GitFL: Uncertainty-Aware Real-Time Asynchronous Federated Learning using Version Control

Ming Hu\textsuperscript{1}, Zeke Xia\textsuperscript{2}, Dengke Yan\textsuperscript{2}, Zhihao Yue\textsuperscript{2}, Jun Xia\textsuperscript{2}, Yihao Huang\textsuperscript{1}, Yang Liu\textsuperscript{1}, Mingsong Chen\textsuperscript{2,3}
\textsuperscript{1}School of Computer Science and Engineering, Nanyang Technological University, Singapore
\textsuperscript{2}MoE Engineering Research Center of SW/HW Co-Design Tech. and App., East China Normal University, China
\textsuperscript{3}MOE International Joint Lab of Trustworthy Software, East China Normal University, China

Abstract—As a promising distributed machine learning paradigm that enables collaborative training without compromising data privacy, Federated Learning (FL) has been increasingly used in large-scale AIoT (Artificial Intelligence of Things) system design. However, due to the lack of efficient management of straggling devices, existing FL methods greatly suffer from the problems of long response time (e.g., training and communication latency) and low inference accuracy. Things become even worse when taking various uncertain factors (e.g., network delays, performance variances caused by process variation) existing in AIoT scenarios into account. To address this issue, this paper proposes a novel asynchronous FL framework named GitFL, whose implementation is inspired by the famous version control system Git. Unlike traditional FL, the cloud server of GitFL maintains a master model (i.e., the global model) together with a set of branch models indicating the trained local models committed by selected devices, where the master model is updated based on all the pushed branch models and their version information, and only the branch models after the pull operation are dispatched to devices. By using our proposed Reinforcement Learning (RL)-based device selection mechanism, a pulled branch model with an older version will be more likely to be dispatched to a faster and less frequently selected device for the next round of local training. In this way, GitFL enables both effective controls of model staleness and adaptive load balance of versioned models among straggling devices, thus avoiding performance deterioration while ensuring real-time performance. Comprehensive experimental results on well-known models and datasets show that, compared with state-of-the-art asynchronous and synchronous FL methods, GitFL can achieve up to 2.64X training acceleration and 7.88\% inference accuracy improvements in various uncertain scenarios.

Index Terms—AIoT, Asynchronous Federated Learning, Uncertainty, Reinforcement Learning, Version Control

I. INTRODUCTION

Along with the prosperity of Artificial Intelligence of Things (AIoT) [1], Federated Learning (FL) [2, 3, 4, 5, 6] is becoming more and more popular in the AIoT applications, e.g., autonomous driving [7, 8], industrial control [9, 10], smart remote sensing [11], and healthcare systems [12, 13]. As a promising distributed collaborative learning paradigm, FL adopts a cloud-based architecture, where AIoT devices focus on local training based on their raw data while the cloud server is responsible for both knowledge aggregation and dispatching. Instead of accessing local data hosted by devices directly, in each FL communication round the cloud server conducts the knowledge aggregation operation by averaging the gradients sent by its selected devices to form a global model for the next round of training. In this way, the inference performance of all the involved devices is improved, while device privacy can be safely guaranteed.

Although FL enables knowledge sharing among AIoT devices without compromising data privacy, existing FL methods inevitably suffer from the problems of low inference performance and long overall training time, caused by i) heterogeneous devices, ii) imbalanced data, and iii) uncertain environment. Generally, a typical AIoT system involves a variety of heterogeneous devices, which have different computation and communication capabilities (e.g., CPU/GPU frequencies, memory size, network bandwidths). In this case, devices with lower performance (a.k.a., stragglers) will take a longer response time (e.g., training and communication latency) than the other selected devices. This is harmful to the most widely-used synchronous FL methods such as Federated Average (FedAvg) [2], since it requires the cloud server to wait for the responses from all the selected devices before aggregation. According to the “wooden barrel theory”, when applying synchronous FL methods, the training time of one FL communication round will be determined by the slowest selected devices. As a result, the existence of stragglers will significantly slow down the process of overall FL training. Worse still, due to concerns about privacy protection, the performance parameters and resource usage of devices are often difficult to obtain. This makes it difficult for FL to perform reasonable device scheduling in advance. In the case of resource constraints, FL is difficult to fully train the model within a given deadline.

Typically, in an AIoT system, due to the difference in located environments and user preference, the amount and distribution of collected data among massive devices are quite different, which results in the “client drift” problem [14, 15]. In specific, since the data among the devices are Non-IID (not Identically and Independently Distributed) [16, 17], the optimal direction of each device is different, which causes serious inference accuracy degradation of the aggregated global model. Moreover, different from the conventional FL applications, due to being deployed into the uncertain physical environment, AIoT devices are indeed susceptible to environmental changes, which can impact their performance and network communication. For example, since AIoT devices often operate on battery power or have limited energy resources, extreme temperatures or fluctuations in power supply, can significantly
impact the energy efficiency and overall performance of these devices, potentially leading to reduced functionality or system failures. Communication between AIoT devices inevitably depends on wireless networks. In an uncertain physical environment, interference, obstacles, and distance can weaken wireless connectivity and affect the signal strength between AIoT devices and the network. This can result in slower data transmission, increased latency, or even intermittent network connections. In addition, AIoT devices deployed in harsh or hazardous environments face additional challenges. Exposure to extreme temperatures, humidity, dust, vibrations, or physical impacts can damage the devices, impair their performance, or even render them inoperable.

To mitigate the shortcomings caused by stragglers in FL, the most of current methods are based on three schemes, i.e., synchronous scheme [16, 18, 19], asynchronous scheme [20, 21], and semi-asynchronous scheme [22, 23]. Synchronous FL methods attempt to deal with stragglers by using a wise device selection strategy [16], model pruning technology [18], or a virtual deadline [19]. However, these methods lead to low participating frequencies of low-performance devices, which results in a decrease in inference performance. Unlike synchronous FL, in each training round, the asynchronous cloud server performs an update of the global model in real-time once it receives a local model from some device [24]. However, since each local model is trained from the global model with a different version in asynchronous FL methods, the stale local models trained by stragglers can easily result in poor training quality, especially for Non-IID scenarios. As a trade-off between synchronous and asynchronous FL methods, by forcing to synchronize multiple stale local models in one training round, semi-asynchronous FL methods (e.g., SAFA [22] and FedSA [23]) can be used to partially alleviate the model staleness issue. However, such semi-asynchronous FL methods also bring new problems. For example, due to large staleness discrepancies, the cloud server may frequently synchronize with some stragglers before they complete their local training. In this case, the stragglers may not have the opportunity to participate in global model updating, thus biasing the training of global models. Things become even worse when various uncertain factors [25] are taken into account in AIoT scenarios, since both asynchronous and semi-asynchronous FL methods assume that the performance metrics (e.g., network delay and computation capacities) of AIoT devices are fixed. However, this is not true in practice. For example, due to process variations during the manufacture of microelectronic circuits, the performance of devices with the same type varies significantly [26]. Due to the over-pessimistic estimation of the performance metrics, the potential of asynchronous FL is strongly suppressed. Clearly, due to the lack of efficient management mechanisms for stale models within an uncertain environment, it is hard for existing asynchronous FL methods to quickly converge to a higher inference performance.

