BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

Project Report on
Deep Learning Application on Resource - Constrained UAVs for IoT use-cases

Instructor in Charge: Dr. GSS Chalapathi

Name: Jagrit Lodha & Satyajit Ghosh

ID: 2019A3PS0165P & 2018B5A80931P

DECLARATION BY THE STUDENTS

We declare that the project report entitled “Deep Learning application on Resource-constrained UAVs for IoT use-cases ” submitted by us to fulfill the course requirements to the Department of Electrical & Electronics, BITS Pilani, Pilani Campus is entirely our own work done under the guidance of Prof GSS Chalapathi and Mr. Sunil in the first semester of session 2021-22. The entire work is plagiarism free and all reference material is duly acknowledged in the report.

Jagrit Lodha	Satyajit Ghosh
2019A3PS0165P	2018B5A80931P

CERTIFICATE BY THE SUPERVISOR

The project report entitled “Deep Learning application on Resource-constrained UAVs for IoT use-cases” submitted by Jagrit Lodha and Satyajit Ghosh is a work done under my supervision in the first semester of the session 2021-22. I recommend this work to be accepted by the department for completion of course requirements. To the best of my knowledge the entire work is plagiarism free and all reference material is duly acknowledged in the report.

Prof GSS Chalapathi

Instructor In-charge

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to our instructor-in-charge Prof GSS Chalapathi who gave us the opportunity to work on the topic “Deep Learning application on Resource-constrained UAVs for IoT use-cases”. We are extremely grateful to our mentor Mr. Sunil who guided us along the way of completion of this project and the moral support he gave us in the entire process of completing this study project. The completion of this report wouldn’t have been possible without the support of our family and friends and their understanding and support in completing this project. Without the help of those mentioned above, we would have faced many difficulties while doing this.

Introduction

There is a growing need for solutions of training and doing inference on resource constrained devices since the recent growth of interest in UAVs which usually lack the compute intensive GPUs to carry out training. UAVs are proposed as a new innovative and cost-effective solution for low altitude sensing with zero deployments however, they don’t have enough resources on it’s own to compute the complex deep learning algorithms.

Computational costs of Deep learning Neural Networks are increasing every year and thus there is a need for resource efficient deep learning strategies.

Literature Review

Distributed CNN inference on Resource-Constrained UAVs for Surveillance Systems: Design and Optimization, 2021

- In this paper, a DNN distribution methodology within UAVs is proposed to enable data classification in resource-constrained devices and avoid extra delays introduced by the server-based solutions due to data communication over air-to-ground links.
- The basic idea is to divide the DNN model into segments (e.g. layers, multiplication tasks, etc.) and each segment is allocated within a participant. Each participant shares the output to the next one until generating the final prediction. In this way, the entire inference can be locally performed at the proximity of the data source, without the need to transmit the original data to the remote servers.
- The proposed method is formulated as an optimization problem that aims to minimize the latency between data collection and decision-making while considering the mobility model.
- Pruning is one method to remove the redundant weights of model to make the architecture smaller and thereby run it on a smaller processing unit. However, not possible with all such architectures.

Partitioning of architectures can happen in the following ways -

1) compute shallower layers on-board and the deeper layers on cloud.

2) hierarchical architecture - model is distributed among cloud, edge and mobile units.

3) rely on pervasive devices to execute the inference requests at the proximity of devices: either adopt-per-layer distribution or to apply per-input partitioning

This paper only discusses the cases of U2U(UAV to UAV)

- Objective function models the end-to-end latency spent to classify all incoming requests to the network composed of N UAVs using collaborative inference.
- End-to-end latency defined as time between gathering images and the decision-making is optimized.
- Objective function in optimization problems is non linear and non convex and such problems have multiple local optimal points of solutions which are very complex. Therefore, convexity and linearity is maintained to arrive at an easier solution.

This converts the problem into a linear problem due to the decision variable that has been introduced.

LeNet and VGG were used in simulation to study the tradeoff between the number of requests that could be processed in parallel, the number of UAVs available to perform collaborative inference as well as their computation capabilities. The impact of the wireless communication between the UAVs on the end-to-end latency through a data rate model including the interferences that occur in the network was also studied.

High-Throughput CNN Inference on Embedded ARM big.LITTLE Multi-Core Processors, 2020

- ARM-CL is highly optimized for edge-specific ARM core architectures with inbuilt support for multi-threading and acceleration through ARM NEON vectorization.
- Single-ISA heterogeneous multi-cores comprise of processing cores that have different power-performance-area characteristics but share the same Instruction Set Architecture (ISA)
- CPU remains the platform of choice for running ML workloads being the most common denominator with high availability in mobile and embedded

platforms.

- Heterogeneous Multi-Processing (HMP) allows execution of kernels using both Big and Small Cores simultaneously.
- Framework called Pipe-it proposed that solves the issue of training on heterogeneous multi-core processors.

Why Pipe-it?

- Throughput obtained by splitting the computational workload from the kernel equally among all threads. However, distributing workload disproportionately does not improve throughput significantly either.
- No ratio of workload split between Big and Small clusters results in statistically significant higher throughput for most CNNs than when kernels run exclusively only on Big cluster.

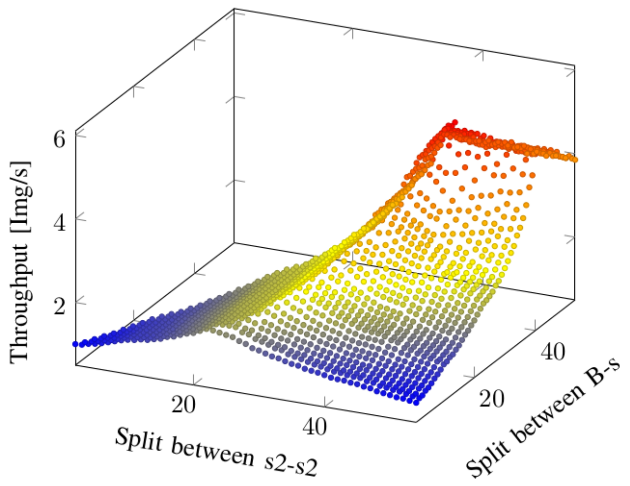
Pipe-it partitions CNN layers across heterogeneous cores to improve throughput. Pipe-it creates a processing pipeline by splitting layers among heterogeneous core clusters, wherein a given set of homogeneous core(s) always process kernels from a fixed set of layers. Different pipeline stages are responsible for concurrently processing different layers corresponding to consecutive images in a stream.

For a basic two-stage layer-level split pipeline processing a network containing W major layers. First X layers are processed on Big cluster with Kernel-level split among all four Big cores and rest (W −X) layers are processed on Small cluster. The challenge is to find an optimal split point X with maximum throughput.

