Chapter 12
Parallel Computing

Huaqing Zhang and Tianyang Yu

Abstract Due to the low sample efficiency of reinforcement learning, parallel computing is an efficient solution to speed up the training process and improve the performance. In this chapter, we introduce the framework applying parallel computation in reinforcement learning. Based on different scenarios, we firstly analyze the synchronous and asynchronous communication and elaborate parallel communication in different network typologies. Taking the advantage of parallel computing, classic distributed reinforcement learning algorithms are depicted and compared, followed by summaries of fundamental components in the distributed computing architecture.

Keywords Distributed computing · Asynchronous advantage actor-critic · Hybrid GPU/CPU A3C · Importance weighted actor-learner architecture (IMPALA) · Scalable efficient deep-RL (SEED) · Distributed proximal policy optimization (DPPO) · Ape-X · Retrace-actor (Reactor) · Recurrent replay distributed DQN (R2D2)

12.1 Introduction

In deep reinforcement learning, large amounts of data is required for model training. Take OpenAI Five (OpenAI et al. 2019) as an example, batches of around two million frames are applied for training every 2 s, so as to let the agents learn and behave smartly in the Dota game. Moreover, from the optimization perspective, large batch size can reduce variance especially for policy gradient methods. However, due to the sequential interactions between the agent and the environment, the reinforcement learning algorithm suffers from the low sample efficiency, result in the unsatisfied

H. Zhang (✉)
Google LLC, Mountain View, CA, USA

T. Yu
Nanchang University, Nanchang, China

© Springer Nature Singapore Pte Ltd. 2020

H. Dong et al. (eds.), *Deep Reinforcement Learning*,
https://doi.org/10.1007/978-981-15-4095-0_12

training performance and slow convergence speed. Parallel computing, referring to the simultaneous computation on separated but independent tasks, is implemented as an efficient solution. Generally, the parallelization can be considered from the following two perspectives:

- **Parallel Computation**: Data computation is the core procedure to perform feature engineering, modeling learning, and performance evaluations. The computation is taken over by computing unit, which can be combined and extended to different scales. Within each level, the performance of computation can be regarded into two perspectives. One is focusing multiple computing units on one task. The other is to map multiple computing units to multiple tasks and apply computation in parallel fashion. Compared with the above computation strategies, with increasing number of computing units applied on one task, the efficiency to finish the task increases but soon converges due to some bottleneck processes. In deep reinforcement learning, when the computing resources are sufficiently provided, in order to further improve the efficiency, it is beneficial to separate the task into multiple independent sub-tasks, each allocated to efficient amounts of computing resources.
- **Parallel Transmission**: When sufficient computing resources are provided, how to manage the data transmission between the computing resources may become the bottleneck to solve the problem. Generally different data transmission network typologies are put forward for different applications to avoid the transmission redundancy, to balance the transmission loads and to reduce the transmission delay. In parallel computing, as there are multiple processes or threads finishing different tasks at the same time, it is challenging to manage the data traffic and guarantee the transmission efficiency in the network with limited communication bandwidth.

In a supervised setting, one simple way to speed up the learning is to process many different input samples as once. However, in deep reinforcement learning, this is not possible because we have to let the agent and environment interact with each other sequentially to obtain all required information. What we should follow in deep reinforcement learning instead is to apply parallelization on different trajectories or batches when updating weights in deep policy and value network. In this chapter, we analyze the parallelization of deep reinforcement learning in the perspective of data computation and data transmission. We further enumerate significant distributed computing algorithms and show the general distributed computing architecture to be applied for large-scale deep reinforcement learning problems.

12.2 Synchronization and Asynchronization

In parallel computing, most of the common data transmission methods apply star topology, which is composed of one master node and multiple slave nodes. The master node generally manages the data information for the problem. It applies

data distribution and collection with each slave node. Based on the accumulated data, the general network parameters are learnt and updated. Each slave node, on the other hand, receives the allocated data from the master node, performs data computation and submits its computing results back to the master node. As there are multiple slave nodes working at the same time, under the management of the master node, the data computation can be done in parallel to finish a large-scale problem in cooperation.

The star topology is widely considered in solving deep reinforcement learning applications. The parallel version of the actor-critic method, for example, usually adopts one master node and multiple slave nodes. Each slave node maintains a deep policy network, which has the same structure as all other slave nodes and the master node. Therefore, the slave nodes can be initialized by copying the weights of the policy network from their master node. Then it can independently interact with the environment for exploration. After several rounds of interactions, the slave node communicates with the master node and sends the information related with the weights of networks. The information can be single-step exploration experience, trajectory exploration experience, buffered exploration experiences with priority information, computed gradients of the network parameters, etc., based on different detailed architecture. Accumulating the feedback and experience from each slave node, the master node can update network parameters and further announce its updated weights to slave nodes for their next round of explorations.

The star topology clearly separates the tasks and accelerates the policy learning with the parallel computing among slave nodes. However, with different computation power, each slave node may explore and collect experience with different time schedules. Then how to determine the pattern for data communication varies for different problems and system architectures, which generally is classified into synchronous communication and asynchronous communication.