Inspired by the implementation of Git, a famous version control system, this paper proposes a novel asynchronous FL approach named GitFL, which enables efficient management of straggler devices to accommodate various uncertainties in an adaptive manner. Unlike traditional FL methods, GitFL does not dispatch its global model directly to its selected devices for local training. Instead, it keeps a set of branch models that can traverse different devices asynchronously and dispatches new branch models pulled from the global model to devices. Here, the global model is only used to accumulate the versioned knowledge obtained by the branch models. To support the adaptive load balance of version models on straggler devices, GitFL is apt to dispatch a pulled branch model with an older version to a faster and less frequent device for the next round of local training. Note that GitFL incorporates a Reinforcement Learning (RL)-based device selection strategy that eliminates the need to obtain device performance parameters. Instead, it leverages the recorded training times of each device to adaptively perceive their real performance, enabling informed device selection. Our RL algorithm continuously updates during the training process, which enables GitFL to promptly sense performance changes in the devices in real-time. In this way, GitFL ensures both the real-time performance of FL training and the inference performance of the model. In summary, this paper makes the following three major contributions:

- Based on the push, pull, and merge operations, we established a novel Git-like asynchronous FL framework, which supports both effective management of straggler devices and version control of their stale models within various uncertain environments.
- We develop a new Reinforcement Learning (RL)-based curiosity-driven exploration method to support the wise selection of straggler devices in an adaptive manner. Our approach enables the load balance of versioned models among AIoT devices, thus avoiding the performance deterioration caused by stragglers.
- We conduct experiments on both well-known datasets and models to show the effectiveness of our approach in terms of convergence rate and inference performance for both IID and non-IID scenarios.

The rest of this paper is organized as follows. Section II introduces the background of FL and RL. Section III presents the motivation of this paper. Section IV details the implementation of our GitFL approach. Section V gives the experimental results. Section VI introduces the related work of existing FL methods. Finally, Section VII concludes this paper.

II. PRELIMINARIES

A. Federated Learning

The traditional FL framework consists of a central cloud server and multiple clients. The cloud server maintains a global model for FL training. In each FL training round, the cloud server dispatches the global model to multiple clients. Then, each client uses its local data for local training and uploads the trained model to the cloud server. Finally, the cloud server generates a new global model by aggregating all the uploaded
models. Typically, the model aggregation is based on FedAvg [2], which is defined as follows:

\[\min_w F(w) = \frac{1}{K} \sum_{k=1}^{K} f_k(w), \quad \text{s.t.,} \quad f_k(w) = \frac{1}{|d_k|} \sum_{i=1}^{|d_k|} \ell(w, (x_i, y_i)),\]

where \(K\) is the number of clients that participate in local training, \(d_k\) is the dataset of the \(k\)th client, the \(|d_k|\) denotes the number of data samples in \(d_k\), \(\ell\) denotes the customer-defined loss function (e.g., cross-entropy loss), \((x_i, y_i)\) denotes a sample \(x_i\) with its label \(y_i\).

B. Reinforcement Learning

As an important branch of machine learning, Reinforcement Learning (RL) is widely used in decision-making tasks such as real-time system design [27, 28], task scheduling [29, 30], and automatic control [31, 32]. In RL, the decision-maker controls the agents to interact with the environment and calculates a reward value based on feedback from the environment. The goal of RL is to train a policy \(\pi\) to control the agents to get the maximum reward when they reach the terminal state. An RL process can be modeled as a Markov Decision Process (MDP)[33], which can be presented as a four-tuple \((\mathcal{S}, \mathcal{A}, \mathcal{F}, \mathcal{R})\), where:

- \(\mathcal{S}\) denotes a set of states, which presents the information for all the agents and environments.
- \(\mathcal{A}\) denotes a set of actions, which specifies candidate behaviors for each agent to interact with the environment.
- \(\mathcal{F}\) : \(\mathcal{S} \times \mathcal{A} \to \mathcal{S}\) is a transition function from states to their successors caused by actions.
- \(\mathcal{R}\) : \(\mathcal{S} \to \mathbb{R}\) is a reward function to evaluate the actions according to the feedback from the environment.

The most classical RL algorithm is Q-Learning [34, 35], which is based on a Q-Table to record the maximum rewards of each state that agents can get when performing a specific action. \(Q(s, a)\) in Q-Table records the expectation of the maximum reward for taking action \(a \in \mathcal{A}(s)\) in state \(s \in \mathcal{S}\). In the training process of Q-learning, there are multiple rounds of search. In each round, the Q-Table will be updated using the Bellman equation as follows:

\[Q(s, a) \leftarrow Q(s, a) + \alpha [R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)],\]

where \(\alpha\) is a learning rate, \(\gamma\) indicates a discount rate, \(s'\) is the next state after perform the action \(a\) from \(s\), and \(\max_{a'} Q(s', a')\) represents the maximum reward expectation for the state \(s'\). The equation makes the value of \(Q(s, a)\) to the maximum reward expectation through iterative learning.

III. MOTIVATION

Figure 1 illustrates both the synchronous and asynchronous FL schemes, indicating the motivation of our GitFL approach. In this example, there are nine AoT devices, which can be divided into three different types depending on their computation capability (i.e., low-quality, medium-quality, and high-quality, respectively). The cloud server selects three devices for local training in each FL round. Figure 1(a) presents the three types of heterogeneous AoT devices. Due to the uncertainty of the environment, the response time of each device may fluctuate.

Limitations in synchronous and asynchronous FL. Figure 1(b) presents the example of the synchronous FL scheme. It suffers from inefficiencies due to the inclusion of low-quality devices \(D_0\) and \(D_4\), which results in the waste of a significant amount of time in each round of local training for the cloud server. Although the cloud server allows three devices to participate in local training, one or two of these devices remain idle for a considerable portion of the time. This limitation becomes particularly problematic in scenarios with time constraints, as synchronous FL cannot ensure the full participation of all devices within a given deadline. In Figure 1(b), for example, devices \(D_4\), \(D_6\), \(D_7\), and \(D_8\) are unable to participate in local training.

