# Reinforcement Learning-Based Network Slice Resource Allocation for Federated Learning Applications

Zhouxiang Wu[1], Genya Ishigaki[2], Riti Gour[3], Congzhou Li[1] and Jason P. Jue[1]
1. Department of Computer Science, The University of Texas at Dallas, Richardson, Texas 75080, USA
2. Department of Computer Science, San Jose State University, San Jose, CA 95192, USA
3. Department of Computer Electronics and Graphic Technology,
Central Connecticut State University, New Britain, Connecticut 06050, USA

***Abstract*—This paper addresses a resource allocation strategy for network slices, where each network slice supports a different federated learning task. A slice is established when a new federated learning model needs to be trained and is released once the training is complete. The goal is to minimize the average network slice holding time while also providing fairness between slice tenants and improving network efficiency. We propose a reinforcement learning-based strategy to periodically reallocate resources according to the current state of each federated learning task. We offer two reinforcement learning models. The first model achieves more stable performance and considers correlations between tasks, while the second model utilizes fewer parameters and is more robust to varying number of tasks. Both approaches have better performance than baseline heuristic methods. We also propose a method to alleviate the effect of various resources scales to make the training stable.**

***Index Terms*—network slice, reinforcement learning, federated learning, resource allocation**

## I. Introduction

In the 5G era, with help of network virtualization, network slicing technologies [1] can be employed to divide the network resources among different users in order to meet each user's customized service requirements. In static network slicing, the amount of resources allocated to each user remains fixed during the service time of the slice. On the other hand, with elastic network slicing, the amount of resources allocated to each user may vary over time. In the elastic scenario, customers may request elastic slices to reduce the cost while still maintaining required service quality. In order to improve resource utilization and to provide fairness across slices, the network operator may dynamically reallocate the resources for the slice based on the state of the tasks that are running over the slice. In this paper, we assume that slice customers are willing to share information regarding the status of tasks that are running over the slice.

As customers emphasize privacy and governments tighten regulations, users are reluctant to share their data. However, data is the fuel to modern machine learning. In [2], the authors propose federated learning (FL), which does not require uploading the data into the cloud. However, local data is usually biased, and the amount of local data is small. Thus, the local trained models are transferred to the cloud and aggregated in some way to avoid over-fitting to the local data.

Federated learning training tasks involve more complicated network resource requirements during their training time compared with traditional machine learning. In traditional machine learning, users upload their data to a corresponding cloud server, where the model is trained on users' data. After training, users download the trained model to solve their local tasks. Traditional training requires users' data to be uploaded and the trained model to be downloaded only once. However, this process may lead to private data leakage since the users' private data is transferred through the network and stored in the cloud. In federated learning, users do not upload their data. Instead, they upload the locally trained model, and the users' models are aggregated in the cloud server. Users download the updated model and train the model again. This process will repeat hundreds or thousands of times until the global model's performance converged. Federated learning has more complex network requirements compared to traditional learning because it involves multiple iterations of downloading and uploading machine learning models of non-negligible size. For example, each federated learning task involves multiple entities, such as participant devices, edge nodes, and the cloud server. The FL task also requires computing and networking resources, such as bandwidth between participants and edge nodes to transfer locally trained models, bandwidth from edge nodes to the cloud to transfer partially aggregated models, and computation resources at edge nodes to partially aggregate locally trained models.

A promising approach for satisfying the complex requirements of FL is to establish a dedicated elastic slice for each FL training task. Once an elastic slice is established, its resources can be dynamically adjusted based on the requirements of the different phases of the FL task. Once the FL training task is complete, the slice is released. Each FL task may be in one of multiple possible states at any specific time, and this state will affect the instantaneous resource requirements of the FL task. For example, if a FL task is currently training local models in participants, then the FL task does not require any bandwidth. If a static network slice strategy is being used, then

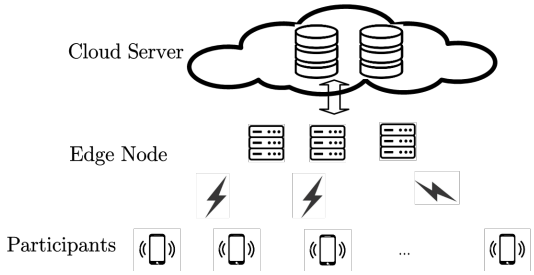


Fig. 1. Experiment Network

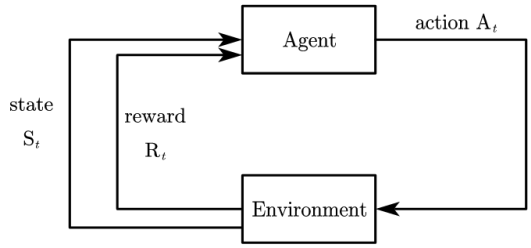


Fig. 2. Reinforcement Learning Procedure

the bandwidth resources may be wasted during this time.

