Distributed Q-Learning: A Democratic Learning Process

Max Robinson max.robinson@jhu.edu
Johns Hopkins University,
Baltimore, MD 21218 USA

Abstract

1. Introduction

Reinforcement learning can be described simply as the learning process of an agent in an environment trying to reach a goal. The agent learns by attempting to maximize a reward. In some situations the learner has very little prior knowledge of the environment. The act of maximizing the reward is the learning process.

Reinforcement learning typically requires an agent to follow a $state \rightarrow action \rightarrow state'$ loop. As a result, the learning process is also typically sequential. For large state spaces or complex environments, this process can be slow. A single agent must often experience many iterations of maximizing a reward in order to learn in even a simple environment.

A famous example of Reinforcement Learning is Tesauro's TD-Gammon agent. The best performing agent required 1,500,000 training games to beat one of the best Backgammon players at the time (Tesauro, 1995). As a more modern example, Mnih et al. (2013) developed an algorithm to play Atari 2600 video games called DQN. To learn to play each game at a human level or higher, 50 million frames were required to train an agent for each game. Total training on all 49 games in the study took about 38 days of game experience (Mnih et al. 2015).

The constraint of a lone agent acting sequentially can create situations where training an agent to learn a task can take an exorbitant amount of time. To combat this, researchers have focused on ways to adapt these reinforcement learning algorithm to run in parallel to decrease the amount of time it takes a single agent to learn.

Recent research has studied speeding up Deep Neural Networks for reinforcement learning such as DQN (Mnih et al., 2013) and others. Quite a few papers have suggested ways of parallelizing both the computation for these methods as well as distributing actors and learners to run in parallel, which send back gradients to be used to update a centralized parameter store.

2. Previous Work

Q-Learning (Watkins, 1989) has been a foundational algorithm in reinforcement learning, especially after it was shown to have optimal convergence in the limit (Watkins & Dayan, 1992). Very soon after this, asynchronous methods of Q-learning were being explored.

Tsitsiklis (1994) studied if Q-learning would still keep its optimal convergence property in an asynchronous setting. The results showed that the Q-learning algorithm does continue

to keep this convergence guarantee in an asynchronous setting, given that old information is eventually discarded through the update process, along with a few other conditions.

More recently there has been a flurry of work done in parallelizing reinforcement learning algorithms. Mannion, Duggan, & Howley (2015) used a distributed Q-learning algorithm with multiple agents and environments to learn in, applied to traffic signal control. In their algorithm, at every time step the agents update a global Q Matrix in accordance with the Q-learning update equation.

In addition to using a tabular implementation of Q-learning, function approximation versions using deep neural networks, such as DQN (Mnih et al., 2013), have also explored asynchronous and distributed architectures for learning.

The General Reinforcement Learning Architecture (Gorila) (Nair et al. 2015) uses a general, massively distributed architecture of agent, and environments to learn. Each agent in Gorila has its own environment, a copy of the model which is periodically updated, a separate replay memory and a learner that aims to learn using a DQN (Mnih et al., 2015). The learner samples data from the memory replay and computes the gradients for the parameters, which are then sent to a centralized parameter server. The parameter server then updates a central copy of the model based on gradients.

A3C (Mnih et al. 2016) collapses Gorila onto a single machine, using multiple processors to parallelize the agents actions in an environment. A3C similarly computes gradients for each agent's model at each time step, locally, but accumulates these gradients over a series of time steps before updating the central model, every so often. This aims to balance computational efficiency with data efficiency.

3. Q-Learning

4. Distributed Q-learning

Distributed Q-Learning (DistQL) uses Q-Learning as a foundation for learning and delegates learning to multiple agents, similar to what is done in A3C and Gorilla. DistQL uses multiple worker agent-environment pairs to learn rather than a single agent and environment. As the agents learn and complete epochs, a central QServer, with memory Q_c, is updated. The QServer is updated at regular intervals by the workers, based on an update frequency τ. The QServer also keeps a central learning rate $\eta_{c,(s,a)}$ for each (s, a) pair. This learning rate is decayed according the the number of times that particular (s, a) pair has been updated.

