CS 547 Project: Distributed Q-Learning

Apratim Mishra [*1], Sayantan Dutta [†2], Shu Li [‡3], and Siddharth Mansingh [§4]

[1,2,3,4]University of Illinois at Urbana-Champaign

December 2019

1 Distributional Perspective on Reinforcement Learning

In reinforcement learning, we use the Bellman equation to approximate the expected value of future rewards. If the environment is stochastic in nature and the future rewards follow multimodal distribution (bimodally distributed commute time), choosing actions based on expected value may lead to suboptimal outcome.

Another obvious benefit of modeling distribution instead of expected value is that sometimes even though the expected future rewards of 2 actions are identical, their variances might be very different. If we are risk averse, it would be preferable to choose the action with smaller variance.

The Bellman equation is a recursive expression that relates the Q functions of consecutive time steps.

$$Q^{\pi^*}(s,a) = r + \gamma \max_{a'} Q(s',a')$$

With distributional Bellman, the $Q(s,a)$ is replaced by $Z(s,a)$ and the distributional version of Bellman equation is given by

$$Z(s,a) = r + \gamma \max_{a'} Z(s',a')$$

The random variable Z is called the Value distribution. The Distributional Bellman equation is used to iteratively approximate the value distribution Z. It's emphasized that Z^π describes the intrinsic randomness of the agent's interactions with its environment, rather than some measure of uncertainty about the environment itself.

The distributional Bellman operator $\mathcal{T}^\pi : \mathcal{Z} \to \mathcal{Z}$ is defined as

$$\mathcal{T}^\pi Z(x,a) := R(x,a) + \gamma P^\pi Z(x,a)$$

The three sources of randomness that define the compound distribution $\mathcal{T}^\pi Z$ are :

- The randomness in reward R
- The randomness in transition P^π
- The next state value distribution $Z(X',A')$

A greedy policy π is for $Z \in \mathcal{Z}$ maximizes the expectation of Z. The set of greedy policies for Z is

$$\mathcal{G}_Z := \{\pi : \sum_a \pi(a|x)\mathbb{E}Z(x,a) = \max_{a' \in \mathcal{A}} \mathbb{E}Z(x,a')\}$$

Thus any operator $\mathcal{T}$ that implements a greedy selection rule i.e.

$$\mathcal{T}Z = \mathcal{T}^\pi Z \text{ for some } \pi \in \mathcal{G}_Z$$

is called a distributional Bellman **optimality** operator.

[*]apratim3@illinois.edu
[†]sdutta26@illinois.edu
[‡]lishu2@illinois.edu
[§]sm38@illinois.edu

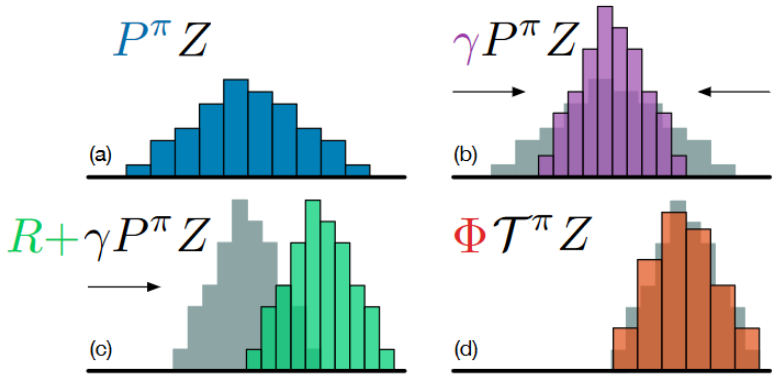

Figure 1: A distributional Bellman operator with a deterministic reward function: (a) Next state distribution under policy π, (b)Discounting shrinks the distribution towards 0, (c) The reward shifts it, and (d) Projection step (Section 1.2)[1]

1.1 Parametric Distribution

Since the value distribution Z is not a scalar quantity like Q, the paper proposes the use of a discrete distribution parameterized by the number of supports (i.e. discrete values) to represent Z. The support is given by a set of **atoms** $\{z_i = V_{\text{MIN}} + i\Delta z : 0 \leq i < N\}, \Delta z := \frac{V_{\text{MAX}}}{V_{\text{MIN}}}$ The atom probabilities are given by a parametric model $\theta : \mathcal{X} \times \mathcal{A} \to \mathbb{R}^N$

$$Z_\theta(x, a) = z_i \quad \text{w.p.} \quad p_i(x, a) := \frac{e^{\theta_i(x,a)}}{\sum_j e^{\theta_j(x,a)}}$$