Figure 1(c) illustrates an example of the asynchronous FL scheme, which can ensure the participation of all devices within a given deadline. In this scheme, the cloud server directly aggregates the received local model to generate a global model and then activates a device for local training. In this way, the FL system consistently maintains three activated devices engaged in local training throughout the process. As a result, all devices participate in the training within the specified deadline. Due to aggregating stale models, the asynchronous FL encounters the problem of low inference accuracy. As shown in Figure 1(c), after local training on \(D_4\), the cloud server has to aggregate the stale local model trained by three devices (i.e., \(D_1\), \(D_2\), and \(D_3\)), while the global model has been updated seven times.

Intuition. To address the problem of stale models in asynchronous FL, the motivation of our approach is from two perspectives, i.e., model aggregation mechanism and device selection strategy, respectively. One common strategy is to
assign lower weights to stale models to reduce the impact of these models on the aggregation process. However, by using this strategy, the global model cannot adequately learn knowledge from the stragglers. Inspired by Git, we treat each uploaded local model as a branch model and assign a version number to each branch model to track the number of its trained times. Each branch model pushes their parameters to generate a global model and then merges with the global model according to the specific weights. In this way, the branch model can pull knowledge from the global model while retaining its learned knowledge. Importantly, we utilize the branch models rather than the global model for local training. For the device selection strategy, we prefer to dispatch the model with a low version to a high-performance device, thus balancing the versions among branch models. As shown in Figure 1(d), if a branch model is trained by a straggler, it will be dispatched to a high-performance device for the next round of local training. Eventually, all the branch models have similar versions, which means that all the branch models have a similar weight to generate the global model. It is worth noting that despite the branch models having different versions, they are all trained toward the same optimization direction since they merge with the global model during model updates. Intuitively, in this way, we ensure that the knowledge of stragglers can be fully learned while preventing the inference accuracy drop caused by aggregating stale models.

IV. OUR GITFL APPROACH

A. Overview

Figure 2 presents the framework and workflow of our GitFL approach, which consists of a central cloud server and multiple clients. The cloud server contains two key components, i.e., the model version controller and the RL-based client selector. The model version controller updates the master model and branch models asynchronously according to the version of each branch model. The RL-based client selector chooses clients for local training according to the number of past selections, historical training time, and the version of the dispatched branch model.

The model version controller includes a model repository to store all the branch models with its version, an uploaded queue to receive trained branch models, and a dispatched queue for assigning updated branch models. The RL-based client selector maintains a client time table and a count table, respectively. The client time table records the average training time of each client and the count table records the selected number of each client. In detail, to perform timekeeping for each model, it starts when the target model is assigned to local clients from the dispatched model queue. After the model is trained in local clients and pushed to the uploaded model. The update of tables is in real-time. In this way, the cloud server can promptly sense the performance of each client according to the training time in real-time. With the help of these two tables, the client selector can prompt all clients adequately participate in training, thus reducing the version difference between branch models. Please note that the selection strategy is according to the version of a target branch model and the two tables which record the operating efficiency of local clients. The branch model with a higher version is preferred to assign to a client with a longer training time, and vice versa. For a model with a medium version, the selector prefers to select the client with less number of selections. As shown in Figure 2, each iteration of a branch model in GitFL includes five key steps:

- **Step 1 (Model Merging):** The controller merges all the branch models in the repository to generate a new master model according to weights calculated by the version of each branch model. To avoid the inference degradation caused by stale models, here a branch model with a low version is assigned a low weight.

- **Step 2 (Model Pulling):** The controller updates the dispatched branch model by merging it with the new master model and pushing it into the dispatched model queue. Here the model with a low version is assigned a low weight for merging, which can make such a stale model learn more knowledge from the master model.

- **Step 3 (Client Selection):** The client selector selects a client for branch model dispatching by using an RL-based client selection strategy, which is presented in...
Section IV-D.
- **Step 4 (Local Training):** The selected client trains the received branch model using its local data. Note that, to protect privacy, our method does not need to obtain any additional information of clients, so the local training process is the same as traditional FL [2].
- **Step 5 (Model Pushing):** The controller pushes the branch model into the repository and updates its version and its corresponding items in the client time table and the curiosity table.

Figure 2(b) illustrates an example of the workflow of GitFL, which presents a whole FL iteration of $m_1$, $m_2$, and $m_K$. In Figure 2(b), the repository has $K$ branch models, where $K$ equals the number of activated clients and the version of $m_1$, $m_2$, and $m_K$ are all equal to three, which means all the three models have participated in local training three times. When receiving the branch model $m_1$ from $c_1$, the cloud server pushes this model into the repository and updates its version to four. Here each branch model has a fixed storage location in the repository. Then, the cloud server merges all the models in the repository to generate a new master model. Here the weight of $m_1$ is larger than $m_2$ and $m_K$ since its version is higher than the version of $m_2$ and $m_K$. Next, the cloud server pulls the master model and merges it with $m_1$ to update a new $m_1$. Note that the updated $m_1$ is not stored in the repository. Finally, the client selector selects the client $c_3$ and dispatches the updated $m_1$ to $c_3$ for local training. Similarly, after local training in $c_2$, $m_K$ is pushed to the repository. After the model merging and model pulling process, the updated $m_K$ is dispatched into $c_2$. When received $m_2$, the cloud server performs the same process as that of $m_1$ and $m_K$. Finally, $m_K$ is dispatched into $c_3$ for local training.

### B. GitFL Implementation

Algorithm 1 presents the implementation of our GitFL approach. We assume that there are at most $K$ clients participating in local training at the same time, which means that GitFL adopts $K$ branch models for local training. Line 2 initializes the model repository $R$, which is implemented as a list to store all the branch models. Line 3 initializes a list $V$, which records the version information of all the branch models. Line 4 initializes the count table $T_c$ and the client time table $T_t$, which is used to record the number of selected times and the average training time of each client, respectively. Line 5 sets the function $Time()$ to record the start time of FL training and is used to get the timestamp in real-time. Lines 7-21 present the FL training process of $K$ branch models, where the “for” loop is a parallel loop. Lines 8-20 present the detail of the training iteration of each branch model. In Line 9, the function $Merging(\cdot)$ merges all the branch models in $R$ to generate a master model $M$. In Line 10, the function $ModelPull(\cdot)$ pulls down the generated master model to aggregate it with the $i$th branch model according to the version list $V$, where $m_i$ is the aggregated model. In Line 11, the function $ClientSel(\cdot)$ selects a client according to $T_c$, $T_t$, and the version of a dispatched branch model. Line 13 dispatches the branch model to the selected client for local training. In Line 12, $t_{start}$ uses the timestamp to record the dispatching operation of a branch model. In Line 14, $t_{end}$ uses the timestamp to record the receiving operation of a branch model. In Line 15, $t$ records the time of a whole local training process of a specified branch model, which includes the time of network communication and local model training. After receiving the branch model, Line 16 updates its version and Line 17 pushes it to the model repository. In addition, Line 18 and Line 19 update $T_c$ and $T_t$, respectively.

