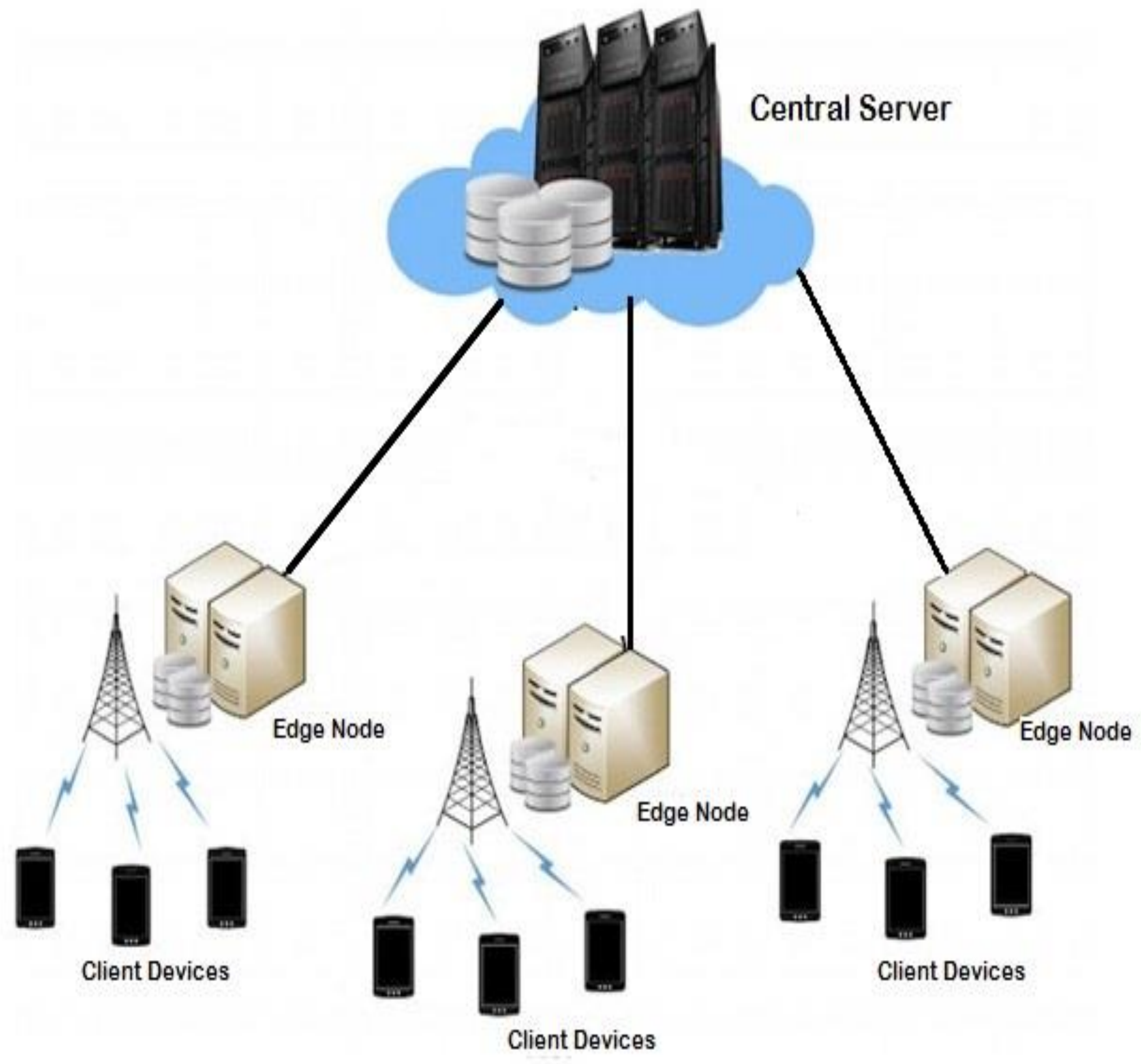
Optimization of Node Selection in Mobile Edge Networks through Federated Deep Reinforcement Learning