Only pipeline configurations with Big cores for initial convolutional layers and Small cores for subsequent convolutional layers are considered as CNNs usually have more compute-intensive convolutional kernels at the beginning

Pipe-it first predicts the execution time of all layers on all possible core configuration from static network-layer configuration descriptors.

Pipe-it then goes through design space heuristically using predicted timing information to obtain near-optimal pipeline configuration and corresponding workload allocation. This heuristic map can be seen in the following image.



- Power efficiency of individual components not studied due to lack of sensors.
- Pipe-it is orthogonal to Quantization and this framework works better.
- A layer-level splitting Pipe-it efficiently uses entire heterogeneous multi-cores to improve CNN inference throughput. A search algorithm was designed to locate a high performing design point within it.
- Pipe-it improves the throughput on average by 39% using all heterogeneous cores in comparison to using only homogeneous cores.

UAVs as a Service: Boosting Edge Intelligence for Air-Ground Integrated Networks, 2020

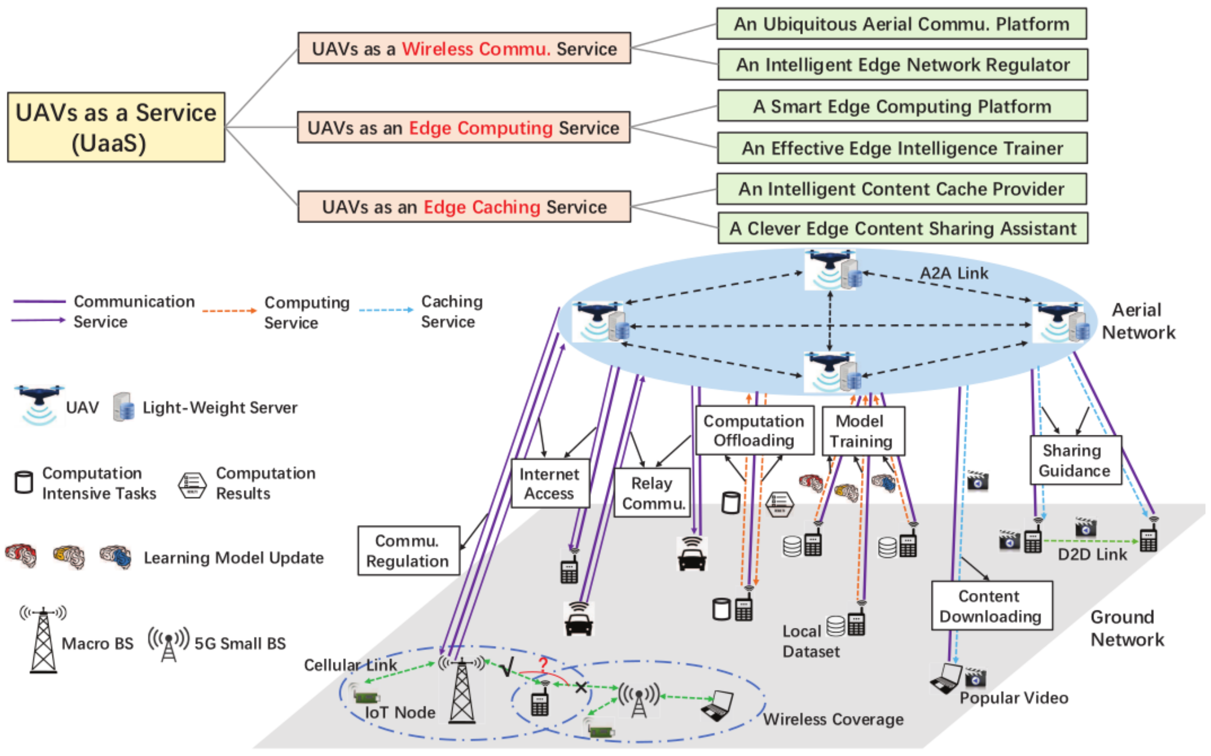
There is a need to intelligently provision various services in 6G networks which is challenging. To meet this need authors propose a novel architecture UaaS (UAVs as a Service) for air-ground network, featuring UAV as a key enabler to boost edge intelligence using machine learning (ML) techniques.

The proposed UaaS architecture could intelligently provision wireless communication service, edge computing service, and edge caching service by a network of UAVs.

A case study was also conducted where UAVs participate in the model training of distributed ML among multiple terrestrial users, whose result shows that the model training is efficient with a negligible energy consumption of UAVs, compared to the flight energy consumption.

From a high-level view, solutions in the UaaS to address the challenges are as follows.

- To satisfy the temporal-spatial dynamic service demand, we provision on-demand services by dispatching UAVs according to the prediction by ML techniques and big data analytics.
- To combat large-scale complex connection decision and resource management, we manage connections and resources by deep reinforcement learning (DRL) benefited from the wide coverage of UAVs.
- To meet the ubiquitous intelligence demand inside the network, we endow terrestrial users with ubiquitous powerful computation ability by UAVs, and enable UAVs to participate in the model training of distributed ML with the flexible deployment of UAVs.



Accelerating DNN Training in Wireless Federated Edge Learning Systems, 2021

This paper makes use of FEEL (federated edge learning) to accelerate the training task of general deep neural networks

- The prior works on FEEL mainly focus on accelerating the training task from the communication perspective, i.e., reducing the communication overhead between centralized node and user devices.
- The major hyperparameter, i.e., the training batchsize is also optimized, which is of paramount importance to the learning performance improvement.

- The main result of this paper shows that the training batchsize should dynamically adapt to the wireless channel condition to achieve the most desirable learning performance.

Offloading Optimization in Edge Computing for Deep Learning Enabled Target Tracking by Internet-of-UAVs

- Due to large computation requirements of UAV applications (Image segmentation, Vehicle tracking, etc. which mostly require ML/DL), and limited computational capabilities of the UAVs, it is imperative to study various offloading or optimization algorithms in order to improve the process.
- The paper aims to optimize a deep learning based target tracking system using UAVs (Unmanned Aerial Vehicles) and MES (Mobile Edge Computing Servers).
- Due to less computational resources on the UAVs, we need to consider offloading tasks to the MES. This can improve the accuracy but at the same time introduce a delay.
- The final goal is to minimize the weighted-sum cost of the system, with the constraints being inference accuracy and time delay.