1.2 Projection Operation

Since the Bellman update $\mathcal{T}Z_\theta$ and the parametrization Z_θ have disjoint supports, the authors propose **projecting** the sample update $\hat{\mathcal{T}}Z_\theta$ unto the support of Z_θ. Let π be the greedy policy w.r.t $\mathbb{E}Z_\theta$. Given a sample transition(x, a, r, x') the Bellman update $\hat{\mathcal{T}}z_j := r + \gamma z_j$ is carried out for each **atom** z_j. The i^{th} component of the projected update $\Phi\hat{\mathcal{T}}Z_\theta(x, a)$ is given by

$$(\Phi\hat{\mathcal{T}}Z_\theta(x, a)_i) = \sum_{j=0}^{N-1} \left[1 - \frac{|[\hat{\mathcal{T}}z_j]_{V_{\text{MIN}}}^{V_{\text{MAX}}} - z_i|}{\Delta z}\right]_0^1 p_j(x', \pi(x'))$$

The sample loss $\mathcal{L}_{x,a}(\theta)$ is the cross entropy term of the KL divergence $D_{KL}(\Phi\hat{\mathcal{T}}Z_{\tilde{\theta}}(x, a)||Z_\theta(x, a))$ This choice of distribution and loss has been established as the **categorical** algorithm. The alogrithm has been briefly described below 1.

2 Description of Dataset and Computational Cost

We use the environments provided by Gym, a Python library that makes various games available for research, as well as all dependencies for the Atari games. Developed by OpenAI, Gym offers public benchmarks for each of the games so that the performance for various agents and algorithms can be uniformly /evaluated. There were two attempts by our group to achieve the standard benchmarks. **Shu** and **Siddharth** tried to build upon the provided Homework 8 and modify it to achieve the benchmarks for distributed Q-learning while **Sayantan** and **Apratim** tried to modify available code from github[2] in order to fine tune and achieve better scores in namely two games, **Pong** and **Breakout**. While Pong scores met standard scores, there was no luck in sight with breakout even after trying a multitude of hyperparameters. We present our results below.

Computational Cost for training the Pong game was 30 hours of compute time with 2 GPU nodes. For Breakout even with 40 hours of training the code didnt achieve a score greater than 40 which shows there might be a problem with the implementation.

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 $j = 0, \ldots, N-1$ **do**
  $\hat{\mathcal{T}} z_j \leftarrow [r_t + \gamma_t z_j]_{V_{\text{MIN}}}^{V_{\text{MAX}}}$
  $b_j \leftarrow (\hat{\mathcal{T}} z_j - V_{\text{MIN}})/\Delta z$ ▷ $b_j \in [0, N-1]$
  $l \leftarrow \lfloor b_j \rfloor, u \leftarrow \lceil b_j \rceil$
  $m_l \leftarrow m_l + p_j(x_{t+1}, a^*)(u - b_j)$
  $m_u \leftarrow m_u + p_j(x_{t+1}, a^*)(b_j - l)$
end for
output $-\sum_i m_i \log p_i(x_t, a_t)$ ▷ Cross-entropy loss

3 Results

The pong results were greater than 15 which is greater than what DQN has achieved. The results were comparable to [1].