The synchronous communication pattern is shown in Fig. 12.1, where the red bar is the time applied for data communication among the nodes and the blue bar is the time for computation within the node. It is noticed that the time for communication falls into the same time intervals for all slave nodes. For the master node, it has same time intervals to communicate with all slave nodes. However, for the slave nodes, the ones computing faster have to wait until all other slower ones finish the

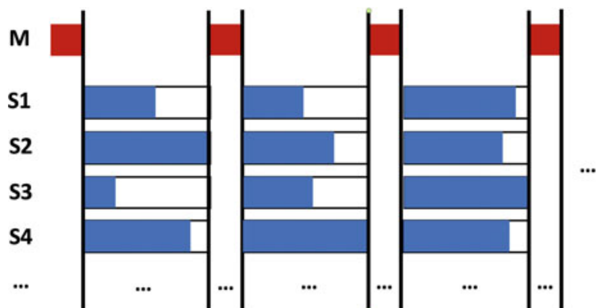

Fig. 12.1 Synchronization

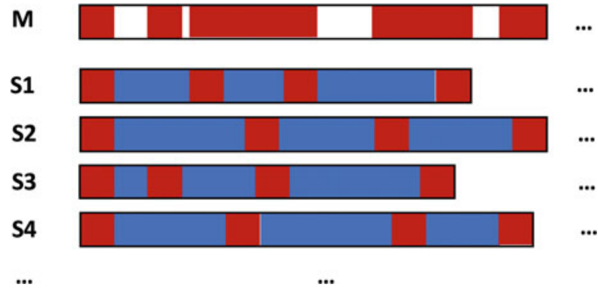

Fig. 12.2 Asychronization

computation within the round. Thus, synchronous communication is more organized for the master node to collect and analyze the computation results from the slave nodes, but there are lots of computing resources wasted on slave nodes due to the waiting for synchronization.

In order to avoid the waiting time for slave node and improve the efficiency to apply computing resources, asynchronous communication is put forward correspondingly. As depicted in Fig. 12.2, each slave node is able to submit the information to the master node as long as it finishes the training task within one round, and the master node collects the information and synchronizes with the slave node whenever the slave node finishes. Accordingly, the data communication to different slave nodes is performed within different time intervals. The master node may require communication with different slave nodes from time to time, but the computing resource are fully adopted for model training for each slave node.

12.3 Parallel Communication and Networking

The star topology is a centralized way to apply parallel computation, where the master node is able to manage and maintain the system to guarantee that all distributed tasks are well-organized. On the other side, the master node is also the weakness part of the system. In order to guarantee high performance, the master node is required to be much more efficient on information processing compared with the slave nodes. Secondly, the data transmission bandwidth towards the master node also requires to be sufficient so as to avoid delay for the data computation for all slave nodes. Moreover, the robustness of the system is highly dependent on the master node. Whatever issues causing the breakdown event on the master node, the whole system stops working even though the computing resources on slave nodes are available and sufficient.

Accordingly, for many application with demanding requirements on robustness and large-scale parallel computing power, a general distributed data computation and communication structure is necessary. We assume there are multiple independent processes, each of which maintains its own deep reinforcement learning network and communicates with others frequently from data synchronization.

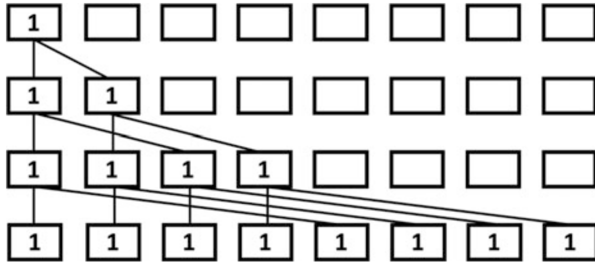

Fig. 12.3 Tree-structured communication

As each process requires to exchange the information with all other processes, when the number of processes increasing, the communication cost exponentially increases. In order to reduce the redundant communication and achieve efficiency on information synchronization, inter-process communication (IPC) are referred with message passing interfaces (MPI). Generally MPI provides basic interfaces for message sending, broadcasting, and receiving for each process. Based on the standard, different communication structures are further provided to improve the communication efficiency. The following takes some communication structures as examples for insights.

- **Tree-Structured Communication**: Assume there are N processors in the system. When a process would like to broadcast its information to all other $N - 1$ processors, it can follow a tree-structure as shown in Fig. 12.3. In the tree-structured communication, the processor first communicates with its $m - 1$ neighboring processes. Then in the next iterations, all neighbors receiving the information will further communicate to $m - 1$ new different processors in parallel to expand to all other processors. Accordingly, with increasing parallel communication, it takes $\lceil \log_m N \rceil$ iterations to broadcast the information to all processors and the information sender processor only requires to send $(m - 1)\lceil \log_m N \rceil$ times. Compared with the method to send its information to all other processors, the tree-structured communication reduces the sending times for each processor but increases the iterations to apply parallel communication.
- **Butterfly Communication**: When all N processors need to broadcast their information to all other processors simultaneously, each processor can follow the tree-structured communication and formulate the butterfly communication structure. In butterfly communication, as shown in Fig. 12.4, each processor first sends its information to its neighbors, who will further accumulate and forward the information to all other processors. As each node can collect and accumulate the information before transmitting to all other processors, the efficiency is further improved in distributed fashion. In general, it takes $\lceil \log_m N \rceil$ iterations to broadcast the information to all processors and each processor only requires to send $(m - 1)\lceil \log_m N \rceil$ times. Moreover, whenever there is a node breakdown in the middle of communication, all other nodes are still able to continue and let the information synchronized on all other processors.