More formally, DistQL has n worker agents $A_1, ..., A_n$. Each agent also has a copy of the environment in which it acts, $E_1, ..., E_n$. Each agent keeps track of a local Q-memory, Q_i, as well as local parameters such as discount factor γ_i, exploration chance ϵ, and learning rates $\eta_{i,(s,a)}$. Each agent acts and learns inside of its environment according to an ϵ-greedy Q-Learning algorithm.

As the worker learns it will send updates to the QServer, Q_c. In order to send updates, a list of $Q_i(s, a)$'s that have been modified since the agent last sent an update is kept by the worker, $Set_{Q_i} = Set(Q_i'(s, a), ...)$. The agent sends updates to Q_c every τ epochs. τ is a hyperparameter that can be adjusted to change the frequency with which workers update Q_c. An architecture digram of DistQL can be seen in Figure 1.

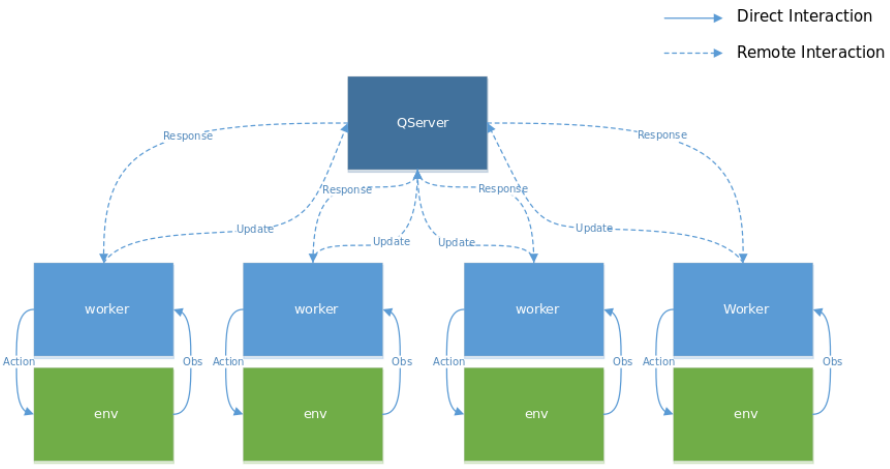

Figure 1: DistQL Architecture with 4 agents

The QServer is modified each time it receives an update from a worker. Q_c is then updated in accordance to equation 1. For readability η_c is used for brevity with the understanding that the learning rate is actually $\eta_{c,(s,a)}$.

$$Q_c(s,a) = (1 - \frac{\eta_c^2}{\eta_i})Q_c(s,a) + \frac{\eta_c^2}{\eta_i}Q_i(s,a),\ 0 < \eta_c \leq \eta_i < 1 \tag{1}$$

Equation 1 is built on the idea that the learning rate η can be thought of as a measure of how certain the algorithm is that the current Q-value stored will be the final Q-value in the table. For instance, if $\eta_{i,(s,a)} = .9$ and $Q_{i,(s,a)} = 1$ the worker will apply a large learning rate on the next update to (s,a). This means that the worker is willing to make larger adjustments to the Q-value, and it is less likely that the current Q-value will be the same as the final Q-value in the table, compared to later on. If $\eta_{i,(s,a)} = .01$ and $Q_{i,(s,a)} = 1$, it is much more likely that the current Q-value for (s,a) will remain close to the current Q-value.

The ratio $\frac{\eta_c^2}{\eta_i}$ is calculated to quantify how much the worker i and Q_c agree about how "stable" the Q-value in Q_c is. The ratio can be re-written as $\frac{\eta_c}{\eta_i}\eta_c$ to demonstrate this more clearly. If $\eta_c = \eta_i$ then Q_c and Q_i agree on how "stable" the Q-value is, and an update of $(1 - \eta_c)Q_c + \eta_c Q_i(s,a)$. Otherwise, $\frac{\eta_c}{\eta_i} < 1$ which puts more emphasis on the Q-value already stored by Q_c rather than Q_i. Intuitively, the idea is that if the central Q-value is more "stable" than a workers, then the central value should not be highly impacted by a worker's value which is less likely to converge to its own Q-value.

