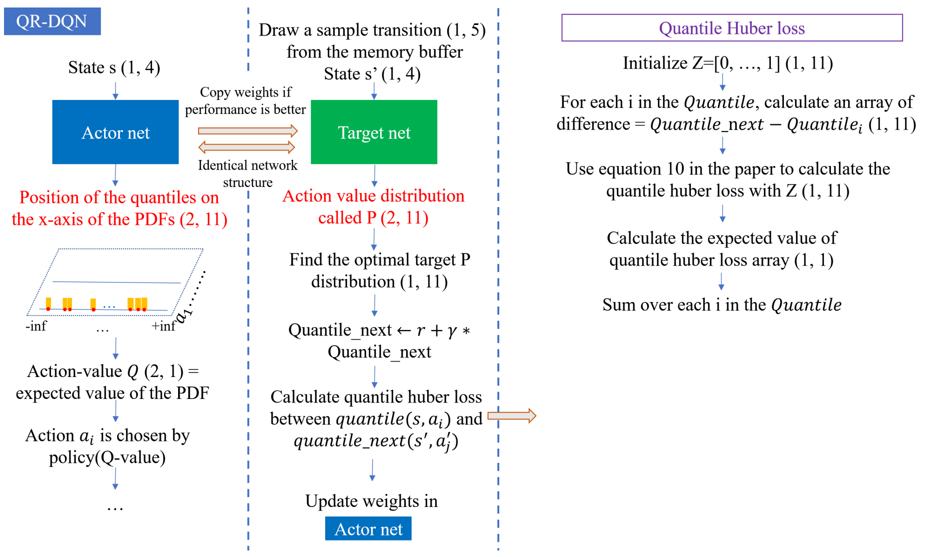
Figure 3: Distribution Bellman Operator [1]

5 Quantile [4]

Another way of approximation is via quantiles. Instead of assuming equal-spaced bins on the x-axis, we assume equal-spaced values on the y-axis. The neural network will output the position of the ith quantile on the x-axis. By further assuming that each position has the same height, we will have an approximated value distribution. Notice that the density of a distribution is not represented by "how tall" a bin is, but rather how "clustered" those equal height bins are. Consequently, it makes more sense to visualize the values in CDF rather than PDF.

In the quantile regression DQN, we also compare actions by comparing the expected value of its PDF. By assuming equal height, the mean action value is simply the mean of all quantile values.

Unlike the seemly sophisticated TD update process in Categorical DQN, the



QR-DQN
State s (1, 4)
Actor net
Copy weights if performance is better
Identical network structure
Target net
Draw a sample transition (1, 5) from the memory buffer
State s' (1, 4)
Position of the quantiles on the x-axis of the PDFs (2, 11)
-inf
…
+inf
a_1
Action-value Q (2, 1) = expected value of the PDF
Action a_i is chosen by policy(Q-value)
…
Action value distribution called P (2, 11)
Find the optimal target P distribution (1, 11)
Quantile_next ← $r + \gamma *$ Quantile_next
Calculate quantile huber loss between $quantile(s, a_i)$ and $quantile\_next(s', a_j')$
Update weights in
Actor net
Quantile Huber loss
Initialize Z=[0, …, 1] (1, 11)
For each i in the $Quantile$, calculate an array of difference = $Quantile\_next - Quantile_i$ (1, 11)
Use equation 10 in the paper to calculate the quantile huber loss with Z (1, 11)
Calculate the expected value of quantile huber loss array (1, 1)
Sum over each i in the $Quantile$

Figure 4: QR-DQN

TD update of QR-DQN is relatively straightforward. Quantile values do not have boundary assumptions. As a result, to perform distribution update, we can simply perform TD update on quantile values. What's left is to find an appropriate loss function to calculate the loss between quantile values from the actor network and the target quantile values from the target network.

To see how the quantile method works in an atari game, one can visit https://www.youtube.com/watch?v=zdh_BTOcVYs. Be aware that the implemented algorithm is Implicit Quantile Network (IQN), which is an improved version, see [3] for more details.

6 Expectile [2]

The expectile method is very similar to the quantile method. We also assume equal-spaced bins on the y-axis. However, different from the quantile method, we say that each value on the x-axis that corresponds to the ith value on the y-axis is the ith expectile value. The neural network will output the position of the ith expectile on the x-axis. Again by further assuming that each position has the same height, we will have an approximated value distribution.

In the expectile regression DQN, we compare actions by comparing the middle expectile because the middle expectile is the mean.