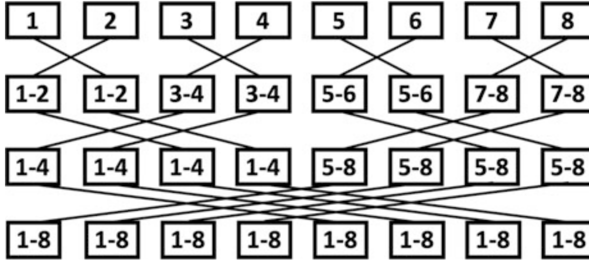

Fig. 12.4 Butterfly communication

Based on different communication structures, the parallel computation and transmission for reinforcement learning algorithm can be widely diverse and flexible. For different applications, the system architecture can be different to improve the parallelization and efficiency. In the next section, we will further summarize the general distributed computing architecture in deep reinforcement learning.

12.4 Distributed Reinforcement Learning Algorithms

12.4.1 Asynchronous Advantage Actor-Critic

Asynchronous advantage actor-critic (A3C) (Mnih et al. 2016) is the distributed algorithm derived from the advantage actor-critic (A2C) method. As shown in Fig. 12.5, there are multiple actor-learners interacting with separated but identical environments by applying A2C algorithm. Each actor-learner maintains a policy network and a value network to make smart actions. For the initialization and synchronization of the network parameters for all actor-learners, a parameter server is established, supporting asynchronous communications with all actor-learners.

From the perspective of each actor-learner, we elaborate the learning algorithm in Algorithm 1. For each learning episode, each actor-learner initially obtains the network parameters from the parameter server asynchronously. Based on the synchronized policy network, the actor-learner chooses actions and interacts with the environment for at most $t_{\max}$ steps. The explored experience is collected to train the policy and value network, generating the accumulated gradients θ and $d\theta_v$, respectively. After $T_{\max}$ steps of exploration, the actor-learner reports the accumulated gradients to the parameter server and updates the general network parameters θ and θ_v asynchronously.

Algorithm 1 Asynchronous advantage actor-critic (Actor-Learner)

Hyperparameters: Total number of steps T_{max}. Maximum steps for each episode t_{max}.
Initialize step counter $t = 1$.
while $T \leq T_{max}$ **do**
  Reset gradients: $d\theta = 0$ and $d\theta_v = 0$.
  Sync with parameter server to obtain network parameters $\theta' = \theta$ and $\theta'_v = \theta_v$.
  $t_{start} = t$
  Set starting state S_t for the episode
  while Reach terminal state **or** $t - t_{start} == t_{max}$ **do**
    Choose action a_t based on policy $\pi(S_t|\theta')$
    Act in the environment and receive rewards R_t and next state S_{t+1}
    $t = t + 1, T = T + 1$
  end while
  if Reach terminal state **then**
    $R = 0$
  else
    $R = V(S_t|\theta'_v)$
  end if
  for $i = t - 1, t - 2, \ldots, t_{start}$ **do**
    Update discounted rewards $R = R_i + \gamma R$
    Accumulate gradients wrt θ', $d\theta = d\theta + \nabla_{\theta'} \log \pi(S_i|\theta')(R - V(S_i|\theta'_v))$
    Accumulate gradients wrt θ'_v, $d\theta_v = d\theta_v + \partial(R - V(S_i|\theta'_v))^2/\partial\theta'_v$
  end for
  Asynchronously update θ with $d\theta$ and θ_v with $d\theta_v$.
end while

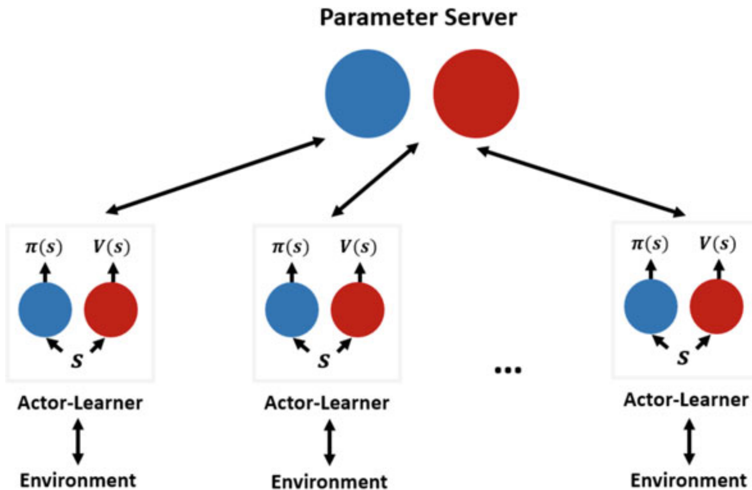

Fig. 12.5 A3C architecture

12.4.2 Hybrid GPU/CPU A3C

In order to better leverage the GPU's computational power and improve the computation efficiency, A3C architecture is further optimized to the hybrid GPU/CPU A3C (GA3C) (Babaeizadeh et al. 2017). As depicted in Fig. 12.6, from the environments or simulators to the learning model, there exist components of agent, predictor, and trainer. The functionality for each component is shown as follows.