<u>The Proposed HMTD Framework</u> -

- HMTD stands for Hierarchical Machine Learning Tasks Distribution.
- Salient Features of the Framework -
    - <u>Deep Learning Layers</u> - Lower Level Layers are embedded at the UAV itself, where the inference accuracy is low. Higher Level Layers are deployed at the MES, where intermediate outputs from UAV could be offloaded to MES to improve accuracy.
    - <u>Inference Modes</u> -
      Fast Inference - given by lower layers (UAV).
      Enhanced Inference - given by higher layers (MES)
    - <u>Offloading Strategy</u> -
      Binary Offloading - Either fast inference or enhanced inference.
      Partial Offloading - Can be a combination of the two.
    - <u>Inference Error Rate</u> - Characterized by the Intersection-over-Union

(IoU, denoted by ω), where finally, the inference error rate is given by: $\epsilon = 1 - \omega$, where $\epsilon \in [0,1]$

Two offloading strategies are considered – Binary and Partial; and offloading probabilities are calculated for each case to minimize the Cost for delay and energy consumption, where we calculate the optimal probability to distribute the tasks between the UAVs and the MES.

Combining Task- and Data-Level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs

In the previous paper, we have identified the various parameters that we need to take into consideration for our project, namely Inference Accuracy, Latency, Energy Consumption and Throughput. In this paper, we focus on one of these important parameters - the Throughput (becomes all the more significant in case of UAV-to-UAV communication).

- We make use of parallelism in Deep Learning models to increase the throughput among devices (U2U or U2G) and thus improve the inference accuracy.
- Task-Level Parallelism - Means different layers of the CNN can be computed simultaneously, which is to be implemented by the CPUs.
- Data-Level Parallelism - Means different computations of the same CNN layer can be done simultaneously, which is to be implemented by GPUs.
- The CPUs would offload the data-parallel computations within a layer to the GPUs.

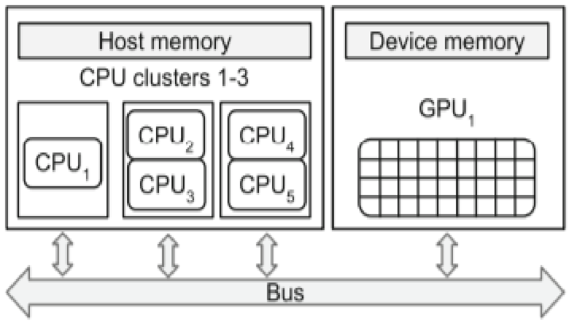


Fig. 2. MPSoC

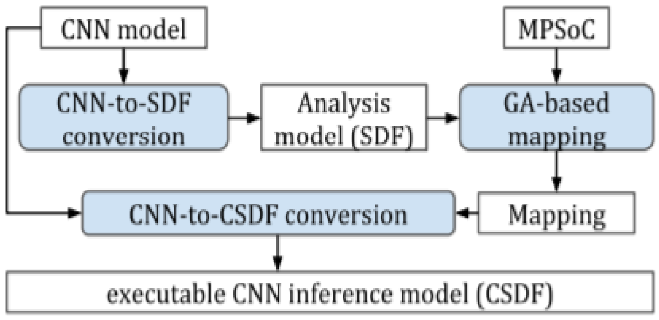


Fig. 3. Our methodology

- 4 Criteria to achieve high Throughput -
    - Efficient handling of task-level parallelism by CPUs
    - CPU workload balancing
    - Efficient handling of data-level parallelism by GPUs
    - Efficient usage of both types of parallelism such that the computation capacity of the MPSoC is matched properly

Methodology followed in the Paper -

Step 1: Convert the CNN model to an SDF (Synchronous DataFlow) model. The SDF model explicitly specifies task- and data-level parallelism, available in a CNN, as well as it explicitly specifies the tasks communication and synchronization mechanisms.

Step 2: We utilize a Genetic Algorithm (GA) to find an efficient mapping of the SDF model on an embedded CPUs-GPUs MPSoC. The mapping describes the distribution of the CNN inference computational workload on an embedded MPSoC.

Step 3: We use the mapping, obtained in Step 2, to convert a CNN model into a final platform-aware executable CycloStatic DataFlow (CSDF) application model. The CSDF model describes the CNN inference as an application, and efficiently matches the computational capacity of the MPSoC.

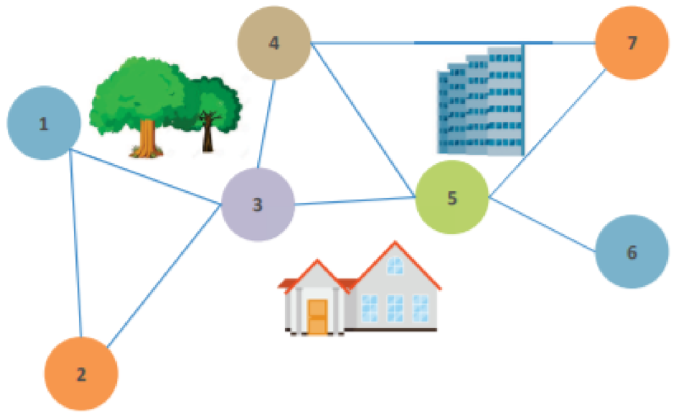
We have now seen how to efficiently transfer data from the UAVs to the Edge Servers by increasing the Throughput while considering other parameters of latency, energy consumption and the model inference accuracy. However, we also need to consider the transmission channel constraints and also the availability of a limited bandwidth only. Thus, sending the entire data collected by the Edge Devices (UAVs) to the Edge Servers is not a very feasible idea, and so we move on to a fairly new area - Federated Learning, where the UAVs would be required to send only the local model parameters and not the entire data.

Decentralized Federated Learning via SGD over Wireless D2D Networks

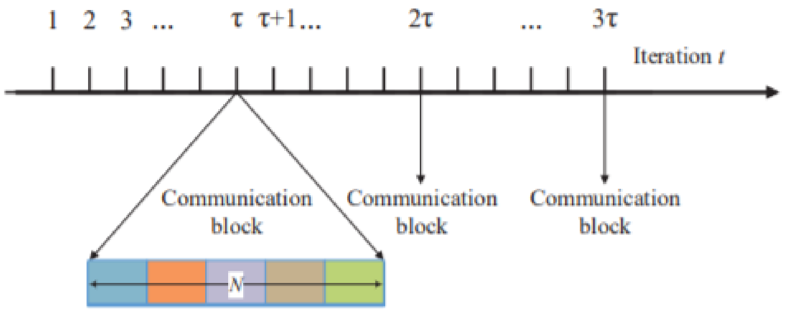
This paper mainly focuses on introducing the basics of Federated Learning, which would be the core of the rest of our literature review.

- Federated Learning (FL) enables collaborative training of a machine learning model over distributed data sets and computing resources with limited disclosure of local data.
- Effective communication is paramount for FL, as a significant amount of data needs to be exchanged among devices.
- Two main topologies are followed -
    - <u>Star Topology</u> - Exchange of information takes place through a center node.
    - <u>Decentralized Device-to-Device (D2D) Topology</u> - Devices can communicate only with their neighbours, and consensus mechanisms are needed to achieve a common goal. These are characterized by Graphs.
- A well known solution to ML problems in D2D networks is the DSGD, which stands for Decentralized Stochastic Gradient Descent.
- This solution guarantees convergence to optimality under assumptions of convexity and connectivity.
- The implementation protocols for both analog and digital transmissions have been discussed, wherein the analog model leverages over-the-air computing.