There are challenges in dynamic resource allocation. We cannot allocate resources solely based on the status of tasks since this strategy can lead to tasks stalling. For example, if one task is downloading in the backbone when the allocation decision is made, then the only resource allocated to the slice is the backbone bandwidth. However, the task may finish the downloading in the backbone very soon after the allocation, in which case the task will be stalled until the next allocation since no wireless bandwidth is allocated to the task. Involving predictions on future states may help. However, we need to decide the prediction format and how the allocation strategy involves the prediction feature. On the other hand, reinforcement learning automatically learns an intelligent strategy from history, bypassing handcrafted prediction feature selection. Thus, we propose a reinforcement learning-based network slicing strategy to dynamically allocate resources to the FL tasks in order to utilize the system resources more efficiently.

There are several papers that employ reinforcement learning (RL) to optimize the training process of FL tasks. In [3], the authors design an RL agent that intelligently chooses client devices to participate in each round of federated learning. In [4], the authors minimize the total system cost that is defined as a weighted sum of training time and energy consumption. However, these papers do not consider resource orchestration among different FL tasks. Our algorithm focuses on allocating resources when multiple FL tasks exist in the system.

In this paper, we design a novel model to dynamically allocate slice resources to FL tasks in order to reduce FL training time. However, since our emphasis is on network resource utilization, we focus on the objective of minimizing the average slice holding time. We employ the Deep Deterministic Policy Gradient (DDPG) [5] algorithm to train the agent. The RL agent achieves shorter holding time of the slices and results a more fair resource allocation compared to other heuristic methods. To the best of our knowledge, this paper is the first to apply a reinforcement learning-based elastic slice control strategy to optimize the slices' average holding time. The remainder of this paper is organized as follows. Section II introduces background on reinforcement learning and deep neural networks. We formulate the problem in Section III and illustrate the feature selection, agent design, and the RL algorithm in Section IV. In Section V, we design experiments to prove the effectiveness of the proposed algorithm. Finally, we conclude the paper in Section VI.

## II. Background on Machine Learning Concepts

In Reinforcement learning (RL) [6], there are two entities: the environment and the agent. We show the procedure of RL in Fig. 2. In each step $t$, the agent takes in state $S_t$, which represents the current environment. The agent generates an output action $A_t$ for the environment, and the environment returns an instant reward $R_t$. The environment produces a new state according to the specified action. The objective of the agent is to maximize total rewards. There is a gap between traditional supervised machine learning and reinforcement learning since RL has no ground truth. RL's objective is to train an intelligent agent that can wisely make decisions to achieve long-term profitability. However, the environment is stochastic, and each action's actual reward is untraceable. Thus, we can only train the agent based on its interaction history with the environment. In this paper, the agent corresponds to the network operator, and the environment corresponds to the network condition and state of existing reinforcement learning tasks.

We employ deep neural networks (DNN) as the value function and the actor. A DNN contains multiple fully-connected layers with an activation function between layers. The input of a DNN is a fixed size vector, and the output is a scalar or fixed size vector depending on the design. The DNN for the value function takes in the state of the environment and the action, and outputs a value that represents the expected total reward after this step. The DNN for the actor takes in the state of the environment and outputs an action, which is a vector.

## III. Problem Formulation

### A. Physical Network Infrastructure

The physical network involves federated learning participants, edge computation nodes, and cloud servers. We assume that participants and edge nodes are connected by a wireless network. Edge nodes and cloud servers, are connected by a wired backbone network.

### B. Network Slice

A network slice contains virtual networking resources and computing resources nodes. We consider end-to-end network slices in this paper. The network operator assigns one slice to each FL task, and the slice contains all the resources required for the FL task. We assume that slices are elastic slices, in

which the resources occupied by the slice for the task may vary every pre-defined period of time.