Outline

- Introduction
    - Mobile Edge Computing
    - Federated Learning
- Contributions
- Deep Reinforcement Learning for Node Selection
    - Agent Based on Deep Q-Network
    - DDQN Agent interaction with FL Server
    - Workflow at the Central Server
    - Workflow at the Edge Node
- Implementation and Evaluation

Mobile Edge Computing (MEC)

- Mobile Edge computing (MEC) aka Multi-access Edge Computing:
- serve the rapidly growing number of mobile and IoT devices at the network edge, for faster response time and better network quality.
- Typical distributed MEC network, consisting of
- IoT/Mobile devices: resource-constrained end-user devices and connect with the edge nodes for network services.
- Edge nodes: such as home gateways, micro servers, base stations are distributed across geographical regions
    - collect and store raw data generated by the devices connected to them
- Remote central server: has unlimited scalable resources available, and edge nodes are connected to the remote central server via backbone links.



Federated Learning

- Federated learning (FL) trains a shared global model by iteratively aggregating model updates from multiple client devices, which may have slow and unstable network connections.
- In each round, the server randomly selects a subset of available client devices to participate in training.
- The selected devices first download the latest global model from the server, train the model on their local datasets, and report their respective model updates to the server for aggregation.
- We formally introduce FL in the context of a *C*-class classification problem:
- which is defined over a compact feature space *X*, and
- label space *Y* = [C], where [C] = 1,…,C.
- (x,y) denotes a particular labeled sample.
- Let *f* : X -> S denote prediction function, where

$$\mathcal{S} = \{\boldsymbol{z} | \sum_{i=1}^{C} z_i = 1, z_i \geqslant 0, \forall i \in [C]\}.$$

- That is, the vector valued function *f* yields a probability vector *z* for each sample *x*, where f_j predicts the probability that the sample belongs to the *ith* class.