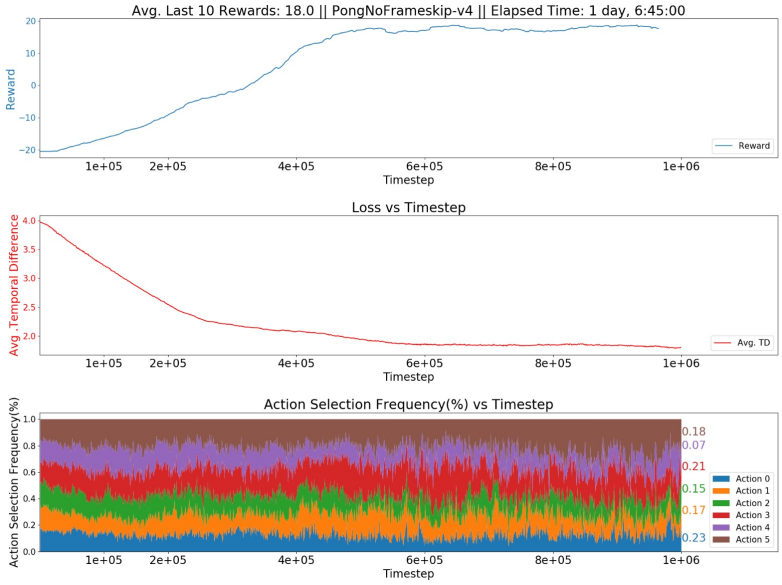

Figure 2: Pong Results

4 Details of Implementation

We first use a deep neural network to represent the value distribution. Since the inputs are screen pixels, the first 4 layers are convolutional layers . The neural network outputs values of distribution predictions for each action. Each set of prediction is a softmax layer with 51 units. In our case of Pong environment atoms is the number of discrete values = 51. First we sample a minibatch of sample paths from the Experience Replay buffer and initialize the corresponding states, reward, and targets variables. Then we store the probability mass of the value distribution and carry out a forward pass to get the next state distributions. The model outputs distributions for each action. Our target is to get the one with the largest expected value to carry out the next update. We then compute the target distribution by scaling with discount factor shifting by reward value. The we project it to the 51

discrete supports. We then perform minimize the mean square error loss using Adam optimizer. At the beginning of training, the agent performs only random actions and as we can see from the plot, the agent gets a reward of around -20 as it is losing most of the time. After around 1000 timesteps the agent already learns to hit the ball and is able to score its first points. We found that the initial learning curve was increased monotonically around 400000 time steps, but learning rate was slow seems to be because of the exploration policy. After that as the agent starts gathering experience and reducing the exploration strategy over time we saw a rate improvement till 500000 time steps. After that the performance saturates to score of around 18.

5 Choice of Hyperparameters

In addition to the existing parameters from the DQN algorithm, the significant hyperparameters were the **atoms**, $\mathbf{V}_{\text{MAx}}$ and $\mathbf{V}_{\text{MIN}}$ which were choosen to be 51, 10 and -10 as is done in the paper.

We started with exploration factor = 1.0 and then gradually decreased it as agent kept learning. During our experimentation we found that updating the target network in the right frequency is very important. After lot of trial and error process we found that update frequency = 4 and target network update = 5000 gave best result, that is the default parameters are updated every 4 steps and target network is updated every 5000 steps. We kept replay buffer size = 100000 Gradients are clipped to a certain threshold value, if they exceed it. As we used Adam in BlueWaters we made sure the optimizer step doesn't go over 1024.

6 Description of Code

6.1 GitHub source

We found a few sources online that had implemented C51 algorithm in pytorch [3] and [2], [3] being the source that has implemented many other reinforcement learning architectures other than C51.While [3] is a very extensive source, it's not readable and thus we went forward with adopting [2] and modified the architecture a bit and that gave above par benchmark results for Pong(scores of 19). The model in this code was modified which resulted in such high scores.The model is as follows

$$\begin{aligned}\textbf{input} \rightarrow &\textbf{Conv2D}(\textit{input}, 32, 8, 4) \xrightarrow{\textbf{BatchNorm2D}(32)} \textbf{Conv2D}(32, 64, 4, 2) \xrightarrow{\textbf{BatchNorm2D}(64)} \textbf{Conv2D}(64, 64, 3, 1) \\ &\xrightarrow{\textbf{BatchNorm2D}(64)} \textbf{Conv2D}(64, 128, 3, 1) \xrightarrow{\textbf{BatchNorm2D}(128)} \textbf{Linear}(3136, 1024) \rightarrow \textbf{Linear}(1024, 512) \\ &\rightarrow \textbf{Linear}(512, \textit{actions} * \textit{atoms}) \rightarrow \textbf{output}\end{aligned}$$

For more details please refer to the file **Categorical-DQN_Github_Mod.py**. The plots have been obtained by the plotting files from the github source.

6.2 Building on existing homework code

There was an attempt to modify the existing code given to us for Homework 8. We modified the files **Model.py** and **Algo.py** in order to accomodate the C51 algorithm. **Model.py** contains new linear layer units to accomodate the value distribution Z for each action. **Algo.py**'s *train, act,observe* methods have been changed accordingly. However this implementation didn't yield positive results so there is still some work that needs to be done.

7 Acknowledgement

We thank the TA Yuanyi Zhong for helpful discussions about the project. We also thank the course instructor Justin for having taught us the awesome class on Deep Learning.

References

[1] Bellemare, M. G., Dabney, W. & Munos, R. A distributional perspective on reinforcement learning `http://arxiv.org/abs/1707.06887v1`.

[2] Qfettes. qfettes/deeprl-tutorials (2019). URL https://github.com/qfettes/DeepRL-Tutorials.

[3] ShangtongZhang. Shangtongzhang/deeprl (2019). URL https://github.com/ShangtongZhang/DeepRL.