### C. Federated Learning States

Federated Learning tasks have four states: local training, uploading model, downloading model, and global aggregation. When the FL task is in the local training state, the task requires no network resources. In contrast, the task requires corresponding network resources during the uploading model or downloading model phases. When the task is in the global aggregation phase, the task requires cloud computing resources. The four states appear in turn repeatedly until the global model's performance converges. In this paper, we divide the downloading phase into two stages: downloading in the wireless network and downloading in the backbone wired network. The uploading phase is also divided in a similar way. We also add a state for the edge aggregation in edge nodes.

### D. Problem Formulation

We state the control problem of network slice resource allocation as follows. Given (1) a network $G = (V, E)$, with computation capacity $cmp_v$ for $v \in V$ and bandwidth $bw_e$ for $e \in E$, (2) a set of FL tasks, and (3) a decision frequency $f$, the problem is to design an agent that reallocates an elastic slice's resources to each task following $f$. During the period between two decisions, the slice resources remain the same, and the corresponding task proceeds according to the given resources. This problem aims to design an agent that makes a series of decisions to minimize the average network slice holding time.

## IV. Algorithm and Training

### A. Feature Selection

We need to select features that represent the environment and are fed into the RL agent. For each FL task, we use a vector to represent its state. The first element indicates the degree of completion of the task. In our experiments, we select the FL model's test accuracy to represent the degree of completion. The second element indicates the degree of completion of the current stage. For example, if the current state is downloading model from the server, and the downloading process is $80\%$ complete, then the second element is $0.8$. For the third element, we choose the current FL task's model size ratio to the sum of all model sizes. We choose the ratio instead of the actual model size to prevent the gap between features from being too large.

We use one-hot encoding to represent the FL task's current state. For example, we consider six states: downloading in the backbone, downloading in wireless, local training, uploading in wireless, uploading in the backbone, and global aggregation. The sequence of stages is shown in Fig. 3. Thus, we need a $1 \times 6$ vector to represent the states of the FL task. The element corresponding to the current FL task's stage is set to 1, and all other elements are set to 0. For example, if we follow the previously defined sequence, and the current stage is local training, then the one-hot encoding vector is $(0, 0, 1, 0, 0, 0)$.

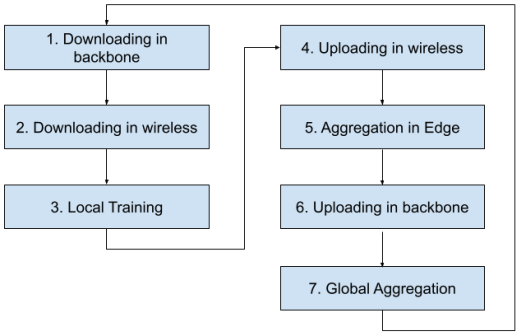


Fig. 3. Federated Learning Stages

Finally, we concatenate the three features and the one-hot encoding vector as a new vector to describe the task. In our case, the new feature of each task should be a $1 \times 9$ vector. If there are three tasks, then the feature should be a matrix whose size is $3 \times 9$.

### B. Agent Design

The agent corresponds to the previously mentioned actor, which takes in the environment states and outputs an action. The input is a matrix, and each row represents a task. However, the input for the typical DNN model is a vector. We propose two types of models to process the input matrix. In the first model, we first flatten the input matrix into a vector and then we feed the vector into the classic DNN model. The output is a vector whose size is $\{1 \times (\text{number of tasks} \times \text{number of resources})\}$. Then, we reshape the vector into a matrix of the required size. The procedure of this model is shown in Fig. 4(a). The DNN model includes multiple fully connected layers with $tanh$ as the activation function between these layers. The model's output should be the resources allocated to the corresponding task. Instead of directly outputting the numerical value of the allocated resources, our model outputs the ratio of the resources allocated to the task to the total resources. This approach alleviates the variance of output value and makes the model more stable during training, allowing the model to transfer its knowledge to different resources scales. Because multiple resources need to be allocated, the model's output is a matrix whose size is the number of tasks multiplied by the number of resources. Because each column of the output sums to 1, we implement the Softmax function on each column, which is defined as follows:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K. \tag{1}$$

The drawback of this design is obvious: the model is tied to the number of tasks. When the number of tasks changes, the state matrix dimension changes accordingly. However, the parameter matrix's size is fixed, so we have to train a new model for each possible number of tasks. Furthermore, flattening the input matrix breaks the information within the