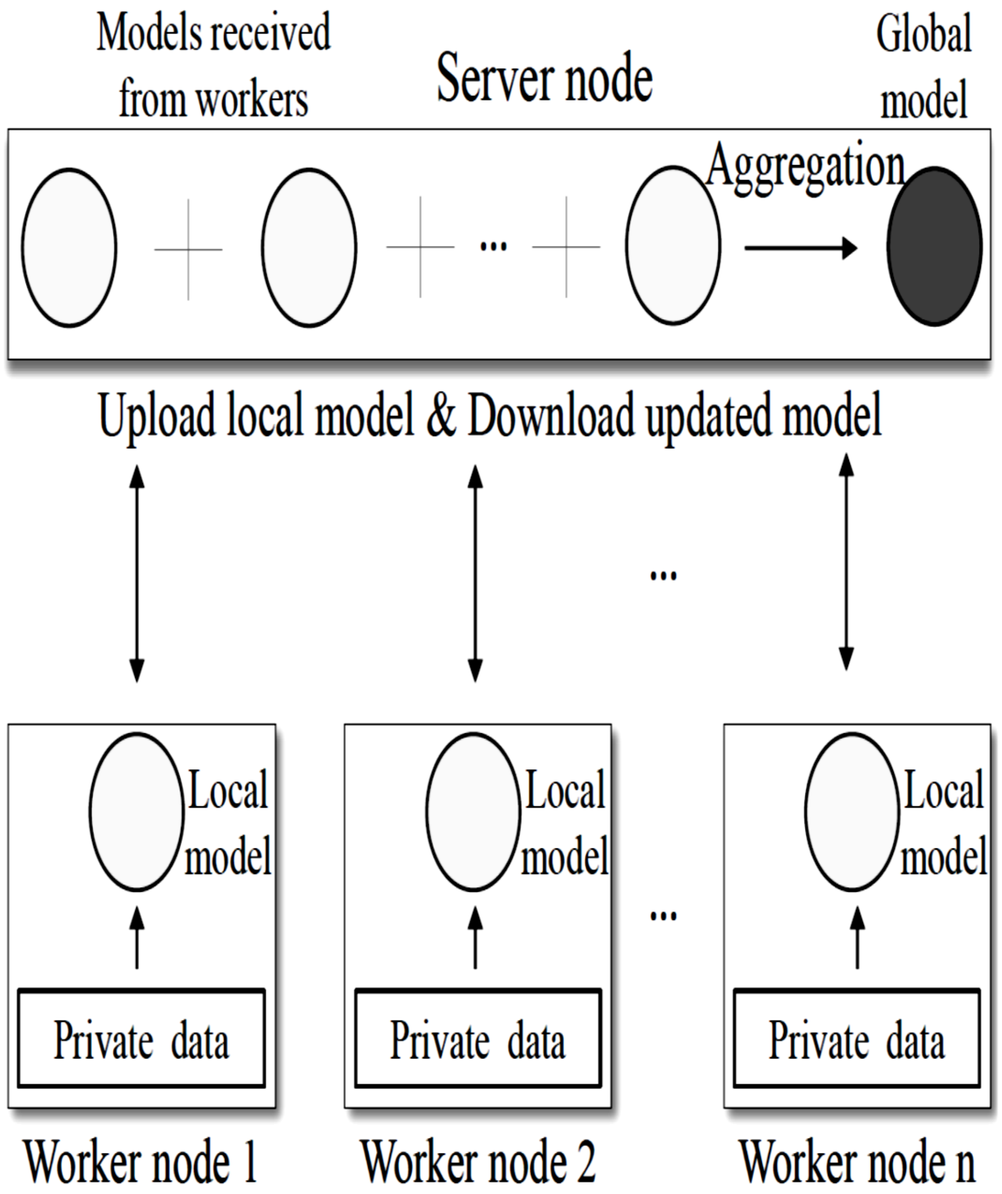


Fig. 1: An overview of federated learning

Let w, model weights. For classification, the common training loss is cross entropy, defined	$\ell(\boldsymbol{w}) := \mathbb{E}_{\boldsymbol{x},y\sim p}\left[\sum_{i=1}^{C} \mathbb{1}_{y=i} \log f_i(\boldsymbol{x},\boldsymbol{w})\right] = \sum_{i=1}^{C} p(y=i)\mathbb{E}_{\boldsymbol{x}\|y=i}[\log f_i(\boldsymbol{x},\boldsymbol{w})]$
The learning problem is to solve the following optimization problem:	$\textbf{minimize}_{\boldsymbol{w}} \quad \sum_{i=1}^{C} p(y=i)\mathbb{E}_{\boldsymbol{x}\|y=i}[\log f_i(\boldsymbol{x},\boldsymbol{w})].$
In FL, *N* clients, the *k*th device has $m^{(k)}$ data samples following $p^{(k)}$, In each round *t*, *K* devices are selected (at random in the original FL), each downloads current global model weights w_{t-1} from server/perform SGD locally:	$\boldsymbol{w}_t^{(k)} = \boldsymbol{w}_{t-1} - \eta\nabla\ell(\boldsymbol{w}_{t-1}) = \boldsymbol{w}_{t-1} - \eta\sum_{i=1}^{C} p^{(k)}(y=i)\nabla_{\boldsymbol{w}}\mathbb{E}_{\boldsymbol{x}\|y=i}[\log f_i(\boldsymbol{x},\boldsymbol{w}_{t-1})],$
Once $\mathbf{w}_t^{(k)}$ is obtained, each device *k* reports model weight difference to the FL server,	$\Delta_t^{(k)} := \boldsymbol{w}_t^{(k)} - \boldsymbol{w}_{t-1}^{(k)}.$
After collecting updates from all devices, FL performs **FEDAVG**, to update the global model	$\Delta_t = \sum_{k=1}^{K} m^{(k)}\Delta_t^{(k)} / \sum_{k=1}^{K} m^{(k)}, \quad \boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} + \Delta_t.$

Contributions

- This work is based on a previous research paper published in 2020.
- The original work uses a two -tier structure for the federated mobile edge network,: client devices directly connect with the central server.
- This scheme can suffer from:
  - high communication latency for reasons like, higher distance between client devices and the central server, client devices having slow and/or unreliable network connection etc.
  - Scalability: especially when enormous number of client devices want to connect directly to the central server, or the client devices are far away from the central server.
  - Network Congestion: all client devices connecting to the single central server
- Communication latency not considered for selection of the client devices in each round to participate in the Federated learning training