- **Agent**: There are multiple agents interacting with their simulated environments, respectively. Each agent does not need to maintain a policy network for decision making. Instead, based on the current state S_t, the agent pushes one request to the prediction queue and lets the predictor assist to choose actions from the general policy network. After the action A_t is taken and the reward R_t and next state S_{t+1} are provided from the environment, the agent submits the experience (S_t, A_t, R_t, S_{t+1}) to the training queue for the model training.
- **Predictor**: The predictor collects the requests from the agent in the prediction queue, batches the requests, and sends to the general policy network for model inference. The batched input data for model inference takes the advantage of the parallel computation in GPU, improving the computation efficiency of the learning model. Based on the number of requests, multiple predictors with multiple prediction queues are supported to balance the trade-off of computation latency and computation efficiency.
- **Trainer**: Receiving the experiences from multiple agents, the trainer collects the data from the training queue, batches the training data, and sends to the general policy and value network for model training. The batched model training improves the computation efficiency with GPU and moreover reduces the variance and fluctuations in model training.

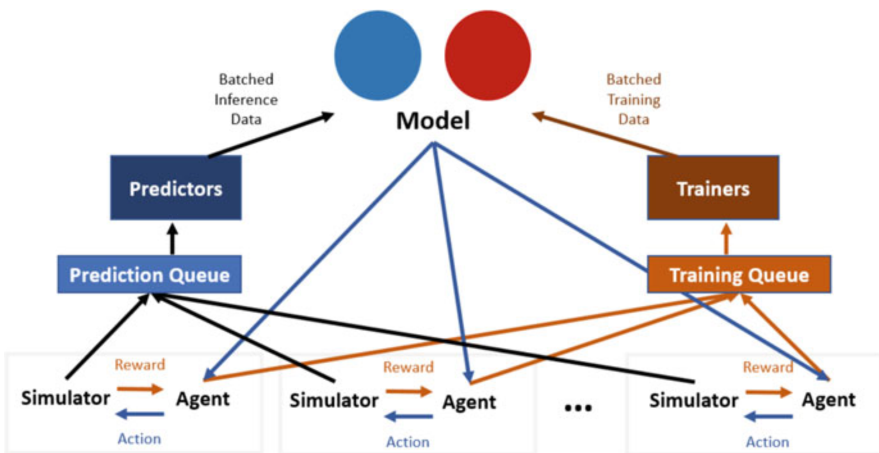

Fig. 12.6 GA3C architecture

12.4.3 Distributed Proximal Policy Optimization

Distributed Proximal Policy Optimization (DPPO) is a distributed version of the PPO algorithm. As depicted in Fig. 12.7, the algorithm includes the chief as the parameter server and workers the same as actor-learners in A3C. It distributes data collection and gradient calculation over multiple workers, which greatly reduces the learning time. Periodically, the chief updates parameters after averaging gradients passed by workers, and then passes the latest parameters to workers synchronously.

The pseudocode of the DPPO algorithm is provided in Algorithms 2, 3, and 4, corresponding to one chief and two different workers. Workers can be one of the two versions of PPO algorithm: PPO-Penalty and PPO-Clip. This section provides the corresponding two DPPO algorithms: DPPO-Penalty and DPPO-Clip. The only difference is the way in which the workers calculate the gradients, while the chief part is the same, as shown in the pseudocode.

The chief collects gradients from workers and update parameters. As shown in Algorithm 2, during each iteration, the chief waits for at least $(W - D)$ gradients from workers and updates with the averaged gradients. The latest parameters are returned to workers to continue the sampling and gradients-calculation process. At each iteration, M and B sub-iterations are performed on actor and critic, respectively.

Workers collect data samples and calculate gradients, then pass the gradients to the chief. Algorithms 3 and 4 have a similar process, except for the methods of calculating the policy gradient. At each iteration, the worker first collects a bunch of data $\mathcal{D}_k$, calculates $\hat{G}_t$ and $\hat{A}_t$, stores π_θ as π_{old}, and then performs M and B sub-iterations on actor and critic, respectively.

In DPPO-Clip, the parameter λ is also shared across workers, but its updates are determined based on local average KL divergence. Other statistical values for

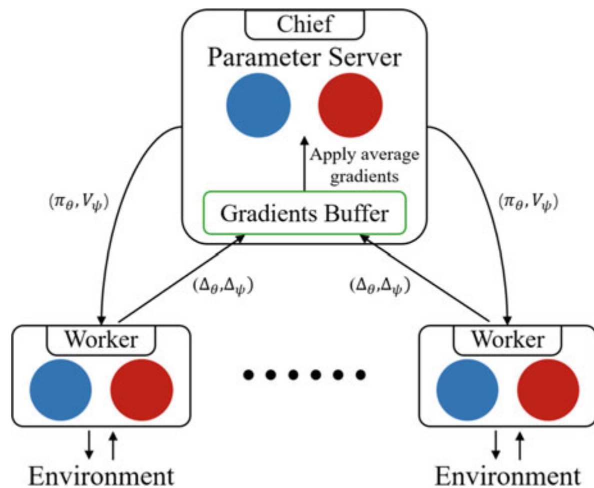

Fig. 12.7 DPPO architecture

Algorithm 2 DPPO (chief)