#### C. Model Version Control

GitFL adopts a model version control strategy to guide knowledge sharing between the master model and branch models, which consists of three key operations, i.e., model merging, model pulling, and model pushing. To implement version control, GitFL sets a model repository that stores all branch models and their version information.

1) **Model Merging:** GitFL generates the master model by merging all branch models in the model repository. Since the branch model with a lower version may reduce the performance of the master model, GitFL gives each branch model a weight based on its version to guide the merging. The merging
process is shown as follows:

$$Merging(R, V) = \frac{\sum_{k=1}^{K} R[k] \times V[k]}{\sum_{k=1}^{K} V[k]}$$

(1)

where \(R\) denotes the model repository, \(K\) presents the number of branch models, and \(V\) is the list of version information. Here, we directly use model versions as the merging weights instead of the difference of versions. At the beginning of training, models with a higher version should dominate the merging, since they are more accurate. However, along with the progress of FL training, all the models are becoming well-trained. At this time, either model versions or differences of versions are both reasonable to be used as weights. Compared with differences of versions, model versions are more likely to equally merge the models.

2) Model Pulling: To enable knowledge sharing across branch models, each branch model needs to pull down the master model before model dispatching. In each pulling step, GitFL updates a branch model by aggregating it with the master model, and the aggregation weight depends on the version of this branch model. Here, a branch model with a higher version will be assigned a larger weight. This is because, on the one hand, the models with higher versions are more likely to be well-trained, which means they only need less knowledge from the master model. On the other hand, the knowledge of stragglers contained in the master model may have negative effects on the models with higher versions. For a model with a lower version, due to insufficient training, it needs to learn more knowledge from the master model. Therefore, it is assigned a lower weight in aggregation. Based on the above consideration, GitFL uses a version control variable to guide the aggregation as follows:

$$v_{ctrl}^i = V[i] - \frac{1}{K} \sum_{k=1}^{K} V[k]$$

(2)

where \(i\) denotes the branch model index, \(v_{ctrl}^i\) denotes the difference between the version of \(i^{th}\) branch model and the average version of all branch models. Based on this, the model-pulling process is defined as follows:

$$ModelPull(i, V, M, m_i) = \frac{\max(10 + v_{ctrl}^i \times 2) \times m_i + M}{\max(10 + v_{ctrl}^i \times 2) + 1}$$

(3)

where \(M\) is the master model generated by Equation (1). Note that the excessive weight of the master model in model pulling makes the update of a branch model too coarse-grained, which may lead to the gradient divergence problem. Therefore, we set the weight of each branch model greater than that of the master model, which enables the branch model to acquire knowledge from the master model more smoothly and fine-grained.

3) Model Pushing: When the uploaded queue receives a trained branch model, the model version controller pushes this received branch model to the model repository and updates its version information. As shown in Figure 2(b), after receiving the model \(m_K\), the model repository updates the version of \(m_K\) from 3 to 4. In addition, the model repository will replace the old version of a model with the new one. Note that for each branch model, the model repository only reserves the latest version. Therefore, in GitFL, the model repository only stores \(K\) models.

D. RL-based Client Selection

Although GitFL uses the master model as a bridge to implement asynchronous training of branch models, due to the uncertainty of clients and network, the version gaps between branch models lead to a low-performance problem. To balance the versions of branch models, we design a reinforcement learning strategy to select clients based on the branch model version, the historical training time of clients, and the selection number of clients. Note that, the historical training time of each client can be recorded by the cloud server during the FL training process and our client selection strategy does not need to require any privacy information about clients.

1) Problem Definition: In GitFL, the goal of the client selection strategy is to balance the versions of branch models. In other words, the client selector prefers to select a client with a faster training time for the branch model with a low version and select a client with a slower training time for the branch model with a high version. Therefore, the client selection depends on i) the version of the target branch model and the average version of all the branch models, and ii) the training time of all the candidate clients. In addition, to balance the number of times that the client participates in training, we also use the number of selected times of each client as one of the metrics for client selection.

The client selection process can be defined as a Markov Decision Process (MDP), which is presented as a four-tuple \(M = (S, A, F, R)\) as follows:

- \(S\) is a set of states. We use a vector \(s = (B_m, V, C_t, T_c)\) to denote the state of GitFL, where \(B_m\) denotes the set of branch models that waits for dispatching, \(V\) is the list of version information for all the branch models, \(C_t\) indicates the set of clients for current local training, and \(T_c\) is a table that records the number of selected times of all the clients.
- \(A\) is a set of actions. At the state of \(s = (B_m, V, C_t, T_c)\), the action \(a\) aims to select a candidate client in \(C_t\) and then dispatches the branch model in \(B_m\) to the selected client. The selection space of \(A\) is all the clients in \(C_t\).
- \(F\) is a set of transitions. It records the transition \(s \xrightarrow{a} s'\) with the action \(a\).
- \(R\) is the reward function. Here, we combine the number of selected times and the training time of the selected client as the reward to evaluate the quality of a selection.

2) Version- and Curiosity-Driven Client Selection: GitFL adopts a version- and curiosity-driven strategy to evaluate rewards. The client selector prefers to select a client with a higher reward. The version-driven strategy is used to balance the version differences between branch models. For the branch model with a lower version, we prefer to dispatch it to an efficient client (i.e., less training time) and vice versa. Since the performance parameters of each client are unavailable and
the environment is uncertain, the cloud server cannot directly precisely predict the training time of each client. To address this problem, GitFL uses a client time table $T_c$ to record the average historical training time of each client. When the $i^{th}$ branch model is waiting for dispatching, the version reward for the client $c$ is measured as follows:

$$R_{v}(c, i) = \frac{1}{\sqrt{T_c[c]}}$$

where $|C|$ denotes the number of clients, $T_c[c]$ indicates the average historical training time of $c$. The difference between the model version and the average version determines the scope of $R_{v}$. Note that, the client selection is independent of client training time when the version of a branch model is equal to the average version.