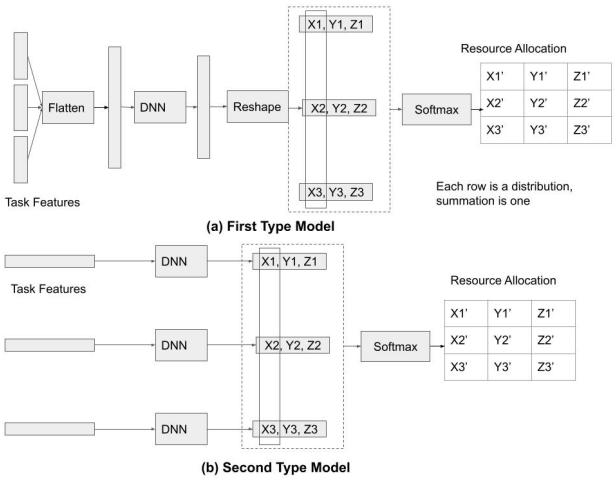


Fig. 4. Two Types of Models

task encoding vector. To avoid this problem, we propose a second type of model, which takes in the row of input and outputs a vector whose size corresponds to the number of resources. Then, we stack all the output vectors as a matrix and implement Softmax on each column. The procedure of the second model is shown in Fig. 4(b). In the second model, every task shares the parameters; thus, the number of parameters does not linearly increase with the number of tasks, and the model is less likely to overfit. Furthermore, the second model can handle any number of tasks, while the first model can only handle a predefined number of tasks. The advantage of the first model is that it can find relationships between tasks, while the second model can only encode the information within the task.

### C. Environment Design

The network operator agent reallocates the resources every fixed period of time. When the environment takes in the action, which is the resource allocation, all of the FL tasks proceed with their assigned resources in the next iteration. After a period of time, the environment outputs the current tasks' states. Since we assume that the network resources do not change during the experiment, we solely utilize tasks conditions to describe the environment state. The task encoding is described in Section IV-A.

### D. Training Algorithm

In our problem, we cannot employ common RL algorithms, e.g. Deep Q Network [7] and Policy Gradient [8], because the action space is continuous. Instead, we select Deep Deterministic Policy Gradient (DDPG) [5], which employs the Deep Q Network algorithm framework and which has the ability to handle a continuous action space. In DDPG, there are two types of models. One model is the previously mentioned agent, which is named as the actor and the other model is another deep neural network named as the critic. The critic's input is a combination of environment state and action, and its output is a number that indicates the expectation of total reward for the resulting state and action, which is named the Q value. Before training the actor and the critic, we need to collect the current actor's trajectories $\{(s_0, a_0, r_0, s_1), (s_1, a_1, r_1, s_2), ..., (s_{n-1}, a_{n-1}, r_{n-1}, s_n)\}$. Suppose the critic gives an accurate Q value $q$ for each state and action pair. The actor tries to maximize the Q value by tuning the actor's parameters through gradient ascent and outputting a more promising action in the next episode. We show this step in Line 13 in Algorithm 1.

However, the Q value is inaccurate because the critic model is initialized randomly. We calculate a more accurate Q value $q'$, which is equal to the summation of the immediate reward $r_t$ and a predicted Q value for the next state and action obtained from the actor given the next state multiply by a discount factor $\gamma$:

$$q'(s_t, a_t) = r_t + \gamma \times q(s_{t+1}, a_{t+1}).$$

Then, we calculate the difference between $q$ and $q'$ and take the derivative of parameters in the critic model to minimize the difference. This step is shown in Algorithm 1 Line 12. To make the training more stable, instead of using the current updated critic model, we use the target critic and the target actor to predict $q'$ and set the target critic equal to the critic and the target actor equal to the actor after several iterations, which is shown in Algorithm 1 Line 19.

The exploration and exploitation tradeoff is an important issue in RL. If we always choose the best action according to the actor, then some better options may not be visited. Thus, during trajectory collection, noise can be added to the actor's model to output a non-greedy action, which is a exploration strategy. This step is shown in Algorithm 1 Line 6. Usually, we fix the noise during one episode to make the training more stable, otherwise the trajectory is not consistent. When testing the model's performance, we use the actor's output action directly, which is a exploitation strategy.

## V. Experiment and Evaluation

In the experiments, we consider the network topology, shown in Fig. 1. Participants connect to an edge node through a wireless network, and the edge nodes connect to the cloud through a backbone network. Each edge node contains computational resources and can aggregate models locally before uploading to the cloud. Thus, in this experiment, there are three types of resources: backbone bandwidth, wireless bandwidth, and edge node computation resources. A slice contains a subset of these resources and is assigned to the corresponding FL task.