Hyperparameters: the number of workers W, threshold for numbers of gradients available workers D, the number of sub-iterations M, B
Input: initial global policy parameters θ, initial global value function parameters ϕ
for k = 0, 1, 2, ... **do**
  for $m \in \{1, \dots, M\}$ **do**
    Wait until at least $W-D$ gradients wrt. θ are available average gradients and update global θ
  end for
  for $b \in \{1, \dots, B\}$ **do**
    Wait until at least $W-D$ gradients wrt. ϕ are available average gradients and update global ϕ
  end for
end for

normalization in data collection are also recommended to be shared among workers, like means and standard deviations of observations, rewards, and advantages. An additional penalty term is also adopted in DPPO-Clip when the KL divergence exceeds the valid change. Early stopping is also used during each sub-iteration on the actor to improve stability.

12.4.4 IMPALA and SEED

Based on the advantage actor-critic (A2C) learning algorithm, the Importance Weighted Actor-Learner Architecture (IMPALA) (Espeholt et al. 2018) applies the trajectory experiences of the agents as the communication information for distributed computation. As shown in Fig. 12.8, the IMPALA architecture is composed of actors and learners, with the detailed introductions as follows.

- **Actor**: Within each actor, a replicated policy network interacts with a simulated environment and stores the experience into the buffer. After certain number of interactions, each actor sends the trajectory of stored experiences to the learners and receive the updates of policy network parameters from the learners in a synchronized fashion.
- **Learner**: When interacting with the actor, the learner receives the trajectory experiences of the actors and applies it for model training. The value approximation at state S_T is defined as the n-step V-trace target, as follows:

$$\text{Target} = V(S_T) + \sum_{t=T}^{T+n-1} \gamma^{t-T} \left(\Pi_{i=T}^{t-1} c_i\right) \delta_t V, \tag{12.1}$$

where $\delta_t V = \rho_t (R_t + \gamma V(S_{t+1}) - V(S_t))$ is the temporal difference. $\rho_t = \min(\bar{\rho}, \frac{\pi(S_t)}{\mu(S_t)})$. $c_i = \min(\bar{c}, \frac{\pi(S_i)}{\mu(S_i)})$. π is the learner policy, which is averagely several updates ahead of the actor's policy μ.

Algorithm 3 DPPO (PPO-Penalty worker)

Hyperparameters: KL penalty coefficient λ, adaptive parameters $a = 1.5, b = 2$, the number of sub-iterations M, B
Input: initial local policy parameters θ, initial local value function parameters ϕ
for k = 0, 1, 2, ... **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy π_θ in the environment
  Compute rewards-to-go $\hat{G}_t$
  Compute advantage estimates, $\hat{A}_t$(using any method of advantage estimation) based on the current value function V_{ϕ_k}
  Store partial trajectory information
  $\pi_{\text{old}} \leftarrow \pi_\theta$
  for $m \in \{1, \dots, M\}$ **do**

$$J_{PPO}(\theta) = \sum_{t=1}^{T} \frac{\pi_\theta(A_t|S_t)}{\pi_{\text{old}}(A_t|S_t)} \hat{A}_t - \lambda \text{KL}[\pi_{\text{old}}|\pi_\theta] - \xi \max(0, \text{KL}[\pi_{\text{old}}|\pi_\theta] - 2\text{KL}_{\text{target}})^2$$

    if $\text{KL}[\pi_{\text{old}}|\pi_\theta] > 4\text{KL}_{\text{target}}$ **then**
      break and continue with next outer iteration $k + 1$
    end if
    Compute $\nabla_\theta J_{PPO}$
    send gradient wrt. θ to chief
    wait until gradient accepted or dropped; update parameters
  end for
  for $b \in \{1, \dots, B\}$ **do**
    $L_{BL}(\phi) = -\sum_{t=1}^{T}(\hat{G}_t - V_\phi(S_t))^2$
    Compute $\nabla_\phi L_{BL}$
    send gradient wrt. ϕ to chief
    wait until gradient accepted or dropped; update parameters
  end for
  Compute $d = \hat{\mathbb{E}}_t\,[\text{KL}[\pi_{\text{old}}(\cdot|S_t), \pi_\theta(\cdot|S_t)]]$
  if $d < d_{\text{target}}/a$ **then**
    $\lambda \leftarrow \lambda/b$
  else if $d > d_{\text{target}} \times a$ **then**
    $\lambda \leftarrow \lambda \times b$
  end if
end for

Moreover, there can be multiple learners, separated as worker learners and master learner. Each learner interacts with different actors and finishes model training independently. Periodically, all worker learners communicate with the master learner with learning gradients and the master announces the update of the network parameters synchronously.