To ensure the adequacy of client selection, we exploit curiosity-driven exploration [36, 37] as one of the strategies to evaluate rewards, while the client with a less number of selections will get a higher curiosity reward. GitFL uses the count table $T_c$ to record the selection number of each client and uses Model-based Interval Estimation with Exploration Bonuses (MBIE-EB) [38] to measure the curiosity reward for the client $c$ as follows:

$$R_c(c, i) = \left( \frac{\sum_{k=1}^{K} V[k]}{K} - \frac{\sum_{j=1}^{K} V[j]}{\max(T_i)} \right)$$

where $C$ denotes the number of clients, $T_i[c]$ indicates the average historical training time of $c$. The difference between the model version and the average version determines the scope of $R_c$. Note that, the initialized value of $R_c$ is 1. The reward of each client is measured by combining $R_{v}$ and $R_c$ as follows:

$$R(c, i) = \max(0, R_{v}(c, i) + R_c(c, i)).$$

If the difference between the model version and the average version is high, the reward will mainly be determined by the version reward. If the model version is close to the average version, the reward will be determined mainly by the curiosity reward. Note that the minimum value of a reward is 0. If the reward of a client equals 0, the client will not be selected. Otherwise, based on the above reward definition, the client selector chooses a client for model dispatching with the probability defined as follows:

$$P(c, i) = \frac{R(c, i)}{\sum_{j=1}^{C} R(j, i)}.$$

In this way, a client with a higher reward has a higher probability of being selected. Clearly, our client selection strategy chooses clients according to the probability but does not directly choose the client with the highest reward. The rationale behind this strategy is to avoid favoring individual clients with extremely high or low computational capabilities, which could result in their monopolization of the training process. By employing a probabilistic selection mechanism, the cloud server encourages all the clients to actively participate in the training process and prevents any small set of clients from dominating the whole FL training.

V. PERFORMANCE EVALUATION

To evaluate the effectiveness of our approach, we implemented GitFL using the PyTorch framework. We compared our GitFL method with both classical FedAvg [2] and three state-of-the-art FL methods (i.e., FedAsync [20], SAFA [22], and FedSA [23]), where the former two are asynchronous and the latter two are semi-asynchronous. For SAFA and FedSA, we set the model buffer size to half the number of activated clients. To enable fair comparison, we used an SGD optimizer with a learning rate of 0.01 and a momentum of 0.5 for both all the baselines and GitFL. For each client, we set the batch size to 50 and performed five epochs for each local training. All the experimental results were obtained from an Ubuntu workstation with an Intel i9 CPU, 64GB memory, and an NVIDIA RTX 3090 GPU.

| Quality  | Device Settings | Network Settings |
|----------|----------------|------------------|
| Excellent | N(100, 5)      | N(10, 1)          |
| High     | N(150, 10)     | N(15, 2)          |
| Medium   | N(200, 20)     | N(20, 3)          |
| Low      | N(300, 30)     | N(30, 5)          |
| Critical | N(500, 50)     | N(80, 10)         |

A. Experimental Settings

1) Settings of Uncertainties: We considered two kinds of uncertainties, i.e., the uncertainty of devices computation time caused by process variation [26], and the uncertainty caused by network delay. In the experiment, we assumed that both of them follow the Gaussian distributions. Table I shows the settings of both uncertainties, which are simulated in the FL training processes of experiments. Here, we assume that there exist five kinds of devices and five communication channels. Generally, devices and communication channels with higher quality have better computation and communication performance with lower variances as specified in the table.

2) Settings of Datasets and Models: We compared the performance of GitFL and four baselines on three well-known datasets, i.e., CIFAR-10, CIFAR-100 and FEMNIST [39]. To investigate the performance of GitFL on non-IID scenarios, we adopted the Dirichlet distribution $Dir(\alpha)$ to control the heterogeneity of client data when using CIFAR-10 and CIFAR-100 datasets, where a smaller value of $\alpha$ indicates a higher data heterogeneity on devices. Note that we did not apply $Dir(\alpha)$ on FEMNIST dataset, since FEMNIST itself is naturally non-IID distributed. For the experiments on datasets CIFAR-10 and CIFAR-100, we assumed each of them involved 100 AIoT devices in total and in each FL round there were only 10 of them selected for local training at the same time. However, for dataset FEMNIST, there are a total of 180 devices and each FL round involves 18 devices selected for local training. Moreover, to show the pervasiveness of our GitFL framework, we also take three DNN models (i.e., CNN, ResNet-18 and VGG-16) with difference sizes and structures into account. Since all the three datasets are image datasets, to evaluate
the ability of GitFL on different tasks, we also choose a text dataset Shakespeare [39]. For Shakespeare dataset, there are 128 devices with naturally non-IID distribution. We use the LSTM model for local training and set 12 devices for local training at the same time.

B. Performance Comparison

1) Comparison of Accuracy: Table II compares the test accuracy between GitFL and all the baselines on three datasets with different non-IID and IID settings using ResNet-18 model. In this table, the first column indicates the dataset type. The second column denotes the data heterogeneity settings of clients, which specify different distributions for client data. The third column has five sub-columns, which present the test accuracy information together with its standard deviation for all the five FL methods, respectively. From this table, we can find that GitFL can achieve the highest accuracy for all the cases. As an example of dataset CIFAR-10, when $\alpha = 0.1$, GitFL can achieve an improvement of 7.88% over the best inference result obtained by FedSA. We can also observe that GitFL can achieve more improvement on CIFAR-10 and CIFAR-100 datasets than that on FEMNIST. This is because the image classification tasks on these two datasets are more difficult than that on FEMNIST. All the methods can achieve a good performance on FEMNIST but GitFL still achieves the best performance compared with all the baselines.

![Learning curves of GitFL and four baseline methods on CIFAR-10](image)

2) Comparison of Training Time and Communication Overhead: Table III compares the training time and communication overhead of different settings.

![Comparison of training time and communication overhead](image)

![Comparison of training time and communication overhead](image)

![Comparison of training time and communication overhead](image)

![Comparison of training time and communication overhead](image)

Figure 3 presents the trend of learning for all the FL methods on CIFAR-10 based on ResNet-18 model. We can observe that a significant accuracy and training time improvement was made by GitFL in all the non-IID and IID scenarios. For the baselines, we can observe that compared with FedAvg, FedAsync and FedSA can also achieve a slight improvement in training time and inference accuracy. SAFA achieve the worst performance on all the baselines. We can also find that although at the early beginning of training, the inference accuracy of GitFL may be slightly lower than FedSA, GitFL can achieve better inference accuracy when all the baselines achieve their best accuracy. Note that since the model is not fully trained at the beginning of training, its inference performance will be very poor. In other words, too little training time will cause all the methods to fail to train a usable model. Therefore, from Figure 3, we can observe that
given a specific reasonable deadline, GitFL can achieve the best accuracy. Moreover, we can find that under an extreme non-IID scenario with $\alpha = 0.1$, the learning curve of GitFL is more stable than the ones of all the other baselines.