It is possible for η_i to be less than η_c when updates are sent to the QServer. One case where this can happen is if the worker has updated a local (s,a) more times than Q_c has updated (s,a). When $\eta_i < \eta_c$, the central learning rate adopts the workers learning rate such that $\eta_c \leftarrow \eta_i$ prior to the update taking place. This ensures that the central learning rate is always less than or equal to the workers learning rate. Since the learning rate is interpreted as a measure of stability for the Q-value, if a worker has a Q-value that is more

stable than the central server, the server wants to acknowledge and adopt the learning rate of the worker, when updating Q_c.

Using a central learning rate η_c allows for the update equation to remain asynchronous. By keeping a central learning rate, the QServer can remain agnostic to the frequency of the updates from workers and from the order of the updates. This means less logic must be written to coordinate the updates from workers. This allows each worker to update the Q_c without having to wait for other workers to also be updating Q_c. This also means that, not all workers have to be started at the same time, or remain in-sync with other works on which epoch they are on. This has benefits from a scaling perspective since allows workers to be far less dependent on the QServer.

After updates to Q_c are made, there are two strategies that are explored for updating the workers. The QServer can either send the entire contents of Q_c to the worker, or it can send back only updates states that the worker initially sent to the QServer. These options are denoted DistQL-ALL and DistQL-Partial respectively. DistQL-ALL is the default option, and is synonymous with DistQL.

In DistQL-ALL, workers will receive the response from the QServer which contains a copy of Q_c. The worker will then overwrite all Q_i Q-values with the corresponding Q_c Q-value. For any states in Q_c that are not in Q_i, the agent adopts both the Q-value and the learning rate from Q_c. This method provides workers with additional information collected from other workers and has the effect of homogenizing their memories. In this variation, state-action pairs that were explored by one worker would then be shared with other workers even if the worker has not explored that state-action pair.

In DistQL-Partial, workers only receive updates for the states submitted to the QServer. If worker A_i updates the QServer with Set_{Q_i}, A_i receives back a set Set'_{Q_i} containing the same set of state-action pairs but with Q-values from Q_c. The worker then replaces any local Q-values with values in the response set. This adaptation just has the effect of reconciling the Q-values known by the worker with the central server. This provides shared knowledge only about states the worker knows about. No new state-action pairs are shared through this update. Thus exploration is not shared across workers.

5. Experiments

To demonstrate the effects of DistQL, the Taxi world (Dietterich, 2000) and Cart Pole (Barto et al. 1983) environments were chosen for agents to learn in. The environment implementations used are part of the OpenAI Gym and are publicly available. The goal in each environment is for the agent to achieve the highest reward possible in the environment.

The taxi world is a 5 by 5 grid world where the agent is a taxi driver, and there 4 marked spaces in the grid. The goal of the agent is to navigate to the passenger, pick him or her up, navigate to the goal, and drop the passenger off at another destination. For rewards, the agent receives +20 points for successfully dropping off the passenger, -1 point for every time step, and -10 penalty for any illegal pick-up or drop-off actions made. The actions available to the agent are to move in the four cardinal and perform a pick-up or drop-off action. An episode finishes after the passenger has been successfully dropped off at their destination.

The Cart Pole environment consists of trying to balance a pole upright while attached by a joint to a cart on a frictionless track. The pole starts upright and the goal is to keep the pole balanced upright for as long as possible. The cart can move left or right along the track by applying a force of +1 or -1. The episode finishes after the pole leans 15 degrees left or right or the cart has moved 2.4 units from the center. A reward of +1 is given to the agent for every time step that the pole stays upright. For this experiment, the OpenAI implementation was modified so that when an episode is terminated a reward of -200 is incurred.

For Cart Pole, the implementation uses a continuous state space with 4 features. In order to apply Q-Learning, the continuous state space must be discretized. The first three features are discretized into 8 bins of equal size, with values between -2.4 to 2.4, -2 to 2 and -1 to 1 for features 0, 1, and 2 respectively. The fourth feature is discretized into 12 bins of equal size between the values of -3.5 to 3.5. This specific discretization is chosen because of the anecdotal and demonstrated success described by Victor Vilches (2017).