### A. Federated Learning Tasks

Since the object of this paper is not to design a state of the art FL algorithm, we chose three well-studied image classification tasks and implemented them in a federated learning scenario. The three tasks are MNIST [9], which classifies images of handwriting digits, Fashion MNIST [10], which classifies images of clothes, and CIFAR-10 [11], which

**Algorithm 1** Slice Allocation Agent Training Algorithm

**Input:**
number of training epoch $T$, number of training sample $L$; critic learning rate $lr_c$, actor learning rate $lr_a$; discount factor $\gamma$ , soft update factor $\tau$; replay buffer $B$, mini-batch size $N$; actor model $A$ and critic model $Q$

**Output:** Trained Actor Model $A$

**Initialize:** Empty replay buffer, randomly initialized actor model $A$ and critic model $Q$
$A^{'} \leftarrow A, Q^{'} \leftarrow Q$
**for** each epoch $t = 0, 1, 2..., T$ **do**
  Initialize tasks state $s_i$
  **while** episode is not finished **do**
    $a_i \leftarrow A.sample(s_i)$
    $s_{i+1}, r_i, f_i \leftarrow env.step(a_i)$ $\{s_{i+1}, r_i, f_i$ indicate the next state, immediate reward, finish flag, respectively.$\}$
    save $(s_i, a_i, r_i, s_{i+1})$ in $B$
    **if** $B$ size $> N$ **then**
      $s_i, a_i, r_i, s_{i+1} f_i \leftarrow$ Sample mini-batch whose size is $N$ from $B$ $\forall i \in \{0, 1, 2, ..., N\}$
      $y_i \leftarrow r_i + \gamma Q'(s_{i+1}, A'(s_{i+1}))$
      Update $Q$ by minimizing the Loss: $L(Q) = \frac{1}{N}\sum_i (Q(s_i, a_i), y_i)^2$
      Update $A$ by maximizing the Q value: $L(A) = \frac{1}{N}\sum_i (Q(s_i, A(s_i)))$
    **end if**
  **end while**
  $A' \leftarrow A, Q' \leftarrow Q$
**end for**
**return** $A$

TABLE I
MODEL SIZE

| Task Name | Model Name | Model Size |
|---|---|---|
| MNIST | VGG 11 | 2475.99 MB |
| FMNIST | VGG 13 | 3628.70 MB |
| CIFAR10 | VGG 19 | 4293.21 MB |

classifies images of ordinary objects. The size of each task's training data is different. The size of MNIST is the smallest, and the size of CIFAR-10 is the largest. The models we choose for FL tasks are from the VGG [12] family. We summarize the model sizes and names in Table I.

### B. Network Structure

In this experiment, we set up 20 participants, three computational edge nodes, and one cloud server. The slice for each task involves backbone bandwidth from cloud to edge, wireless bandwidth from edge to participants, and computational resources on edge nodes.

### C. Reward Function Design and Termination Condition

Because the objective is to minimize the average training time for multiple FL tasks, we simply set the immediate reward for each step to $-1$, which means that the more extended the training phase is, the less reward the agent obtains. When the performance of the FL model converges, the task should terminate. The whole process terminates when every model's performance converges.

### D. Machine Learning Configuration

We implement six fully-connected layers for the actor and the critic, and each layer contains $64$ hidden neurons. The activation function is $tanh$. The initial learning rates for the actor and the critic are both $0.001$. The discount factor $\gamma$ is $0.99$. Instead of updating model parameters every step, we update parameters every $200$ steps. To avoid the lack of data at the beginning of training, we collect $5000$ pieces of data before learning.


Fig. 5. Agent Training Process

### E. Training and Result

During the process, the RL agent samples an action which is a resource allocation strategy for each task, and we save the information in the buffer used to train the RL agent. We show the reward obtained by the RL agent during training in Fig. 5. At the beginning of training, the agent tries to sample a larger action space. Thus, the performance of the agent is not stable. After a hundred training iterations, the agent converges to a stable state.