The Scalable, Efficient, Deep-RL (SEED) architecture (Espeholt et al. 2019) is closely related with the IMPALA. The key difference is that the inference policy network is moved from the actor to the learner, which reduces the computation requirement for the actor and decreases the communication latency. The detailed SEED architecture is shown in Fig. 12.9. As each actor implements one or multiple

Algorithm 4 DPPO (PPO-Clip worker)

Hyperparameters: clip factor ϵ, the number of sub-iterations M, B
Input: initial local policy parameters θ, initial local value function parameters ϕ
for k = 0, 1, 2, ... **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy π_θ in the environment
  Compute rewards-to-go $\hat{G}_t$
  Compute advantage estimates, $\hat{A}_t$(using any method of advantage estimation) based on the current value function V_{ϕ_k}
  Store partial trajectory information
  $\pi_{\text{old}} \leftarrow \pi_\theta$
  for $m \in \{1, \ldots, M\}$ **do**
    Update the policy by maximizing the PPO-Clip objective:

$$J_{PPO}(\theta) = \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(A_t|S_t)}{\pi_{\text{old}}(A_t|S_t)}\hat{A}_t, \text{clip}(\frac{\pi(A_t|S_t)}{\pi_{\text{old}}(A_t|S_t)}, 1-\epsilon, 1+\epsilon)\hat{A}_t\right)$$

    Compute $\nabla_\theta J_{PPO}$
    send gradient wrt. θ to chief
    wait until gradient accepted or dropped; update parameters
  end for
  for $b \in \{1, \ldots, B\}$ **do**
    Fit value function by regression on mean-squared error:

$$L_{BL}(\phi) = -\frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(S_t) - \hat{G}_t\right)^2$$

    typically via some gradient descent algorithm
    send gradient wrt. ϕ to chief
    wait until gradient accepted or dropped; update parameters
  end for
end for

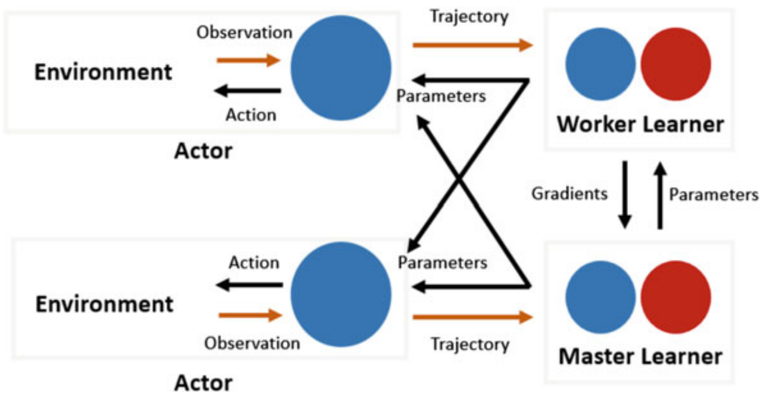

Fig. 12.8 IMPALA architecture

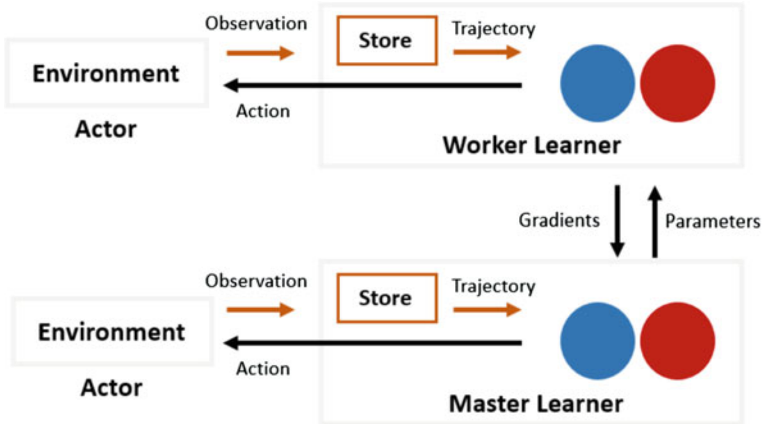

Fig. 12.9 SEED architecture

environments only, all kinds of machines with weak computation power can be regarded as the actor in the architecture. Based on the instructed actions from the learner, the actor provides one-step feedback experience to the learner and the experience is stored in the experience buffer within the learner. After several iterations, the trajectory data is applied to model training, where V-trace target in Eq. (12.1) is also applied as the value approximation.

12.4.5 Ape-X, Reactor, and R2D2

In distributed network, considering the multiple interactions between agents and environments, it is scalable and beneficial to extend prioritized experience replay to the architecture. Ape-X (Horgan et al. 2018) is the typical distributed architecture including prioritized experience replay. As shown in Fig. 12.10, there exist multiple independent actors. Within each actor, an agent interacting with an environment with the guidance from the policy network. Based on the experience collected from multiple actors, the learner train the network parameters and learn the optimal policy. Most importantly, apart from the actor and learner, there exists a replay buffer collecting the experience from actors, updating the priorities of each experience entry and batching the prioritized ones to the learner for model training. The batched prioritized experiences improve the computation efficiency and model learning performance.

The algorithms from the perspective of each actor are elaborated in Algorithm 5. Each actor initially synchronizes with the learner on network parameters. The updated parameters then instruct the agent to interact with the environment. Receiving the feedback from the environment, the actor calculates the priorities of the explored experience and sends both the data and priority information to the replay buffer.