- In this work we consider a three-tier mobile edge network, with edge nodes as the middle layer between the central server and the client devices.
- The client devices can connect to the closest edge node. This can help in: both reducing the communication latency and increasing scalability.
- Reducing communication latency, as the edge nodes are closer to the client devices as compared to the central server, which results in:
  - faster response time, communication and reduction in resource consumption of the client devices.
- Multiple edge nodes allow for a larger number of client devices to be connected to the network, and result in more scalable network.
  - The workload is also distributed to the multiple edge nodes and the central server, which can result in better overall performance.
  - Avoid Network congestion
- In this work,
  - introduce a penalty for the communication latency: penalize the slow devices or the devices with unreliable network connection.
  - Introduce staleness threshold, the model update received after this threshold will not be considered for model aggregation
- In this work, we introduce the utility of edge nodes or the contribution they make to the global model update.
  - the central server allocates k for each of the edge nodes based on the contribution of the edge node to the global model.

DRL for Node Selection : Agent Based Deep Q-Network

- Suppose there is a federated learning job on *N* available devices with a target accuracy Ω. In each round, using a DQN, *K* devices are selected to participate in training.
- Considering limited available traces from federated learning jobs, DQN can be more efficiently trained and can reuse data more effectively than policy gradient methods and actor-critic methods.
- **State:** Let the state of round t be represented by a vector

$$\boldsymbol{s}_t = (\boldsymbol{w}_t, \boldsymbol{w}_t^{(1)}, \ldots, \boldsymbol{w}_t^{(N)}).$$

- **Action:** At beginning of each round t, the agent needs to decide to select which subset of K devices from the N devices.
- **Reward**: We set the reward observed at the end of each round t to be

$$r_t = \Xi^{\{(\omega_t - \Omega - \phi)\}} - 1, t = 1, \ldots, T$$

The DQN agent is trained to maximize the expectation of the cumulative discounted reward given by

$$R = \sum_{t=1}^{T} \gamma^{t-1} r_t = \sum_{t=1}^{T} \gamma^{t-1} \left(\Xi^{\{(\omega_t - \Omega) - \phi\}} - 1 \right)$$

Continued…

- RL is the learning process of an agent that acts in corresponding to the environment to maximize its rewards.
- At each time step t:
- the RL agent observes states s_t and performs an action a_t.
- The state s_t of the environment then transits to s_{t+1}, and
- the agent will receive reward r_t .
- The state transitions and rewards follow a (MDP), which is a discrete time stochastic control process, denoted by a sequence $(s_1, a_1, r_1, s_2, \ldots, r_{t-1}, s_t, a_t, \ldots)$.
- Training the agent with Double DQN

$$\begin{aligned} Q_\pi(\boldsymbol{s}_t, a) : &= \mathbb{E}_\pi[\sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k-1} | \boldsymbol{s}_t, a] \\ &= \mathbb{E}_{\boldsymbol{s}_{t+1}, a}[r_t + \gamma Q_\pi(\boldsymbol{s}_{t+1}, a) | \boldsymbol{s}_t, a_t], \end{aligned}$$

$$Q^*(s_t, a) := \mathbf{E}_{\mathbf{s}_{t+1}} \left[r_t + \gamma \max_a Q^*(s_{t+1}, a) \middle| s_t, a \right]$$

$$\ell_t(\theta_t) := \left(r_t + \gamma \max_a \mathbf{Q}(s_{t+1}, a; \theta_t) - Q(s_t, a; \theta_t) \right)^2$$

