Training 2048 game AI with Parallel Architecture

Chia-Hun Yu
0411254 NCTU CS
yamiefun@gmail.com

Wei-Chieh Yu
0411228 NCTU CS
rftheitu0130@gmail.com

Li-Yang Wang
0416014 NCTU CS
osinoyan.cs04@g2.nctu.edu.tw

Introduction

In recent years, deep learning has already been used in multiple research area, for instance, object detection, machine translation and speech recognition. Most of these applications adopt supervised learning, therefore, numerous label data which demands plenty of time for researchers to arrange is needed. Moreover, the quality of these label data hardly affects the performance of agent.

Reinforcement learning is a method that agents acquire experience by interacting with environment. However, reinforcement learning costs a bunch of time exploring in the environment, greatly increase the difficulty for training a neural network.

Reducing the time of training seems to be not only challenging but also meaningful in order to improve the efficiency at reinforcement learning field. In this project we attend to accelerate training process by parallel exploring the environment through multiple workers.

Statement of Problem

We choose the programming language C++ as the environment in our experiment. Our project focus on the acceleration of the training process of a well-known mathematical computer game called *2048* by parallel programming methods.

Why we choose 2048?

2048 is a mathematical computer game with simple operating rules, while having tremendous number of branches of steps if we aim to get a higher score in the game. In the approach we adopted to simulate this game in our experiment, there is no need for the agent to learn to analyze the screen of the game. The environment that we use, simulating the game with only binary operations, is a more efficient method for agent to interact with the environment and collect experience. In this way, first, we can reduce the complexity of the neural network to reduce the cost of time required for training under a low computational resource environment; second, we may put more focus on analyzing the efficiency of parallel programming in this experiment.

Approaches

Our main idea is referred to the paper *Distributed deep q-learning* (Distributed deep q-learning; Chavez, K et al. 2015)[1], and modified according to our experiment. Algorithm 1 illustrates what every slave threads should execute, the greatest difference from DQN algorithm is that worker threads don't update their parameters by themselves on local side, instead, workers simply send the resulting gradient to master thread, then get the return updated network parameter from master thread. That is, worker threads will simply work on four procedures :

1) Fetching the latest model, from the server
2) Generating new data for its local shard
3) Computing a gradient using this model and mini-batch from its local replay memory dataset
4) Sending the gradient back to the server

Algorithm 1 Worker k: ComputeGradient
state: Replay dataset $\mathcal{D}_k$, game state s_t, target model $\hat{\theta}$, target model generation ℓ

Fetch model θ and iteration number n from server.
if $n \geq \ell\ C$ **then**
$\hat{\theta} \leftarrow \theta$
$\ell \leftarrow \lfloor n/C \rfloor + 1$
$q \leftarrow \max\{(N_{\max} - n)/N_{\max}, 0\}$
$\epsilon \leftarrow (1-q)\epsilon_{min} + q\epsilon_{max}$
Select action $a_t = \begin{cases} \max_a Q(\phi(s_t), a; \theta) & \text{w.p. } 1-\epsilon \\ \text{random action} & \text{otherwise} \end{cases}$
Execute action a_t and observe reward r_t and frame x_{t+1}
Append $s_{t+1} = (s_t, a_t, x_{t+1})$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
Store experience $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}_k$
Uniformly sample minibatch of experiences X from $\mathcal{D}_k$, where $X_j = (\phi_j, a_j, r_j, \phi_{j+1})$
Set $y_j = \begin{cases} r_j & \text{if } \phi_{j+1} \text{ terminal} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \hat{\theta}\right) & \text{otherwise} \end{cases}$

$$\Delta\theta \leftarrow \nabla_\theta f(\theta; X) = \frac{1}{b}\sum_{i=1}^{b} \nabla_\theta \left(\frac{1}{2}(Q(\phi_i, a_i; \theta) - y_i)^2\right)$$

Send $\Delta\theta$ to parameter server.

Algorithm 2 is the main part of the training process. Master thread receives every return $\Delta\theta$, then pass the updated network parameter to specific thread in order to continue on their works. Considering the time cost of message passing might overwhelm, under the premise of not affecting the training quality, we will probably lower the update frequency.

Algorithm 2 Distributed deep Q-learning

function RMSPROPUPDATE($\Delta\theta$)
  $r_i \leftarrow 0.9r_i + 0.1(\Delta\theta)_i^2$ for all i
  Acquire write-lock
  $\theta_i \leftarrow \theta_i - \alpha(\Delta\theta)_i / \sqrt{r_i}$
  Release write-lock
Initialize server $\theta_i \sim \mathcal{N}(0,\xi)$, $r \leftarrow \mathbf{0}$, $n \leftarrow 0$
for all workers k **do**
  AsyncStart
    Populate $\mathcal{D}_k$ by playing with random actions
    repeat
      COMPUTEGRADIENT
    until server timeout
  AsyncEnd
while $n <$ MaxIters **do**
  if latest model requested by worker k **then**
    Acquire write-lock
    Send (θ, n) to worker k
    Release write-lock
  if gradient $\Delta\theta$ received from worker k **then**
    RMSPROPUPDATE($\Delta\theta$)
    $n \leftarrow n+1$
Shutdown server

In Figure 1, the parameter server will save a set of global parameters (e.g. θ). After every workers interacts with their own gaming environment asynchronously, they save their transitions in their experience replay memory and sample batches out for training to calculate the gradient (e.g. $\Delta\theta$) which will be sent to server end. While the server receives a update request, it will update the global parameter and send the updated one back to the requesting worker to continue the training process.