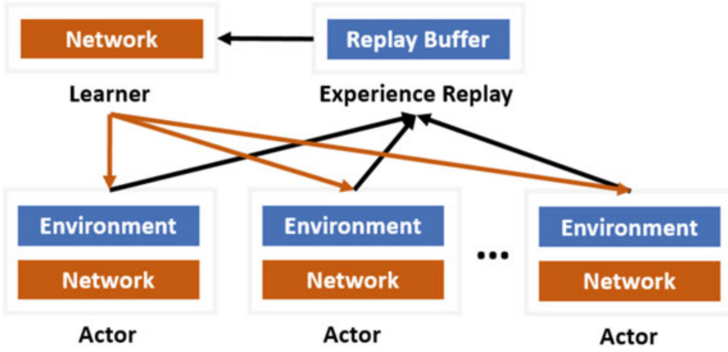

Fig. 12.10 Ape-X architecture

Algorithm 5 Ape-X (Actor)

Hyperparameters: Send to replay with batch size B in local buffer. Number of iterations T
Sync with learner to obtain latest network parameters θ_0.
Get initial state S_0 from environment.
for t = 0, 1, 2, ..., $T - 1$ **do**
  Choose action A_t based on policy $\pi(S_t|\theta_t)$
  Add experience (S_t, A_t, R_t, S_{t+1}) to the local buffer
  if The local buffer reaches its size requirements B **then**
    Get buffered data with batch size B
    Calculate the priorities p of the buffered data.
    Send the batched buffered data and its priorities to the replay
  end if
  Periodically sync and update the latest network parameters θ_t
end for

When the replay buffer collects certain amount of experiences from the actors, the learner interacts with the replay buffer for learning. The algorithm from the perspective of the learner is shown in Algorithm 6. For each episode of model learning, the learner firstly samples prioritized batch of experience data from the replay buffer. Each data entry is represented as (i, d), where i denotes the index of the data and d is the detailed information of the experience including state, action, reward, and next state. The batched data is employed to train network parameters of the learner, which will periodically synchronize with network parameters of all actors. After model training, the priorities of the sampled data are adjusted and updated in replay buffer. Due to the limits of the replay buffer size, periodically, the data with low priorities will be removed in replay buffer.

Following the general architecture, Ape-X DQN and Ape-X DPG are proposed when the learning model follows the DQN and DPG algorithm, respectively. In Ape-X DQN, the Q-network exists in the learner and all actors. The actions for the actor are guided with the Q values from the network. In Ape-X DPG, both policy

Algorithm 6 Ape-X (Learner)

Hyperparameters: Number of learning episodes T.
Initialize the network parameters θ_0.
for $t = 1, 2, 3, \ldots, T$ **do**
  Sample a prioritized batch of data (i, d) from replay
  Applying training with the batched data
  Update network parameters to θ_t
  Calculate the priorities p for batched data d
  Update the priorities p for data with index i on replay
  Periodically remove data with low priorities in replay
end for

network and value network exist in learner, while each actor only replicates the policy network to instruct the actions.

Built upon the prioritized distributed replay, Retrace-Actor (Reactor) (Gruslys et al. 2017) is further put forward based on the actor-critic architecture. Instead of the single experience, the sequence of experiences are pushed into the buffer and distributional Retrace(λ) algorithm is implemented to update the estimation of Q values. In the perspective of the neural network, the LSTM network is added in both policy and value network for better model learning.

Similarly, Recurrent Replay Distributed DQN (R2D2) (Kapturowski et al. 2018) applies fixed-length sequence of experiences in prioritized distributed replay. Developed from the DQN, R2D2 implements LSTM layer in the network and train the LSTM from replay with stored states.

12.4.6 Gorila

Implemented from the Deep Q-Network algorithm, general reinforcement learning architecture (Gorila) (Nair et al. 2015) is depicted in Fig. 12.11. Synchronizing the parameters of deep Q-network from the parameter server, the actors interact with the environment based on the policy instructed by the deep Q-network. The experiences received from the interactions are instantly forward to the replay buffer. The replay buffer stores and maintains the collected experience from all actors. Fetching the batched experience data from the replay buffer, the learner applies model learning and calculates the gradients of the Q-network. Within the learner, there are one learning Q-network and one target Q-network to calculate the TD-error. The learning Q-network sync with the parameter sever for each step of learning, while the target Q-network sync with the parameter server every N steps. Periodically, the parameter server receives the gradients from the learner and updates the network parameters for future explorations.