TABLE II
RESULTS FOR DIFFERENT DECISION PERIOD

| Decision Period | Even Split | Split on Size | Split on State | RL 1 | RL2 |
|---|---|---|---|---|---|
| 1 | 7306 | 6850 | 22450 | 6058 | **5847** |
| 5 | 1462 | 1370 | 2561 | **1205** | 1314 |
| 10 | 731 | 685 | 1091 | 663 | **608** |
| 20 | 366 | 343 | 356 | **309** | 349 |

TABLE III
FIXED MODEL ON DIFFERENT SCENARIOS

| Decision Period | Even Split | Split on Size | Split on State | RL 1 Fixed | RL1 Dedicated |
|---|---|---|---|---|---|
| 1 | 7306 | 6850 | 22450 | **6024** | 6058 |
| 5 | 1462 | 1370 | 2561 | **1205** | 1205 |
| 10 | 731 | 685 | 1091 | **603** | 663 |
| 20 | 366 | 343 | 356 | **302** | 309 |

### *F. Performance*

We compare our algorithm to three standard heuristic methods of allocating resources. The first method is to segment the resources evenly among every existing task. The second method is to split the resources proportional to the FL model size. This approach assumes that larger FL model sizes require more resources for training. The third method is to allocate resources based on the state of tasks. For example, when only two tasks use the backbone link, the backbone bandwidth would be divided evenly between these tasks during the next slot. The first and second methods are static slice control strategies. The units of data appearing in Table II, Table III, and Table IV are unit time. We test different decision periods and summarize the results in Table II. The average training time for the RL agents decreases as the decision frequency increases. However, increasing the frequency adds a burden on network resource management. Thus, there is a trade-off between the frequency and the performance of the model. From the experiments, we cannot conclude which RL agent performs better; however, both RL agents tend to perform better than pre-defined methods. Because the type 1 RL agent has more stable performance compared to the type 2 RL agent, we test its robustness over different decision frequencies and summarize its results in Table III. We employ the trained model under a decision period of ten and apply the model to other decision frequency scenarios. The model is stable in every scenario and has even better performance than the dedicated model trained to the specific model.

Another interesting observation is that the RL agents achieve more fair resource allocation. We show the training times and the standard deviation of training times for different tasks under different strategies in Table IV. The results show a trade-off between efficiency and fairness. We suppose that the system should allocate resources evenly to homogeneous tasks. In our experiments, the three tasks have the same level of priority. Thus, the standard deviation of their training time should be as low as possible. As Table IV shows, the RL agents have a minor standard deviation without loss of performance.

TABLE IV
STANDARD DEVIATION AMONG DIFFERENT TASKS' TRAINING TIME UNDER DIFFERENT STRATEGY CONTROL

| | Task 1 | Task 2 | Task 3 | Total | Std Deviation |
|---|---|---|---|---|---|
| Even Split | 269 | 857 | 1462 | 1462 | 596.52019 |
| Split on Size | 306 | 843 | 1370 | 1370 | 532.00783 |
| Split on State | 250 | 1998 | 2561 | 2561 | 1205.0722 |
| RL 1 | 463 | 1186 | 1205 | 1205 | 423.01576 |
| RL 2 | 391 | 1063 | 1314 | 1314 | 477.23404 |

## VI. CONCLUSION

We apply a reinforcement learning-based elastic network slice strategy to federated learning tasks and design two RL agents. Both RL agents achieve better performance in the simulation experiments than heuristic methods in general. We revised the DDPG algorithm to fit our problem setup and uniform the input and output to same scale in order to have a numerical stable performance.

The FL tasks are not constrained to the image classification. We can implement the similar framework to network problems such as federated multi-domain anomaly detection. For example, if each domain does not wish to upload its data but wants to train a anomaly detection model together, then these domains can employ this framework to fulfill the requirement.

## VII. ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under Grant No. CNS-2008856.

## REFERENCES

[1] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, "Network slicing and softwarization: A survey on principles, enabling technologies, and solutions," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2429–2453, 2018.

[2] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

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

[4] Y. Zhan, P. Li, and S. Guo, "Experience-driven computational resource allocation of federated learning by deep reinforcement learning," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 234–243, IEEE, 2020.

[5] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[6] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[7] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, 2016.

[8] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in neural information processing systems*, vol. 12, 1999.

[9] "The mnist database of handwritten digits." http://yann.lecun.com/exdb/mnist/. Accessed: 2020-03-19.

[10] "Fashion mnist dataset, an alternative to mnist." https://keras.io/api/datasets/fashion_mnist/. Accessed: 2020-03-19.

[11] "The cifar-10 dataset." https://www.cs.toronto.edu/~kriz/cifar.html. Accessed: 2020-03-19.

[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.