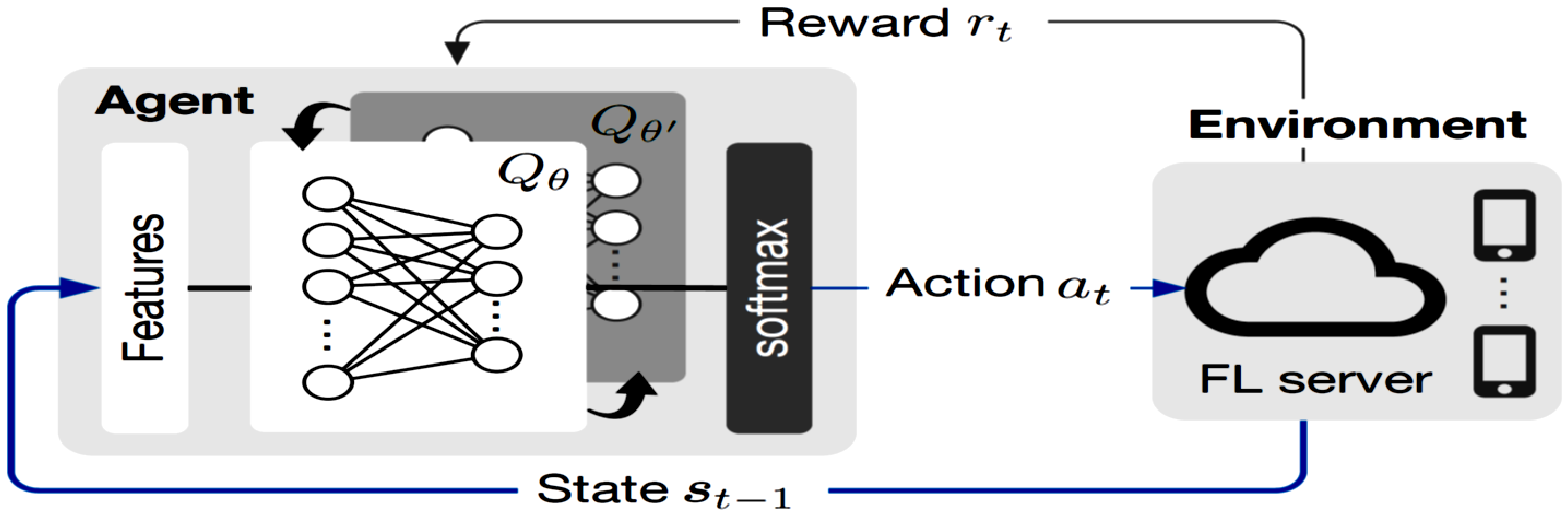
$$\ell_t(\theta_t) = \left(Y_t^{DoubleQ} - Q(s_t, a; \theta_t) \right)^2$$

$$\begin{aligned} Y_t^{DoubleQ} &:= r_t + \gamma \max_a Q(s_{t+1}, a; \theta_t) \\ &= r_t + \gamma Q\left(\mathbf{s}_t, \operatorname*{argmax}_a Q(s_t, a; \theta_t); \theta_t' \right) \end{aligned}$$