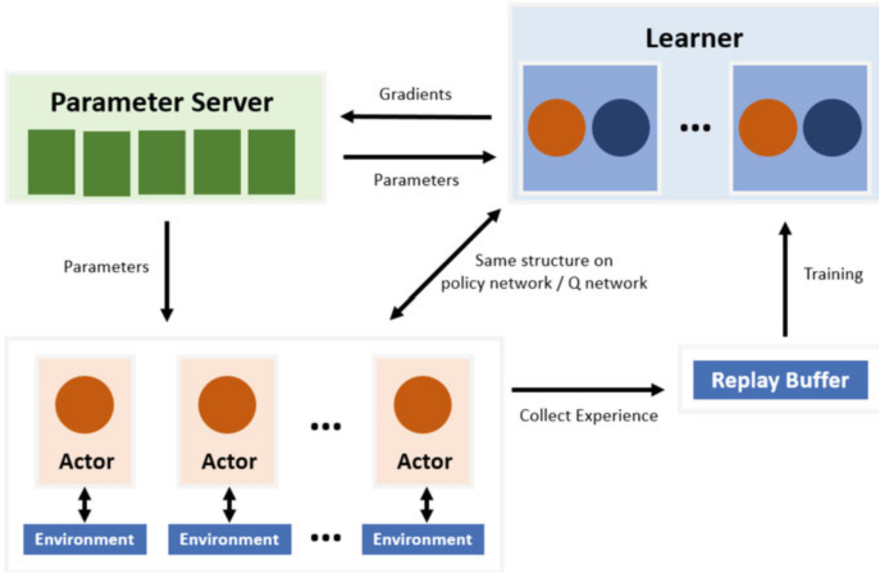

Fig. 12.11 Gorila architecture

12.5 Distributed Computing Architecture

Based on the basic patterns and structures in parallel computing, in distributed reinforcement learning, the large-scale parallel computing architecture can be further explored and investigated. Generally, the system can be composed of the following basic components:

- **Environments**: The environment is the component the agent interact with. In large-scale parallel computing of deep reinforcement learning, the environment can have multiple replicas which are mapped to different replicas of actors to gain experience in parallel fashion. Moreover, in model-based reinforcement learning, the model can also be regarded as simulated environment in the system to help learning in parallelization.
- **Actors**: The actor concept in the system refers to the component directly interacting with the environment. There can be multiple actors mapping to single or multiple real or simulated environments, and each actor is able to make actions independently with each other in the assigned environment. The action is determined by its own or shared policy network or Q-network from the parameter servers or its corresponding learners. With multiple actions applied sequentially in the environment, trajectories are formed, which will be pushed into the replay memory buffers or directly fed into the learners. As interactions between actor and environment can be costly in time, the parallelization on actors can improve

the speed to generate experiences and contributes to the training performance for learners.

- **Replay Memory Buffers**: The replay memory buffer is the component to collect the planning trajectories from actors and provide to learners for policy learning or Q learning. As the memory buffer requires to perform fast data writing, shuffling, and data reading, the storage structure should also support in dynamic and parallel fashion. Moreover, as most learners reply on the data in replay memory buffer for training, it is recommended to allocate reply memory buffers closely connected with learners, so as to guarantee the learning efficiency.
- **Learners**: The learners are the key component for deep reinforcement learning. Based on different deep reinforcement learning algorithms, the structure for each learner will be different. Normally, each learner maintains a policy network or Q-network and trains the deep network weights based on the actors' experience in replay memory buffers. Before and after training, the learner will communicate with parameter servers for synchronization on deep network weights or its learning gradients. Either synchronous or asynchronous communication approach can be applied between multiple learners and parameter servers.
- **Parameter Servers**: The parameter server is the component to collect all information from the learners and maintain the weights of policy network or Q-network in general. The parameter server will periodically synchronize with all learners for weight updates and assist all learners to start learning based on the training results from other learners. Moreover, the parameter server can instruct the actors for its interaction with the environments. In large-scale deep reinforcement learning system, in order to guarantee robustness and efficiency for data interactions from parameter servers to learners and actors, the parameter severs can also be in different kinds of structures and the communication among parameter servers can be centralized or distributed.

The general computing architecture is combined based on the above components. As parallel computing is applicable within each component, the structure can be flexible and easily adapted for the requirements from problems.

References

Babaeizadeh M, Frosio I, Tyree S, Clemons J, Kautz J (2017) Reinforcement learning through asynchronous advantage actor-critic on a GPU. In: International conference on learning representations

Espeholt L, Soyer H, Munos R, Simonyan K, Mnih V, Ward T, Doron Y, Firoiu V, Harley T, Dunning I, et al (2018) IMPALA: scalable distributed deep-RL with importance weighted actor-learner architectures. Preprint. arXiv:180201561

Espeholt L, Marinier R, Stanczyk P, Wang K, Michalski M (2019) SEED RL: scalable and efficient Deep-RL with accelerated central inference. Preprint. arXiv:191006591

Gruslys A, Dabney W, Azar MG, Piot B, Bellemare M, Munos R (2017) The reactor: a fast and sample-efficient actor-critic agent for reinforcement learning. Preprint. arXiv:1704.04651

Horgan D, Quan J, Budden D, Barth-Maron G, Hessel M, van Hasselt H, Silver D (2018) Distributed prioritized experience replay. Preprint. arXiv:1803.00933

Kapturowski S, Ostrovski G, Quan J, Munos R, Dabney W (2019) Recurrent experience replay in distributed reinforcement learning. In: International Conference on Learning Representations (ICLR).

Mnih V, Badia AP, Mirza M, Graves A, Lillicrap T, Harley T, Silver D, Kavukcuoglu K (2016) Asynchronous methods for deep reinforcement learning. In: International conference on machine learning (ICML), pp 1928–1937

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

OpenAI: Berner C, Brockman G, Chan B, Cheung V, Dębiak P, Dennison C, Farhi D, Fischer Q, Hashme S, Hesse C, Józefowicz R, Gray S, Olsson C, Pachocki J, Petrov M, de Oliveira Pinto HP, Raiman J, Salimans T, Schlatter J, Schneider J, Sidor S, Sutskever I, Tang J, Wolski F, Zhang S (2019) Dota 2 with large scale deep reinforcement learning. Preprint. arXiv:1912.06680