Similar to the QR-DQN, we can perform TD update directly on expectile values so that the expectile values are closer to the target expectile (i.e. shifting the CDF towards the target CDF by moving the x-axis values). However, this naive method turns out to be problematic in the expectile setting (see section

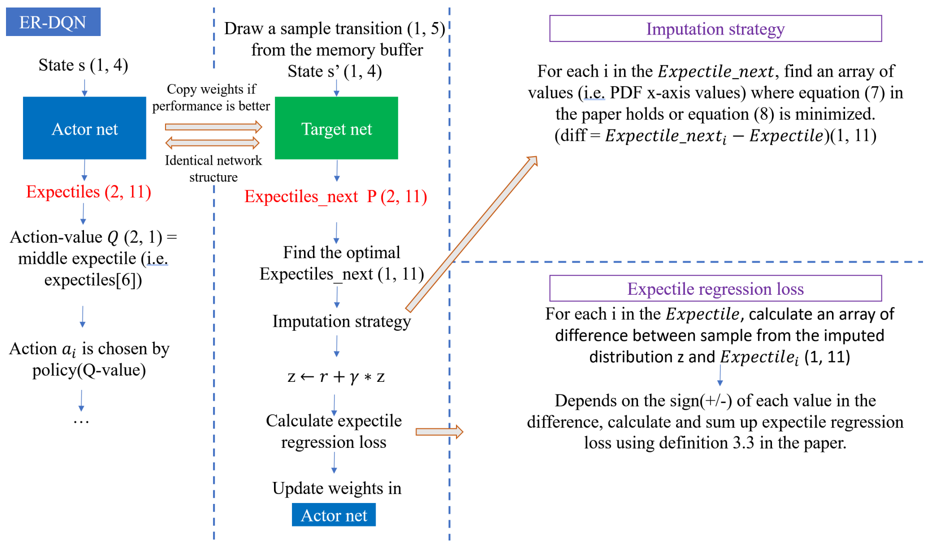

ER-DQN
State s (1, 4)
Actor net
Expectiles (2, 11)
Action-value Q (2, 1) = middle expectile (i.e. expectiles[6])
Action a_i is chosen by policy(Q-value)
…
Copy weights if performance is better
Identical network structure
Draw a sample transition (1, 5) from the memory buffer
State s' (1, 4)
Target net
Expectiles_next P (2, 11)
Find the optimal Expectiles_next (1, 11)
Imputation strategy
$z \leftarrow r + \gamma * z$
Calculate expectile regression loss
Update weights in
Actor net
Imputation strategy
For each i in the $Expectile\_next$, find an array of values (i.e. PDF x-axis values) where equation (7) in the paper holds or equation (8) is minimized.
(diff = $Expectile\_next_i - Expectile$)(1, 11)
Expectile regression loss
For each i in the $Expectile$, calculate an array of difference between sample from the imputed distribution z and $Expectile_i$ (1, 11)
Depends on the sign(+/-) of each value in the difference, calculate and sum up expectile regression loss using definition 3.3 in the paper.

Figure 5: ER-DQN

3.1 in the original paper).

A better approach is to re-generate the entire distribution from statistics (i.e. quantile or expectile), and then perform TD update on the re-generated distribution. This process of re-generating distribution from statistics is called an "imputation strategy".

After the target network outputs target expectile values, we want to find a distribution in which its expectile matches the target expectile values (root-finding problem) or minimises the expectile-regression loss, given the target expectile values (minimization problem).

We will then perform a TD update on the target distribution (samples). The final step is to find weights in the actor network so that its output expectiles minimise the expectile regression loss, given the update target distribution.

7 General Discussion and Others

After studying the three algorithms carefully, I realize that in a nutshell, distributional RL intends to learn action value better via distributions. The system can be complicated and fragile. However, if designed and implemented properly, distributonal RL can learn much richer representations and thus lead to more accurate action values.

In my opinion, categorical RL is the easiest to understand. However, the obvious disadvantage is the complicated process to perform the Bellman update. Additionally, choices of V_{min}, V_{max} and the number of atoms really matter. One has to roughly know where the true action value is to be able to carry out a successful training.

Although in QR-DQN we still have to assume the number of quantiles, we do not necessarily have to know where the true action value is. However, we may have to be a bit careful when designing the loss function, as stated in the original paper.

The key to a successful ER-DQN algorithm is the imputation strategy. In practice, training could be problematic if the root finding (or the minimization) is not successful. Plus, training time can be significantly longer than the categorical method and QR-DQN due to the time spent on imputing distributions.

Finally, as I mentioned at the beginning, this notes focus on the RL part. An appropriate state representation, or network structure, is also crucial to a successful training, particularly when it comes to complicated environment.

References

[1] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 449–458. JMLR.org, 2017.

[2] Marc G Bellemare, Nicolas Le Roux, Pablo Samuel Castro, and Subhodeep Moitra. Distributional reinforcement learning with linear function approximation. *arXiv preprint arXiv:1902.03149*, 2019.

[3] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, 2018.

[4] Will Dabney, Mark Rowland, Marc G Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[5] Will Dabney, Zeb Kurth-Nelson, Naoshige Uchida, Clara Kwon Starkweather, Demis Hassabis, Rémi Munos, and Matthew Botvinick. A distributional code for value in dopamine-based reinforcement learning. *Nature*, pages 1–5, 2020.

[6] Clare Lyle, Pablo Samuel Castro, and Marc G Bellemare. A comparative analysis of expected and distributional reinforcement learning. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 2019.

[7] Mark Rowland, Marc G Bellemare, Will Dabney, Rémi Munos, and Yee Whye Teh. An analysis of categorical distributional reinforcement learning. *arXiv preprint arXiv:1802.08163*, 2018.

[8] Mark Rowland, Robert Dadashi, Saurabh Kumar, Rémi Munos, Marc G Bellemare, and Will Dabney. Statistics and samples in distributional reinforcement learning. *arXiv preprint arXiv:1902.08102v1*, 2019.