A very basic Decentralized Federated Learning model image is given below -



(a) A connectivity graph for a D2D wireless network with path loss, fading and blockages.



(b) Timeline of iterations with multi-slotted communication blocks used to exchange model information.

Fig. 1. Decentralized federated learning over wireless D2D networks.

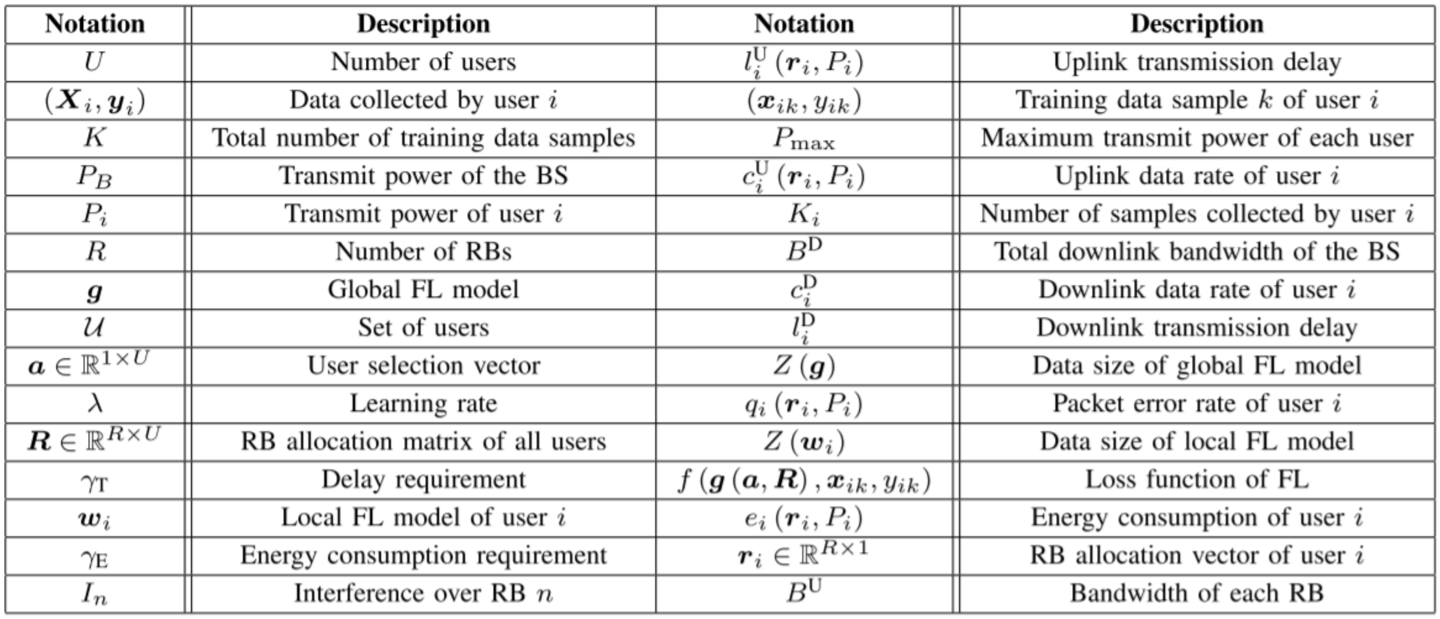
A Joint Learning and Communications Framework for Federated Learning Over Wireless Networks

In this paper, we formulate and solve an Optimization Problem to factor-in all of the constraints that we have gone through in the aforementioned papers.

- We consider three main factors to formulate our problem -
    - Minimizing the global FL loss function.
    - We are wirelessly transmitting the local FL parameters to the Base Station, and hence allocation of wireless resources is another important factor. This is called the uplink resource block (RB) allocation.
    - Packet errors might cause problems while evaluating our FL Model,

and so, selection of users is the final factor.