A series of experiments were run in each environment. DistQL-ALL and DistQL-Partial were run with 1, 2, 4, and 8 agents each with 10 trials. Each trail consists of 2000 epochs for each agent. A limit of 2000 epochs per trail is used to allow for reasonable training times on a smaller machine but still demonstrate learning occurring in each environment. For each number of agents run, τ was adjusted to be 10, 50, and 100. This allows a comparison of how update frequency impacts learning. Table 1 shows all experiments performed. A total of 48 experiments were run.

Each agent and the central Q server used the same hyperparameters for all experiments. The learning rate η for each (s, a) was initialized to .5 and decayed so that $\eta = 0.5 \times 0.999^i$ where i is the ith update to (s, a). The exploration rate ϵ was initialized to .1 and decayed so that $\epsilon = 0.5 \times 0.999^j$ where j is the jth episode in the learning process. The discount factor γ was set to .9. These hyperparameters were chosen based on anecdotal success of a single Q Learner acting in the environments. An exhaustive search for optimal hyperparameters and decay rates was not performed.

For these experiments, the DistQL was implemented in Python 3.5 using an http-based client-server architecture. Each agent was run in a thread and provided updates and received responses through the http protocol. The server implementation is single-threaded and does not allow for concurrent modification of the Q-memory. The Q-Learning agent is also written in python as part of a framework for DistQL. All source code for the DistQL implementation can be found at https://github.com/MaxRobinson/DistributedQMemory.

5.1 Evaluation

Since in each environment gives a reward at each time step, the cumulative reward for each episode is used to judge the performance of the agent or agents in each environment. Specifically, the average cumulative reward for each set of agents over the 10 trails is compared with each other set of agents in that environment. For example, the performance of DistQL-ALL with 2 agents is calculated by taking the average cumulative reward for each epoch that each agent experienced over the 2000 epochs individually, and then averaging the performance of the two agents together. This provides an aggregate average cumulative reward for all agents in DistQL-ALL for 2 agents.

	DistQL-ALL		DistQL-Partial	
Environment	Number of Agents	Tau	Number of Agents	Tau
Taxi World	1	10	1	10
		50		50
		100		100
	2	10	2	10
		50		50
		100		100
	4	10	4	10
		50		50
		100		100
	8	10	8	10
		50		50
		100		100
Cart Pole	1	10	1	10
		50		50
		100		100
	2	10	2	10
		50		50
		100		100
	4	10	4	10
		50		50
		100		100
	8	10	8	10
		50		50
		100		100

Table 1: Experiment Table

The performance of the DistQL with the different number of agents are compared to one another. A set of agents is considered to have done better if the average cumulative reward achieved by the set of agents is higher than other sets of agents earlier on. The higher the average cumulative reward achieved earlier, or in fewer episodes, the better.

5.2 Results

6. Discussion

7. Conclusion

References

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, *SMC-13*(5), 834–846.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym..

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *J. Artif. Int. Res.*, *13*(1), 227–303.

Mannion, P., Duggan, J., & Howley, E. (2015). Parallel reinforcement learning for traffic signal control. *Procedia Computer Science*, *52*, 956 – 961. The 6th International

Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015).

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In Balcan, M. F., & Weinberger, K. Q. (Eds.), *Proceedings of The 33rd International Conference on Machine Learning*, Vol. 48 of *Proceedings of Machine Learning Research*, pp. 1928–1937, New York, New York, USA. PMLR.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*, 529 EP –.

Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., Maria, A. D., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., Legg, S., Mnih, V., Kavukcuoglu, K., & Silver, D. (2015). Massively parallel methods for deep reinforcement learning. *CoRR*, *abs/1507.04296*.

Tesauro, G. (1995). Temporal difference learning and td-gammon. *Commun. ACM*, *38*(3), 58–68.

Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and q-learning. *Machine Learning*, *16*(3), 185–202.

Vilches, V. M. (2017). Basic reinforcement learning tutorial 4: Q-learning in openai gym..

Watkins, C. (1989). *Learning From Delayed Rewards*. Ph.D. thesis, Cambridge University, U.K.

Watkins, C., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292.