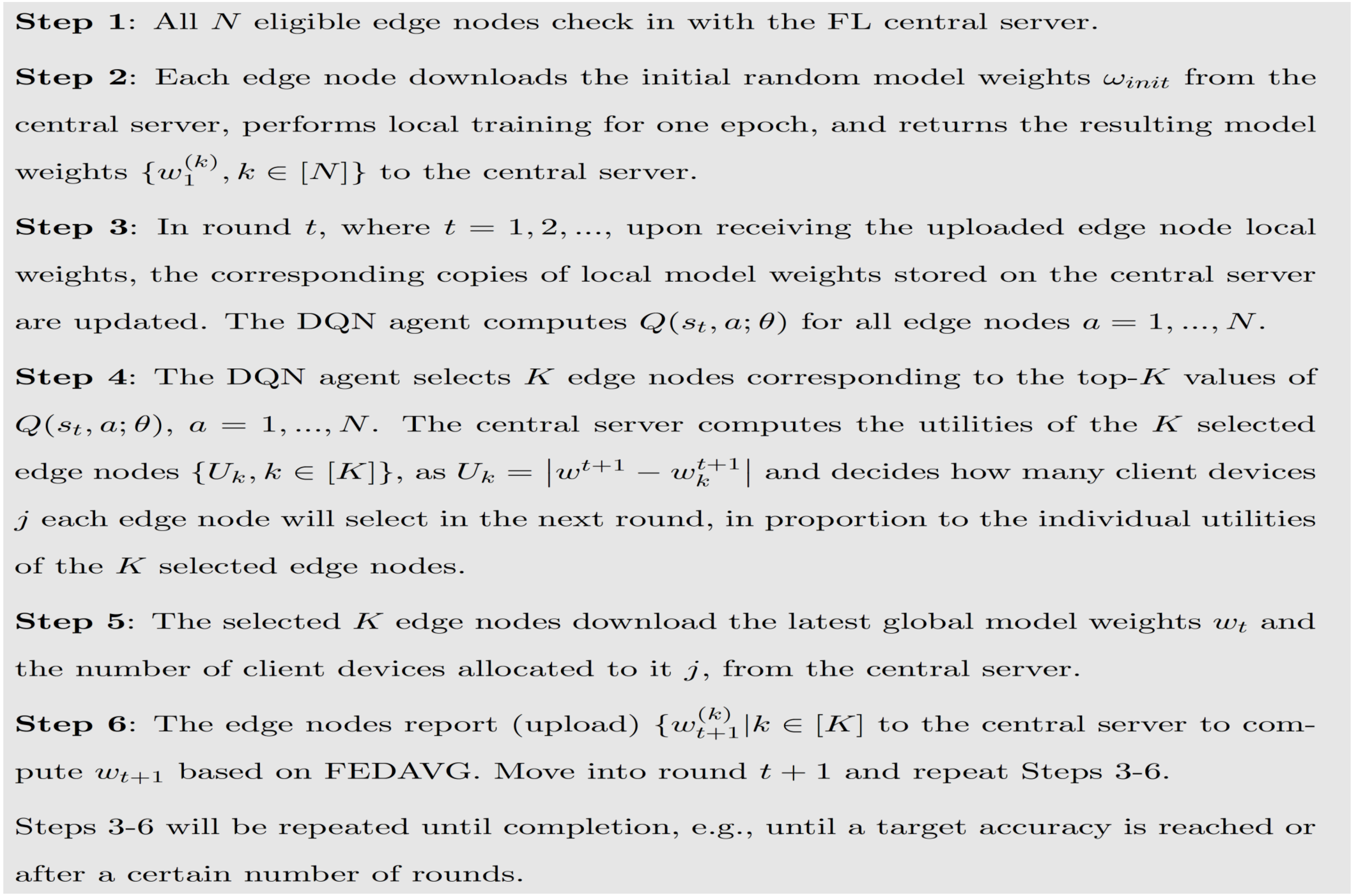
$$\theta_{t+1} = \theta_t + \alpha \left(Y_t^{DoubleQ} - Q(s_t, a; \theta_t) \right) \nabla_{\theta_t} Q(s_t, a; \theta_t)$$

DDQN Agent Interaction with FL Server

- To train the DRL agent, the FL server first performs random device selection to initialize the states
- Figure: the states are fed into one of the double DQNs $Q(s_t, a; \theta_t)$.
- The DQN generates an action *a* to select a device for the FL server.
- After several rounds of FL training, DRL agent has sampled a few action-state pairs, with which the agent learns to minimize the loss.
- Double DQN uses two action-value functions to update, in which θ_t is the online parameters updated per time step, and θ'_t is the frozen parameters to add stability to action-value estimation.



Workflow at Central Server



Step 1: All N eligible edge nodes check in with the FL central server.

Step 2: Each edge node downloads the initial random model weights ω_{init} from the central server, performs local training for one epoch, and returns the resulting model weights $\{w_1^{(k)}, k \in [N]\}$ to the central server.

Step 3: In round t, where $t = 1, 2, ...,$ upon receiving the uploaded edge node local weights, the corresponding copies of local model weights stored on the central server are updated. The DQN agent computes $Q(s_t, a; \theta)$ for all edge nodes $a = 1, ..., N$.