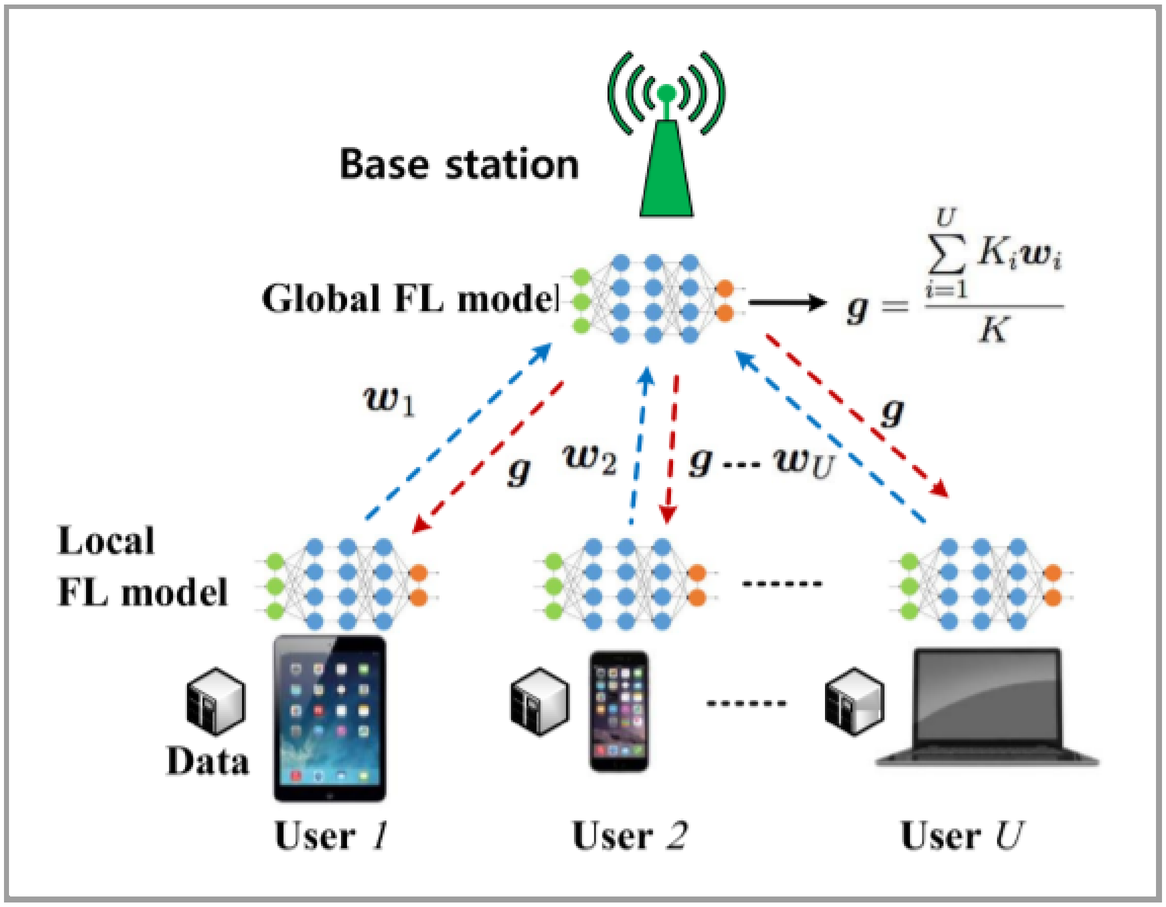
List of Notations used in the Paper -



Notation	Description	Notation	Description
U	Number of users	$l_i^{\mathrm{U}}(\boldsymbol{r}_i, P_i)$	Uplink transmission delay
$(\boldsymbol{X}_i, \boldsymbol{y}_i)$	Data collected by user i	$(\boldsymbol{x}_{ik}, y_{ik})$	Training data sample k of user i
K	Total number of training data samples	$P_{\max}$	Maximum transmit power of each user
P_B	Transmit power of the BS	$c_i^{\mathrm{U}}(\boldsymbol{r}_i, P_i)$	Uplink data rate of user i
P_i	Transmit power of user i	K_i	Number of samples collected by user i
R	Number of RBs	B^{D}	Total downlink bandwidth of the BS
$\boldsymbol{g}$	Global FL model	c_i^{D}	Downlink data rate of user i
$\mathcal{U}$	Set of users	l_i^{D}	Downlink transmission delay
$\boldsymbol{a} \in \mathbb{R}^{1 \times U}$	User selection vector	$Z(\boldsymbol{g})$	Data size of global FL model
λ	Learning rate	$q_i(\boldsymbol{r}_i, P_i)$	Packet error rate of user i
$\boldsymbol{R} \in \mathbb{R}^{R \times U}$	RB allocation matrix of all users	$Z(\boldsymbol{w}_i)$	Data size of local FL model
γ_{T}	Delay requirement	$f(\boldsymbol{g}(\boldsymbol{a}, \boldsymbol{R}), \boldsymbol{x}_{ik}, y_{ik})$	Loss function of FL
$\boldsymbol{w}_i$	Local FL model of user i	$e_i(\boldsymbol{r}_i, P_i)$	Energy consumption of user i
γ_{E}	Energy consumption requirement	$\boldsymbol{r}_i \in \mathbb{R}^{R \times 1}$	RB allocation vector of user i
I_n	Interference over RB n	B^{U}	Bandwidth of each RB

The Machine Learning Model -

$$\min_{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_U} \frac{1}{K} \sum_{i=1}^{U} \sum_{k=1}^{K_i} f(\boldsymbol{w}_i, \boldsymbol{x}_{ik}, y_{ik}),$$
$$\text{s. t. } \boldsymbol{w}_1 = \boldsymbol{w}_2 = \ldots = \boldsymbol{w}_{\mathrm{U}} = \boldsymbol{g},$$



After identifying this basic model, we factor in further communication and user-selection constraints to get the final Optimization Problem -

$$\min_{\boldsymbol{a},\boldsymbol{P},\boldsymbol{R}} \frac{1}{K}\sum_{i=1}^{U}\sum_{k=1}^{K_i} f\left(\boldsymbol{g}\left(\boldsymbol{a},\boldsymbol{P},\boldsymbol{R}\right),\boldsymbol{x}_{ik},y_{ik}\right), \tag{11}$$

$$\text{s. t. } a_i, r_{i,n} \in \{0,1\}, \quad \forall i \in \mathcal{U}, n = 1,\ldots,R, \tag{11a}$$

$$\sum_{n=1}^{R} r_{i,n} = a_i, \quad \forall i \in \mathcal{U}, \tag{11b}$$

$$l_i^{\mathrm{U}}\left(\boldsymbol{r}_i, P_i\right) + l_i^{\mathrm{D}} \leq \gamma_{\mathrm{T}}, \ \ \forall i \in \mathcal{U}, \tag{11c}$$

$$e_i\left(\boldsymbol{r}_i, P_i\right) \leq \gamma_{\mathrm{E}}, \ \ \forall i \in \mathcal{U}, \tag{11d}$$

$$\sum_{i\in\mathcal{U}} r_{i,n} \leq 1, \ \ \forall n = 1,\ldots,R, \tag{11e}$$

$$0 \leq P_i \leq P_{\max}, \quad \forall i \in \mathcal{U}, \tag{11f}$$

The constraints of the above problem can be explained as follows -

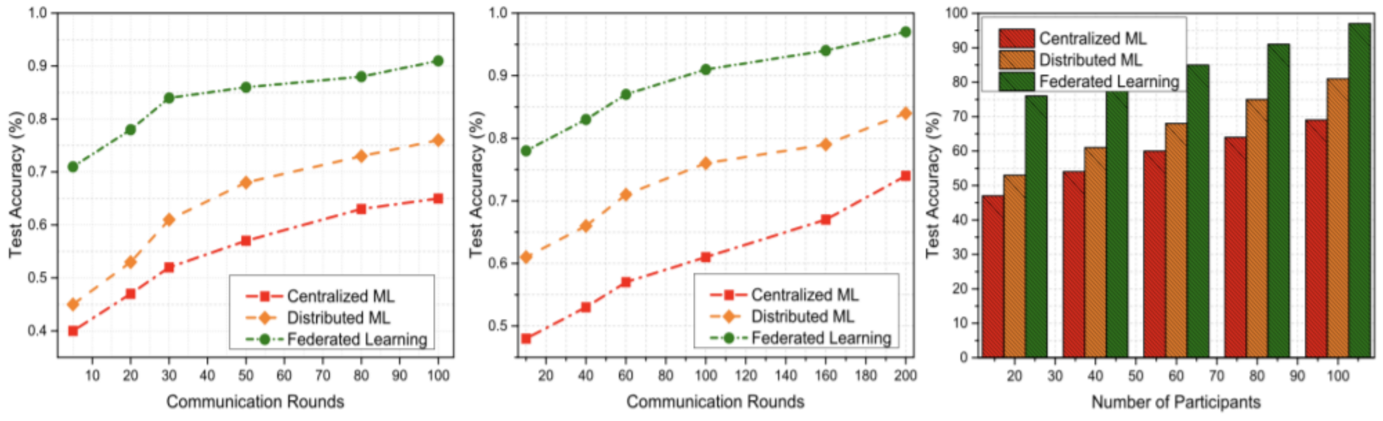
- γ_T is the delay requirement for implementing the FL algorithm, γ_E is the energy consumption of the FL algorithm and B is the total downlink bandwidth.
- (11a) and (11b) indicates that each user can occupy only one RB for uplink data transmission.
- (11c) is the delay needed to execute the FL algorithm at each learning step.
- (11d) is the energy consumption requirement of performing an FL algorithm at each learning step.
- (11e) indicates that each uplink RB can be allocated to at most one user.
- (11f) is a maximum transmit power constraint.