Since every worker need to execute on a independent CPU core, there gaming environment and local memory are independent of each others, also there is no need of communications between workers which is similar to master-slave model, this enables the execution of workers to be parallelized.

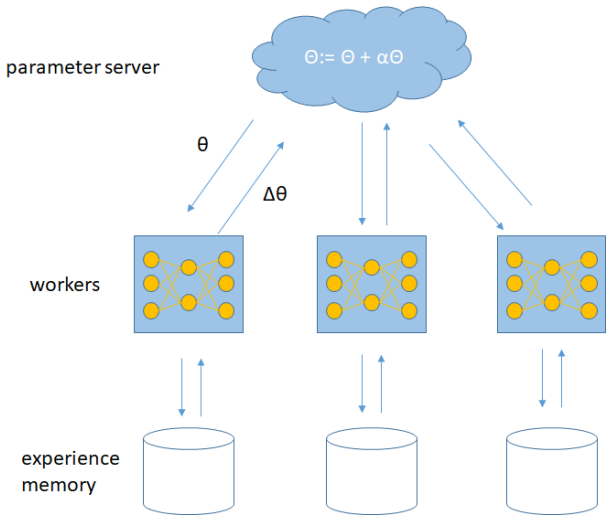

Fig. 1: The architecture of our algorithm model

Language Selection

We parallelize C++ code using Message Passing Interface(MPI). Since our task adopts master slave model which requires frequently communication among threads, MPI is a appropriate way to implement.

Related Work

A. Markov decision process

In an MDP, an agent chooses action A_t at time t after observing state S_t. The agent then receives reward R_t, and the state evolves probabilistically based on the current state-action pair.

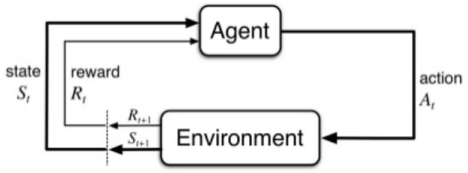

Fig. 2: Basic MDP diagram

The goal is to find a policy π^* that, if followed, the agent will maximize the expected sum of rewards given by the environment. π^* is related to the optimal state-action value function $Q^*(s,a)$, which is the expected value when starting in state s, taking action a, and then following actions dictated by π^*. Mathematically, it obeys the Bellman recursion

$$Q^\star(s,a) = R(s,a) + \sum_{s' \in \mathcal{S}} T(s,a,s') \max_{a' \in \mathcal{A}} Q^\star(s',a').$$

By using value iteration, we can compute the state-action value function. To obtain the optimal policy for state s, we compute

$$\pi^\star(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q^\star(s,a).$$

B. Reinforcement learning

The problem Reinforcement learning(RL) seeks to solve differs from the standard MDP in that the state space and transition and reward functions are unknown to the agent, hence, the agent needs to interacts with an environment through a sequence of observations, actions, and rewards and learns from the experience.

The basic idea is to estimate

$$Q^{\star}(s,a) = \max_{\pi} \mathbf{E}[R_t \mid s_t = s, a_t = a, \pi],$$

where π maps states to actions, with the additional knowledge that the optimal value function obeys Bellman equation

$$Q^{\star}(s,a) = \mathop{\mathbf{E}}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^{\star}(s',a') \mid s,a\right],$$

where ε is the MDP environment.

C. Deep Q-learning

Google DeepMind proposed Deep Q Network(DQN) on Nature in 2015[2]. They combine nonlinear estimation in Deep Learning with the advantage of learning from experience in Reinforcement Learning and advance a new concept, Deep Reinforcement Learning(DRL). They apply it on Atari Games and successfully exceed human-level performance in most of the games, becomes a significant milestone in this field.

D. Asynchronous Advantage Actor-Critic

Asynchronous Advantage Actor-Critic(A3C; Google DeepMind, 2016)[3] is a training method advanced by DeepMind. It combines the asynchronous method, advantage value and deep deterministic policy gradient together to make the convergence result better.

In A3C, because the exploration work is done by multiple workers instead of one worker originally, result in better exploration property.

Expected Results

Even though this algorithm requires frequently message passing between master thread and slave threads, we still expect it to have a decent acceleration under multi-core environment.

Timetable

Date	Schedule
11/1~11/20	design & implement the algo.
11/20~12/31	do experiments
1/1~1/11	analyze the result

References

[1] ONG, H. Y., CHAVEZ, K., AND HONG, A. Distributed deep q-learning. CoRR abs/1508.04186 (2015).

[2] 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.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. Nature 518(7540):529–533.

[3] Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In International Conference on Machine Learning.