2) Comparison of Training Time and Communication Overhead: Given a specific overall FL training target in terms of accuracy, Table III compares the training time and communication overhead between GitFL and all the baselines. From this table, we can find that from the perspective of training time, GitFL outperforms all the baselines in seven out of eight cases. From the perspective of communication overhead, GitFL outperforms all the baselines in four out of eight cases. Note that here all four cases indicate the higher accuracy targets of different settings, respectively. In other words, when the accuracy target becomes higher, our approach uses much less training time and communication overhead than the other baselines. As an example of dataset CIFAR-10, when $\alpha = 0.1$ and the target accuracy is $45\%$, GitFL outperforms FedSA by 2.64X and 1.85X in terms of training time and communication overhead, respectively. Note that in this case, FedSA has the second-best result.

C. Impacts of Different Configurations

To demonstrate the pervasive and scalability of our GitFL approach on various scenarios, we checked the impacts of different configurations on GitFL from the following five perspectives: different compositions of involved devices, different number of simultaneously training clients, different numbers of total involved clients, different types of underlying AI models, and different AI tasks.

1) Impacts of Different Device Compositions: Table IV presents the four configurations for different devices used in the experiments, where the device settings are provided in Table I. Note that in each configuration, we do not force the devices with better settings to be equipped with communication modules with higher quality. For example, a device with excellent computation capability may be equipped with a critical communication module.

![Fig. 5. Learning curves for different numbers of simultaneously training clients](image)

Figure 4 presents the learning curves of the four device composition configurations, where we applied GitFL (with ResNet-18 model) on dataset CIFAR-10 with IID distribution. From Figure 4, we can find that GitFL achieves the highest inference performance for all four device compositions. Note that we can observe similar trends of learning curves for non-IID scenarios. We can also find that all the FL methods can achieve their best accuracy more quickly with Configuration 1 but slower with Configuration 3. This is because, in Configuration 1, there are more devices with strong computation and network communication capabilities. We can also observe that the convergence speeds of all the asynchronous FL methods (i.e., FedAsync and GitFL) with Configuration 2 and Configuration 4 are similar. For the synchronous FL method (i.e., FedAvg) and semi-asynchronous FL methods (i.e., FedSA and SAFA), their convergence speeds with Configuration 4 are slower than that with Configuration 1. This is because the performance of the synchronous FL method is seriously limited by stragglers and in Configuration 4 the number of devices with low performance is more than that in Configuration 2. For semi-asynchronous FL methods, since they still need to wait for a...
certain number of devices for aggregation, their performance is also affected by low-performance devices. For asynchronous FL methods, their convergence performance is mainly affected by the average performance of all the devices.

2) Impacts of Different Numbers of Simultaneously Training Clients (Activated Clients): To investigate the impacts of the number of concurrently training clients on GitFL, we considered four different activated client settings, where the numbers of simultaneously training clients are 5, 10, 20, and 50, respectively. Figure 5 shows all experimental results conducted on CIFAR-10 with IID distribution using ResNet-18. From this figure, we can find that GitFL achieves the highest accuracy with all the settings. Especially, when more clients are involved in simultaneous training, the more improvements GitFL can obtain compared with baseline methods. For baselines, we can observe that SAFA and FedAsync achieve a lower performance with a small number of activated devices and the FedSA can achieve a higher inference performance when \( K = 5 \). For FedAvg, its convergence speed becomes slower as the number of activated devices increases. Compared with all the baselines, GitFL can achieve similar high inference and convergence performance with all the settings of the number of activated devices.

3) Impacts of Number of Involved Devices: To evaluate the scalability of GitFL, we conducted experiments with four different settings of the number of involved devices, i.e., \( |C| = 50, 200, 500, \) and 1000, respectively, on CIFAR-10 dataset using ResNet-18 models. For all the settings, we select 10% devices as activated devices for local training at the same time on IID scenario. Figure 6 presents the learning curves of GitFL and all the baselines with four settings of involved devices. We can observe that compared with all the baselines, GitFL can achieve the best inference accuracy with all the settings of involved devices. For baselines, FedSA performed well with the settings of fewer involved devices (e.g., \( |C| = 50 \)), and FedAsync performed well with the settings of more involved devices (e.g., \( |C| = 1000 \)). Since with the increase of involved devices, each device is assigned fewer data samples, GitFL and all the baselines can perform well with fewer involved devices.

4) Impacts of AI Model Types: We investigate two GitFL variants with different underlying AI models, i.e., CNN and VGG-16. Figure 7 shows the impacts of such two variants with different data distributions (i.e., non-IID with \( \alpha = 0.1 \) and IID). From this figure, we can find that GitFL achieves the highest accuracy in all the cases regardless of underlying AI model types or data distributions. We can also find that when using CNN model, GitFL can achieve the best accuracy with the whole training process. When using VGG-16 model, GitFL cannot perform the best inference at the early beginning of training but can achieve better inference accuracy when all the baselines achieve their best accuracy. This is because the CNN network structure is relatively simple, and its training requires relatively less time, but VGG-16 is a connection-dense network that needs more time for training. For baselines, we can find that the convergence speeds of FedAsync and FedSA are faster than that of FedAvg and SAFA also perform the worst performance among all the baselines.

5) Impacts of AI Tasks: Since CIFAR-10, CIFAR-100, and FEMNIST are all image datasets, to demonstrate the pervasive
of GitFL on different types of tasks, we conducted experiments on the well-known text dataset Shakespeare [39] using the LSTM model for all the five FL methods. Table V presents the test accuracy of all five FL methods. We can observe that GitFL achieved the best inference accuracy. For the four baselines, FedAvg, FedAsync, and FedSA can achieve similar test accuracy and SAFA performed the worst performance. Overall, our GitFL still shows an obvious improvement in the text dataset.

### D. Ablation Study

To demonstrate the effectiveness of our RL-based client selection strategy, we developed three variants of GitFL: i) “GitFL+R” that selects clients for local training in a random manner; ii) “GitFL+C” that selects clients only based on curiosity reward (see Equation 5); and iii) “GitFL+V” that selects clients only using version reward (see Equation 4). To facilitate understanding, we use “GitFL+CV” to indicate the full version of GitFL implemented in Algorithm 1. Figure 8 presents the ablation study results on CIFAR-10 dataset with ResNet-18 following IID distribution. We can observe that “GitFL+CV” achieves the highest inference performance among all four designs and “GitFL+V” outperforms “GitFL+R”, which demonstrates the effectiveness of the version control strategy. Note that the improvement of “GitFL+C” over “GitFL+R” is negligible, since without the model staleness information the curiosity strategy itself cannot benefit FL training.