This problem is then solved by means of the Hungarian Algorithm.

Federated Learning Versus Classical Machine Learning: A Convergence Comparison

- In this paper, three main model classifiers are considered -
  - Classical Machine Learning
  - Distributed Machine Learning
  - Federated Learning
- Data training assumptions are much more robust for FL as compared to ML. FL assumes non i.i.d. and uneven distribution of data among participants. This is also the key difference between Distributed machine learning and federated learning.

Experiments were conducted to compare the convergence rates of the three models and the results obtained are as follows -



(a) Scenario one (b) Scenario two (c) Scenario three

Fig. 2: Convergence comparison on MNIST dataset in three different scenarios: 1) 100 communication rounds, 2) 200 communication rounds, 3) various number of participants.

From the three scenarios, it is clearly evident how FL outperforms centralized and distributed ML. In recent times, due to the increasing amount of big data generated, performing all computations on a single server is highly infeasible. Increasing privacy threats are also a major concern, which can be tackled by FL as it performs local computations on the edge devices itself.

Why Federated Learning?

The training data in wireless networks is unevenly distributed over a large amount of resource-constrained user devices, whereas each device only owns a small fraction of data. Therefore, learning at devices suffers from isolated data islands and will induce long training latency, making it hard to implement AI algorithms at user devices. Conventional solutions generally offload the local training data to a remote cloud center for centralized learning. Nevertheless, this method suffers from two key disadvantages. On the one hand, the latency for data transmission is often very large because of the limited communication resources. On the other hand, the privacy information involved in the training data may be leaked since the cloud center is inevitably attacked by some malicious third parties. Hence, traditional cloud-based learning framework is no longer suitable for the scenarios where data privacy is of paramount importance, such as intelligent healthcare systems and smart bank systems.

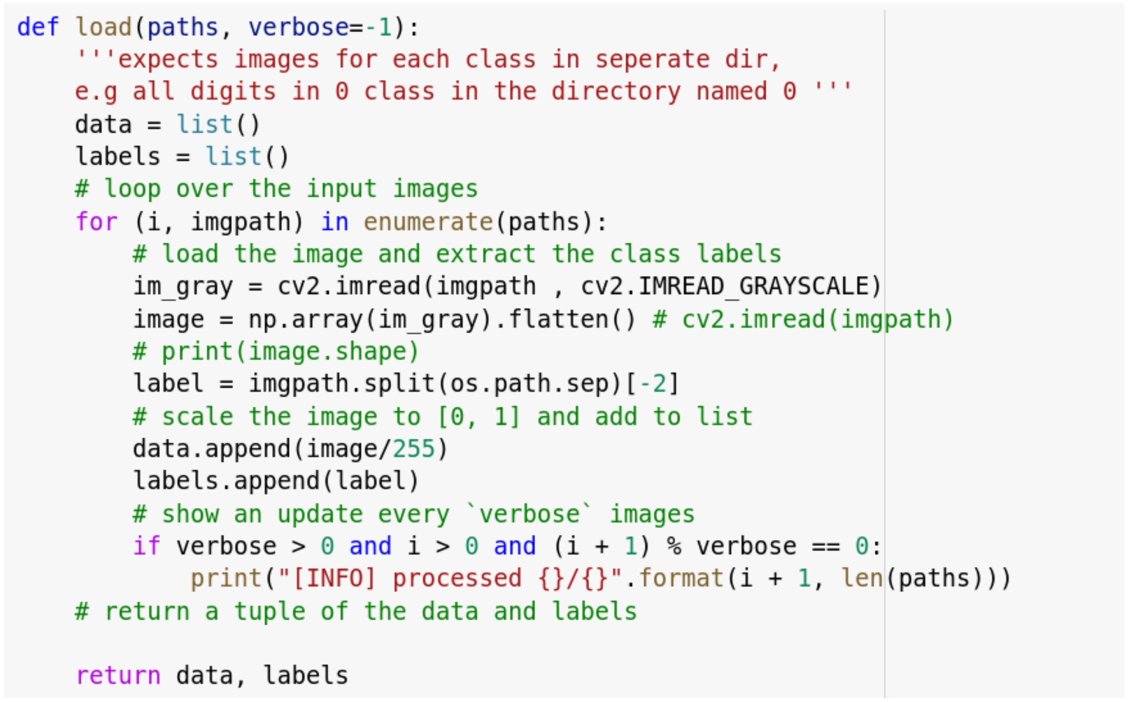
To address this issue, federated learning was innovated. The key idea of this framework is to aggregate locally computed learning updates (gradients or parameters) in a centralized node while keeping the privacy-sensitive data remained at local devices. Toward this end, the benefit of the shared models trained from the rich data can be reaped and the computation resources of user devices can be exploited

Rather than bringing the data to model, federated learning brings the model to data.