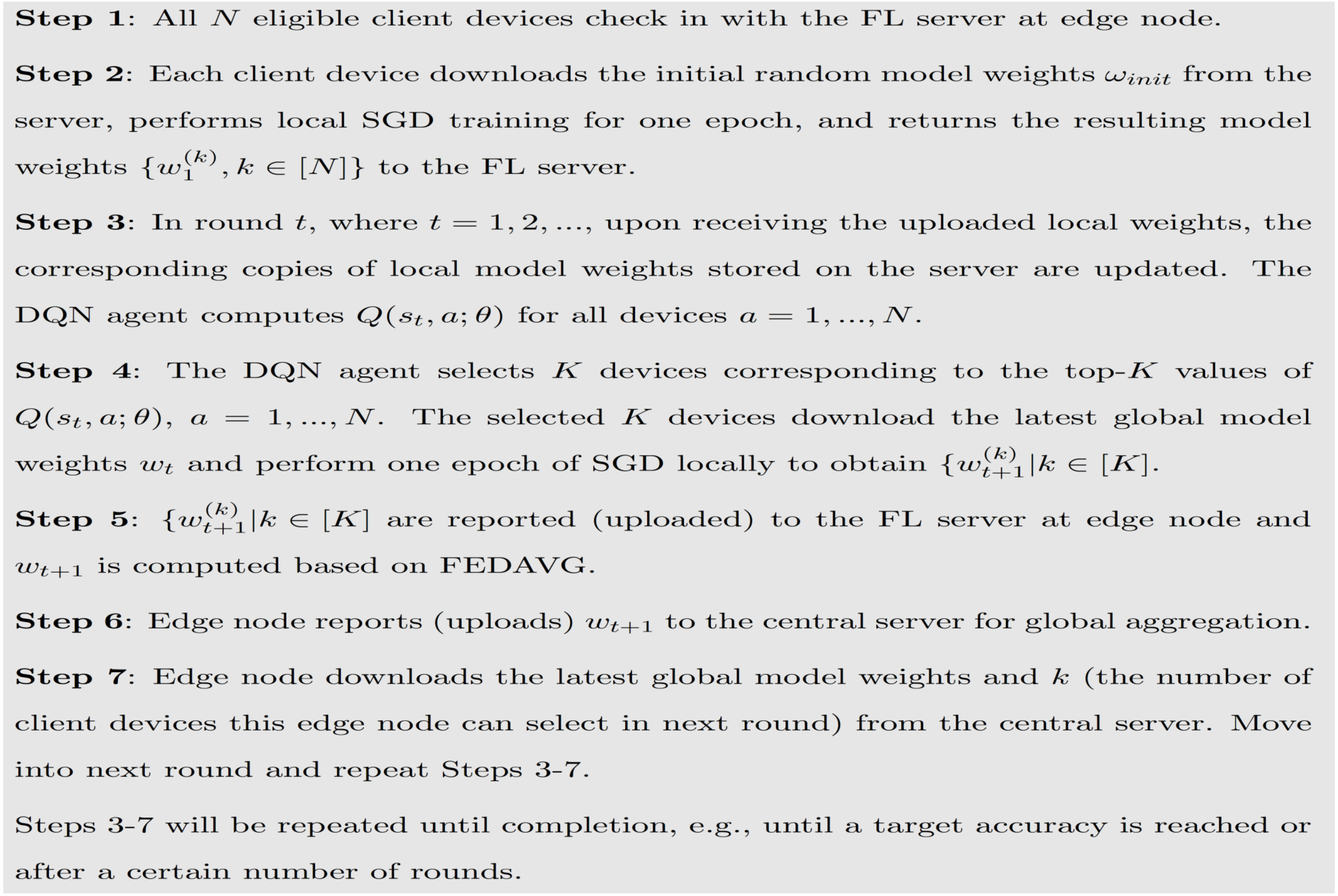
Step 4: The DQN agent selects K edge nodes corresponding to the top-K values of $Q(s_t, a; \theta)$, $a = 1, ..., N$. The central server computes the utilities of the K selected edge nodes $\{U_k, k \in [K]\}$, as $U_k = \left|w^{t+1} - w_k^{t+1}\right|$ and decides how many client devices j each edge node will select in the next round, in proportion to the individual utilities of the K selected edge nodes.

Step 5: The selected K edge nodes download the latest global model weights w_t and the number of client devices allocated to it j, from the central server.