![Fig. 8. Ablation study for RL-based client selection](image)

### E. Evaluation on Real Test-bed

We implemented GitFL and all four baselines in the real test-bed platform in Figure 9, which includes six Raspberry Pi 4B boards with an ARM quad-core A72 CPU, and a Broadcom VideoCore VI GPU, five Jetson Nano boards with a quad-core ARM A57 CPU, 128-core NVIDIA Maxwell GPU, and 4GB of LPDDR4 RAM, and the cloud server on top of an Ubuntu workstation with an Intel i9 CPU. We conducted experiments on the CIFAR-10 dataset using CNN models with IID scenarios, assuming that there are four devices activated for the purpose of local training at the same time.

Figure 10 presents the learning curves of all five FL methods obtained from our real test-bed platform. We can observe that GitFL can achieve the best inference accuracy compared to all the baselines. For baselines, FedSA and SAFA performed well in inference accuracy, while FedAsync achieved the worst performance. Compared to the counterparts in Figure 7(b), FedAsync and FedAvg in Figure 10 performs worse on the real test-bed, though SAFA can achieve better performance. Note that GitFL and FedSA performs well in both cases.

![Fig. 9. Real test-bed platform for our experiment](image)

![Fig. 10. Learning curves obtained from the real test-bed](image)

### VI. RELATED WORK

To deal with the straggler issue caused by heterogeneous devices, the current FL methods can be classified into three categories, i.e., synchronous FL methods, asynchronous methods, and semi-asynchronous FL methods. For synchronous FL, in [18], Xu et al. proposed Helios to accelerate the training speed of stragglers by adaptive reducing model parameters. Similarly, in [40], Jiang et al. presented a pruning-based FL method PruneFL based on the synchronous scheme to accelerate local training of stragglers. In [16], Wang et al. proposed an experience-driven device selection strategy to counterbalance the bias introduced by non-IID data on each device and to speedup FL training by minimizing the number of communication rounds. Since the synchronous scheme needs to aggregate local models of all the selected devices, synchronous FL methods still waste a lot of time waiting for stragglers. For asynchronous FL, in [20], Xie et al. proposes an asynchronous federated learning framework FedAsync, in which the cloud server aggregates the uploaded model with the global model in real-time according to the staleness of nodes. For semi-asynchronous FL, in [22], Wu et al. proposes a semi-asynchronous FL method SAFA, which

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
uses a cache structure to bypass uploaded models according to their versions for model aggregation. Similarly, in [23], Ma et al. sets a model buffer for model aggregation to achieve semi-asynchronous FL. Although these methods can mitigate the problem of stragglers, due to the aggregation of stale models, their inference capabilities are strongly limited. Moreover, due to a lack of consideration of the uncertainty scenarios, existing methods have to face challenges in adapting to scenarios where device and network performance undergo changes.

To the best of our knowledge, GitFL is the first attempt to use a version control mechanism for asynchronous FL and an RL-based client selection strategy to enable the load balance of versioned models among AIoT devices and adaption to uncertain scenarios. Compared with state-of-the-art FL methods, our GitFL approach can achieve the best test convergence rate and inference performance for both IID and non-IID scenarios.

VII. CONCLUSION

To address notorious straggler issue that results in low inference accuracy and long response time in FL, this paper introduced a novel asynchronous FL framework named GitFL, whose implementation is inspired by the famous version controller system Git. Unlike traditional FL methods, GitFL aggregates the master model based on both the pushed branch models and their version information. Meanwhile, by adopting our proposed RL-based device selection heuristic, GitFL supports adaptive load balance of versioned branch models on devices according to their staleness. Specifically, a pulled model with a newer version will be more likely to be assigned on a less frequently selected straggler. Comprehensive experimental results show that, compared with state-of-the-art asynchronous FL methods, GitFL can achieve better FL performance in terms of both training time and inference performance considering various uncertainties.

ACKNOWLEDGEMENT

This work was supported by the National Research Foundation Singapore and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-2020-019), the Natural Science Foundation of China (62272170), the National Research Foundation, Singapore, the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, and “Digital Silk Road” Shanghai International Joint Lab of Trustworthy Intelligent Software (22510750100). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore. Mingsong Chen is the corresponding author (mschen@sei.ecnu.edu.cn).

REFERENCES

[1] M. Hu, E. Cao, H. Huang, M. Zhang, X. Chen, and M. Chen, “AIoTML: A unified modeling language for aiot-based cyber-physical systems,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.
[2] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, “Communication-efficient learning of deep networks from decentralized data,” in *Proc. of Artificial Intelligence and Statistics*. PMLR, 2017, pp. 1273–1282.
[3] R. Chandrasekaran, K. Ergun, J. Lee, D. Nanjunda, J. Kang, and T. Rosing, “Phdnn: Communication efficient and robust federated learning for aiot networks,” in *Proc. of Design Automation Conference*, 2022, pp. 37–42.
[4] X. Zhang, M. Hu, J. Xia, T. Wei, M. Chen, and S. Hu, “Efficient federated learning for cloud-based aiot applications,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 11, pp. 2211–2223, 2020.
[5] A. Li, L. Zhang, J. Wang, J. Tan, F. Han, Y. Qin, N. M. Freris, and X.-Y. Li, “Efficient federated-learning model debugging,” in *Proc. of IEEE International Conference on Data Engineering*. IEEE, 2021, pp. 372–383.
[6] A. Li, L. Zhang, J. Wang, F. Han, and X.-Y. Li, “Privacy-preserving efficient federated-learning model debugging,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, pp. 2291–2303, 2021.
[7] H. Zhang, J. Bosch, and H. H. Olsson, “End-to-end federated learning for autonomous driving vehicles,” in *Proc. of International Joint Conference on Neural Networks*, 2021, pp. 1–8.
[8] Y. Lu, X. Huang, K. Zhang, S. Maharjan, and Y. Zhang, “Blockchain empowered asynchronous federated learning for secure data sharing in internet of vehicles,” *IEEE Transactions on Vehicular Technology*, vol. 69, no. 4, pp. 4298–4311, 2020.
[9] W. Sun, S. Lei, L. Wang, Z. Liu, and Y. Zhang, “Adaptive federated learning and digital twin for industrial internet of things,” *IEEE Transactions on Industrial Informatics*, vol. 17, no. 8, pp. 5605–5614, 2021.
[10] X. Lin, J. Pan, J. Xu, Y. Chen, and C. Zhuo, “Lithography hotspot detection via heterogeneous federated learning with local adaptation,” in *Proc. of Asia and South Pacific Design Automation Conference*. IEEE, 2022, pp. 166–171.
[11] Z. M. Fadlullah and N. Kato, “On smart iot remote sensing over integrated terrestrial-aerial-space networks: An asynchronous federated learning approach,” *IEEE Network*, vol. 35, no. 5, pp. 129–135, 2021.
[12] Y. Wu, D. Zeng, Z. Wang, Y. Sheng, L. Yang, A. J. James, Y. Shi, and J. Hu, “Federated contrastive learning for dermatological disease diagnosis via on-device learning,” in *Proc. of International Conference On Computer Aided Design*, 2021, pp. 1–7.
[13] S. Baghersalimi, T. Teijeiro, D. Atienza, and A. Aminifar, “Personalized real-time federated learning for epileptic seizure detection,” *IEEE Journal of Biomedical and Health Informatics*, vol. 26, no. 2, pp. 898–909, 2022.
[14] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Ben-
nis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings et al., “Advances and open problems in federated learning,” *Foundations and Trends® in Machine Learning*, vol. 14, no. 1–2, pp. 1–210, 2021.