Federated Learning
Simulation - Code Walkthrough

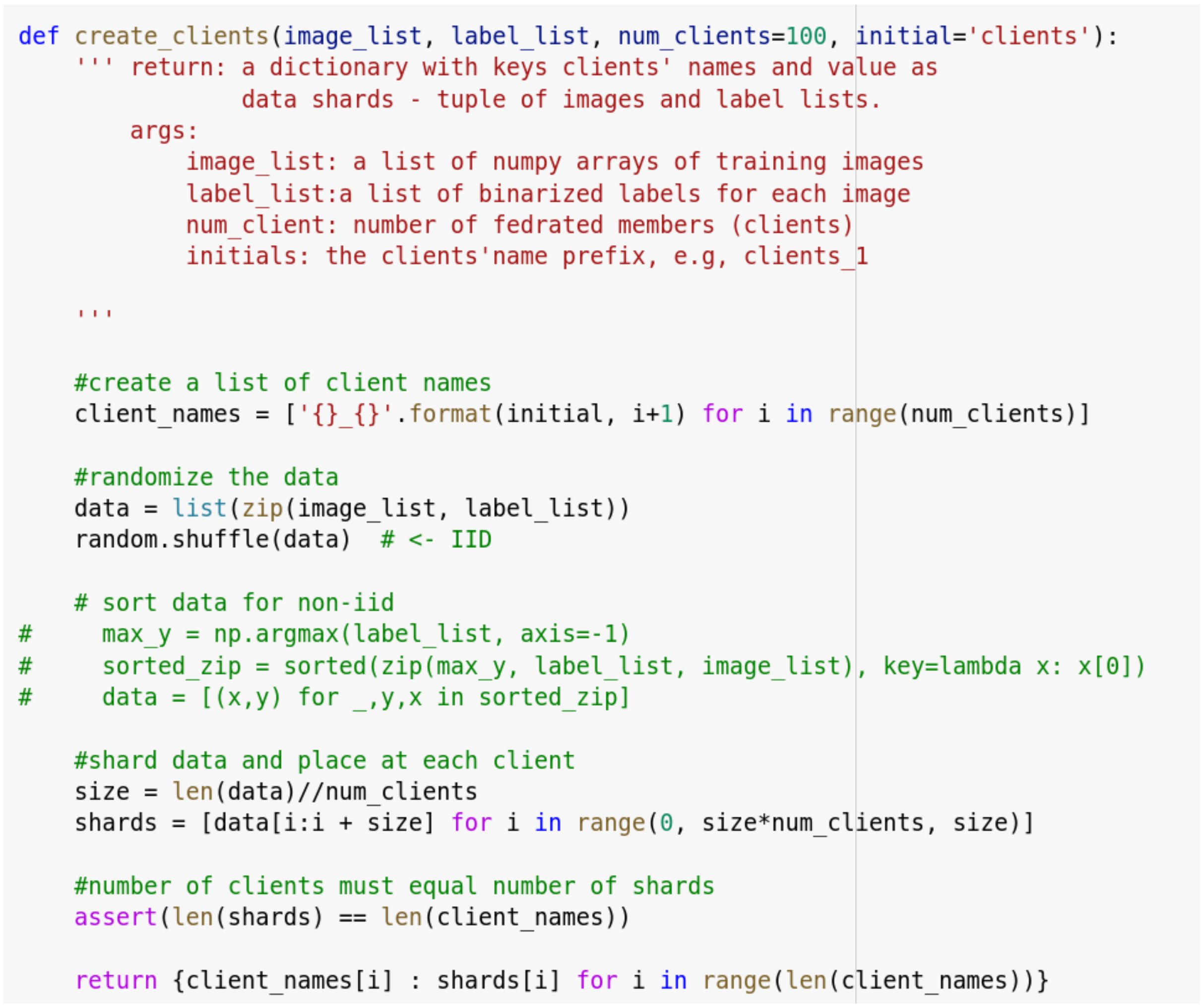
In this simulation, we train our model for classification on the MNIST dataset. The MNIST dataset consists of handwritten digits from 0-9 and it has 42000 digit images with each class kept in separate folders. The data partitioning is going to create an independent and identical dataset. We first create utility functions for a lot of operations that are required for the training loop.

The following code block shows such a utility function to load the MNIST data which will help us in taking the image from the path location and then also putting a label. The images are converted into numpy arrays which can be used for the matrix operations required during the gradient descent algorithm of deep learning technique being used here.



```
def load(paths, verbose=-1):
    '''expects images for each class in seperate dir,
    e.g all digits in 0 class in the directory named 0 '''
    data = list()
    labels = list()
    # loop over the input images
    for (i, imgpath) in enumerate(paths):
        # load the image and extract the class labels
        im_gray = cv2.imread(imgpath , cv2.IMREAD_GRAYSCALE)
        image = np.array(im_gray).flatten() # cv2.imread(imgpath)
        # print(image.shape)
        label = imgpath.split(os.path.sep)[-2]
        # scale the image to [0, 1] and add to list
        data.append(image/255)
        labels.append(label)
        # show an update every `verbose` images
        if verbose > 0 and i > 0 and (i + 1) % verbose == 0:
            print("[INFO] processed {}/{}".format(i + 1, len(paths)))
    # return a tuple of the data and labels

    return data, labels
```

The following code cell is responsible for creating clients from the whole dataset that is present. In real world implementation of federated learning, the data is usually isolated in user devices however, since MNIST dataset gives us the whole data at the same palace, we have manually partitioned the data into client’s data. Note that after dividing the data into client data, we are creating a tensorflow dataset object which is being fed in as a batch of data for training purposes.

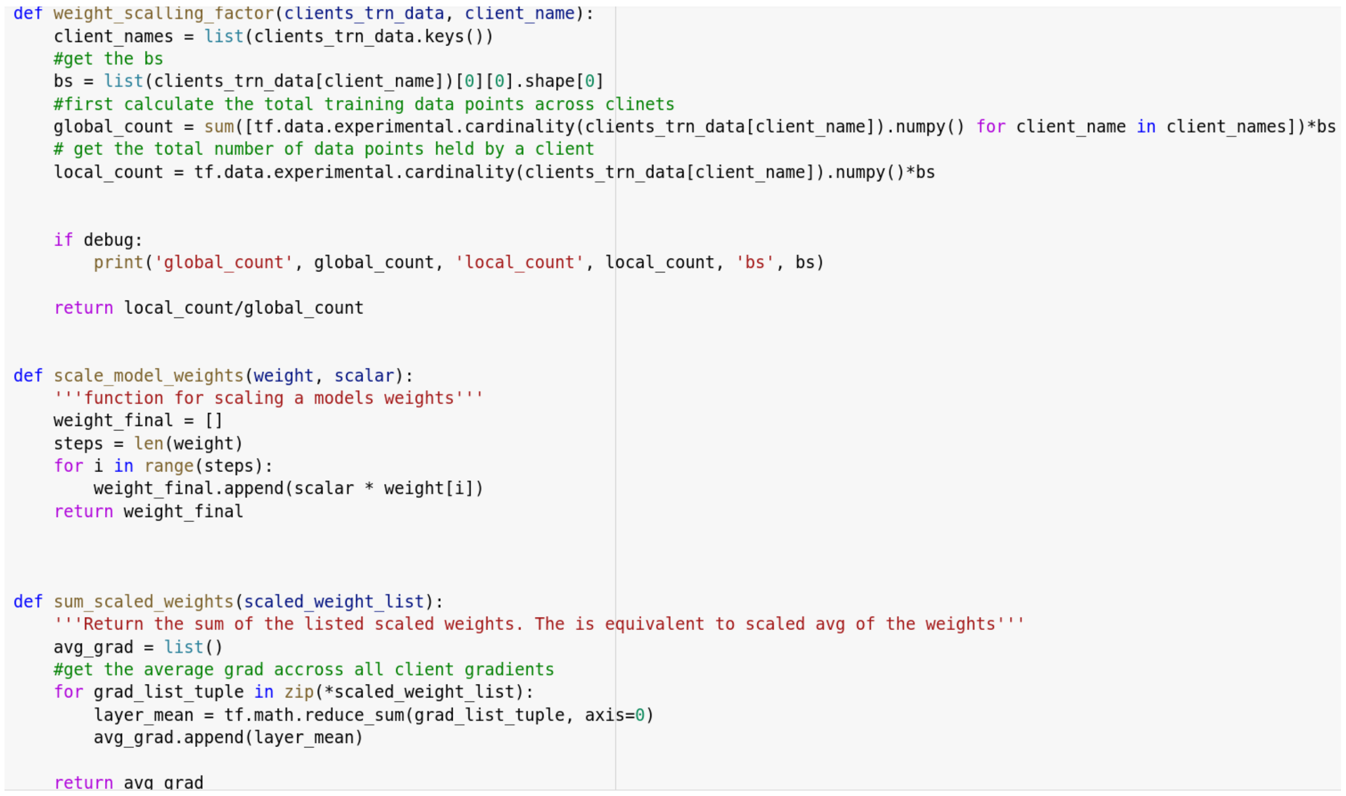


```python
def create_clients(image_list, label_list, num_clients=100, initial='clients'):
    ''' return: a dictionary with keys clients' names and value as 
                data shards - tuple of images and label lists.
        args: 
            image_list: a list of numpy arrays of training images
            label_list:a list of binarized labels for each image
            num_client: number of fedrated members (clients)
            initials: the clients'name prefix, e.g, clients_1 
            
    '''

    #create a list of client names
    client_names = ['{}_{}'.format(initial, i+1) for i in range(num_clients)]

    #randomize the data
    data = list(zip(image_list, label_list))
    random.shuffle(data)  # <- IID
    
    # sort data for non-iid
#     max_y = np.argmax(label_list, axis=-1)
#     sorted_zip = sorted(zip(max_y, label_list, image_list), key=lambda x: x[0])
#     data = [(x,y) for _,y,x in sorted_zip]

    #shard data and place at each client
    size = len(data)//num_clients
    shards = [data[i:i + size] for i in range(0, size*num_clients, size)]

    #number of clients must equal number of shards
    assert(len(shards) == len(client_names))

    return {client_names[i] : shards[i] for i in range(len(client_names))}
```