Step 6: The edge nodes report (upload) $\{w_{t+1}^{(k)} | k \in [K]$ to the central server to compute w_{t+1} based on FEDAVG. Move into round $t + 1$ and repeat Steps 3-6.

Steps 3-6 will be repeated until completion, e.g., until a target accuracy is reached or after a certain number of rounds.

Workflow at Edge Node



Step 1: All N eligible client devices check in with the FL server at edge node.

Step 2: Each client device downloads the initial random model weights ω_{init} from the server, performs local SGD training for one epoch, and returns the resulting model weights $\{w_1^{(k)}, k \in [N]\}$ to the FL server.

Step 3: In round t, where $t = 1, 2, ...,$ upon receiving the uploaded local weights, the corresponding copies of local model weights stored on the server are updated. The DQN agent computes $Q(s_t, a; \theta)$ for all devices $a = 1, ..., N$.

Step 4: The DQN agent selects K devices corresponding to the top-K values of $Q(s_t, a; \theta)$, $a = 1, ..., N$. The selected K devices download the latest global model weights w_t and perform one epoch of SGD locally to obtain $\{w_{t+1}^{(k)}|k \in [K]$.

Step 5: $\{w_{t+1}^{(k)}|k \in [K]$ are reported (uploaded) to the FL server at edge node and w_{t+1} is computed based on FEDAVG.

Step 6: Edge node reports (uploads) w_{t+1} to the central server for global aggregation.

Step 7: Edge node downloads the latest global model weights and k (the number of client devices this edge node can select in next round) from the central server. Move into next round and repeat Steps 3-7.

Steps 3-7 will be repeated until completion, e.g., until a target accuracy is reached or after a certain number of rounds.

Implementation and Evaluation

- PyTorch, training popular CNN models on three benchmark datasets:
  - MNIST, FashionMNIST, and CIFAR-10,
  - with FEDAVG and K-Center as the groups of comparison.
- We evaluate the accuracy of the trained models using the testing set from each dataset.
- We briefly describe our methodology and settings as follows.
- **Datasets and models**: different combinations of hyper-parameters are explored for CNN models on different datasets and following hyper-parameters chosen leading to best performance of FEDAVG.
  - **MNIST**. We train a CNN model with two 5x5 convolution layers. The first layer has 20 output channels and the second has 50, with each layer followed by 2x2 max pooling. On each device, the batch size is 10 and the number of epoch is 5.
  - **FashionMNIST**. We train a CNN model with two 5x5 convolution layers. The first layer has 16 output channels and the second has 32, with each layer followed by 2x2 max pooling. On each device, the batch size is 100 and the number of epoch is 5.
  - **CIFAR-10**. We train a CNN model with two 5x5 convolution layers. The first layer has six output channels and the second having 16, with each layer followed by 2x2 max pooling. On each device, the batch size is 50 and the number of epoch is 5.
- **Performance metrics**: In federated learning, due to the limited computation capacity and network bandwidth of mobile devices, reducing the number of communication rounds is crucially important. Thus, we use the number of communication rounds as the performance metric.

Training the DRL Agent

- 100 total devices.
- The DDQN model in the DRL agent consists of two two-layer MLP networks, with 512 hidden states.
- The input size is 10,100, (100+1 model weights)
- The output size of the second layer is 100.
- Each output passing through a softmax layer becomes the probability of selecting the particular device.
- An episode starts at the initialization of a FL job and ends when the job converges to the target accuracy Ω.
- The total return is the cumulative discounted reward *R* obtained in one episode.
- The target accuracy Ω is set to:
  - 99% for training on the MNIST,
  - 85% for training on FashionMNIST, and
  - 55% for training on CIFAR-10

Different Levels of Non-IID Data

σ = 0 IID **K = 10**

σ = 1.0 data on each device only belong to one label

σ = 0.8 80% data belong to one label, 20% other labels

σ = 0.5 50% data belongs to one label, 50% other labels

σ = H data evenly belong to Two labels

Increasing the Parallelism

- Varying numbers of selected devices.
- The number of selected devices *K* is set to :
- K = 10,
- K = 50, and
- K = 100

to study the performance of federated learning with different parallelism.