[15] S. P. Karimireddy, S. Kale, M. Mohri, S. Reddi, S. Stich, and A. T. Suresh, “Scaffold: Stochastic controlled averaging for federated learning,” in *Proc. of International Conference on Machine Learning*, 2020, pp. 5132–5143.

[16] H. Wang, Z. Kaplan, D. Niu, and B. Li, “Optimizing federated learning on non-iid data with reinforcement learning,” in *Proc. of IEEE Conference on Computer Communications*, 2020, pp. 1698–1707.

[17] D. A. E. Acar, Y. Zhao, R. Matas, M. Mattina, P. Whatmough, and V. Saligrama, “Federated learning based on dynamic regularization,” in *Proc. of International Conference on Learning Representations*, 2020.

[18] Z. Xu, F. Yu, J. Xiong, and X. Chen, “Helios: Heterogeneity-aware federated learning with dynamically balanced collaboration,” in *Proc. of Design Automation Conference*. IEEE, 2021, pp. 997–1002.

[19] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, “Federated optimization in heterogeneous networks,” *Proc. of Machine learning and systems*, vol. 2, pp. 429–450, 2020.

[20] C. Xie, S. Koyejo, and I. Gupta, “Asynchronous federated optimization,” *arXiv preprint arXiv:1903.03934*, 2019.

[21] J. Cao, Z. Lian, W. Liu, Z. Zhu, and C. Ji, “Hadfl: heterogeneity-aware decentralized federated learning framework,” in *Proc. of Design Automation Conference*. IEEE, 2021, pp. 1–6.

[22] W. Wu, L. He, W. Lin, R. Mao, C. Maple, and S. Jarvis, “SAFA: A semi-asynchronous protocol for fast federated learning with low overhead,” *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 655–668, 2020.

[23] Q. Ma, Y. Xu, H. Xu, Z. Jiang, L. Huang, and H. Huang, “FedSA: A semi-asynchronous federated learning mechanism in heterogeneous edge computing,” *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 12, pp. 3654–3672, 2021.

[24] C. Xu, Y. Qu, Y. Xiang, and L. Gao, “Asynchronous federated learning on heterogeneous devices: A survey,” *arXiv preprint arXiv:2109.04269*, 2023.

[25] M. Hu, W. Duan, M. Zhang, T. Wei, and M. Chen, “Quantitative timing analysis for cyber-physical systems using uncertainty-aware scenario-based specifications,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4006–4017, 2020.

[26] P. Gupta, Y. Agarwal, L. Dolecek, N. Dutt, R. K. Gupta, R. Kumar, S. Mitra, A. Nicolau, T. S. Rosing, M. B. Srivastava et al., “Underdesigned and opportunistic computing in presence of hardware variability,” *IEEE Transactions on Computer-Aided Design of integrated circuits and systems*, vol. 32, no. 1, pp. 8–23, 2012.

[27] K. Dole, A. Gupta, J. Komp, S. Krishna, and A. Trivedi, “Event-triggered and time-triggered duration calculus for model-free reinforcement learning,” in *Proc. of IEEE Real-Time Systems Symposium*. IEEE, 2021, pp. 240–252.

[28] M. Hu, J. Ding, M. Zhang, F. Mallet, and M. Chen, “Enumeration and deduction driven co-synthesis of ccsl specifications using reinforcement learning,” in *Proc. of IEEE Real-Time Systems Symposium*. IEEE, 2021, pp. 227–239.

[29] D. Liu, S.-G. Yang, Z. He, M. Zhao, and W. Liu, “CARTAD: Compiler-assisted reinforcement learning for thermal-aware task scheduling and dvfs on multicore,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 6, pp. 1813–1826, 2021.

[30] Y. Yang and H. Shen, “Deep reinforcement learning enhanced greedy optimization for online scheduling of batched tasks in cloud hpc systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 3003–3014, 2021.

[31] Q. Liu, T. Xia, L. Cheng, M. Van Eijk, T. Ozcelebi, and Y. Mao, “Deep reinforcement learning for load-balancing aware network control in iot edge systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1491–1502, 2021.

[32] S. K. Moorthy, M. McManus, and Z. Guan, “ESN reinforcement learning for spectrum and flight control in thz-enabled drone networks,” *IEEE/ACM Transactions on Networking*, vol. 30, no. 2, pp. 782–795, 2021.

[33] M. Van Otterlo and M. Wiering, “Reinforcement learning and markov decision processes,” in *Reinforcement learning*, Springer, 2012, pp. 3–42.

[34] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, pp. 279–292, 1992.

[35] H. Hasselt, “Double q-learning,” *Advances in neural information processing systems*, vol. 23, 2010.

[36] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, “Curiosity-driven exploration by self-supervised prediction,” in *Proc. of International conference on machine learning*. PMLR, 2017, pp. 2778–2787.

[37] M. Hu, M. Zhang, F. Mallet, X. Fu, and M. Chen, “Accelerating reinforcement learning-based ccsl specification synthesis using curiosity-driven exploration,” *IEEE Transactions on Computers*, vol. 72, no. 5, pp. 1431–1446, 2023.

[38] M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, “Unifying count-based exploration and intrinsic motivation,” *Advances in neural information processing systems*, vol. 29, 2016.

[39] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečný, H. B. McMahan, V. Smith, and A. Talwalkar, “Leaf: A benchmark for federated settings,” *arXiv preprint arXiv:1812.01097*, 2018.

[40] Y. Jiang, S. Wang, V. Valls, B. J. Ko, W.-H. Lee, K. K. Leung, and L. Tassiulas, “Model pruning enables efficient federated learning on edge devices,” *IEEE Transactions on Neural Networks and Learning Systems*, 2022.