The next 3 utility functions are the core of the federated learning that will be used in the training loop. The *weight_scalling_factor* function calculates the proportion of a client's local training data with the overall training data held by all clients. First we obtained the client's batch size and used that to calculate its number of data points. We then obtained the overall global training data size. Finally we calculated the scaling factor as a fraction. The training data will be disjointed, therefore no single client can correctly estimate the quantity of the combined set. In that case, each client will be expected to indicate the number of data points they trained with while updating the server with new parameters after each local training step.

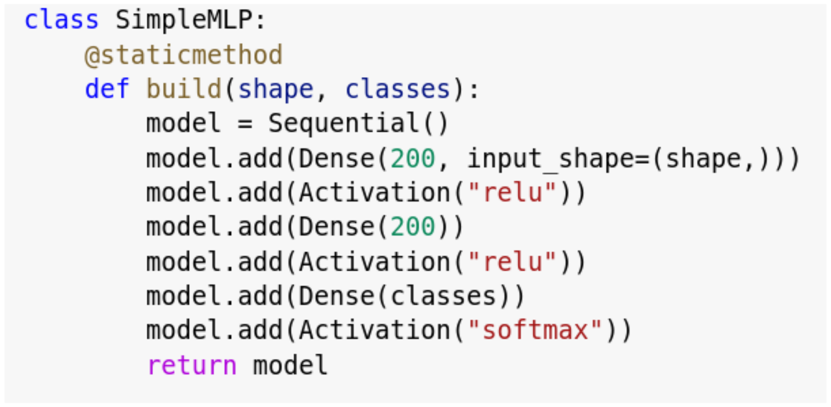
scale_model_weights scales each of the local model's weights based the value of their scaling factor calculated in *weight_scalling_factor.*

sum_scaled_weights sums all clients' scaled weights together.



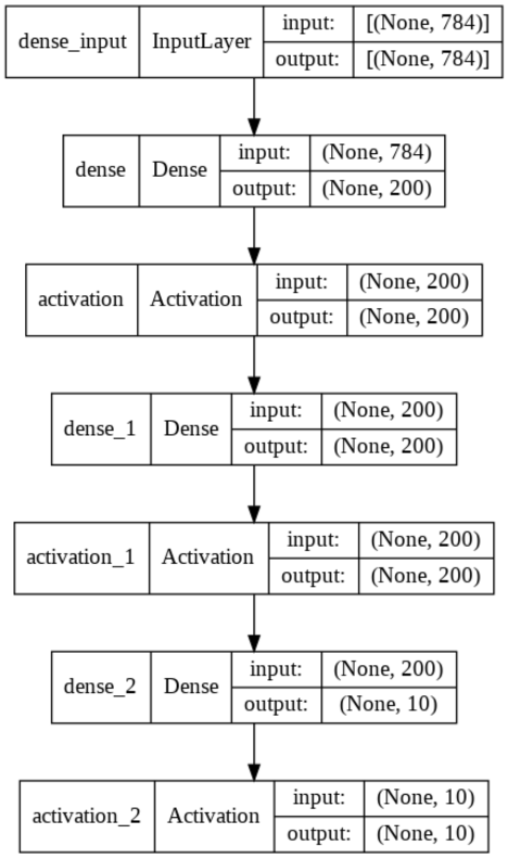
```python
def weight_scalling_factor(clients_trn_data, client_name):
    client_names = list(clients_trn_data.keys())
    #get the bs
    bs = list(clients_trn_data[client_name])[0][0].shape[0]
    #first calculate the total training data points across clinets
    global_count = sum([tf.data.experimental.cardinality(clients_trn_data[client_name]).numpy() for client_name in client_names])*bs
    # get the total number of data points held by a client
    local_count = tf.data.experimental.cardinality(clients_trn_data[client_name]).numpy()*bs


    if debug:
        print('global_count', global_count, 'local_count', local_count, 'bs', bs)

    return local_count/global_count


def scale_model_weights(weight, scalar):
    '''function for scaling a models weights'''
    weight_final = []
    steps = len(weight)
    for i in range(steps):
        weight_final.append(scalar * weight[i])
    return weight_final



def sum_scaled_weights(scaled_weight_list):
    '''Return the sum of the listed scaled weights. The is equivalent to scaled avg of the weights'''
    avg_grad = list()
    #get the average grad accross all client gradients
    for grad_list_tuple in zip(*scaled_weight_list):
        layer_mean = tf.math.reduce_sum(grad_list_tuple, axis=0)
        avg_grad.append(layer_mean)

    return avg_grad
```

We have used a basic 3 layer neural network consisting of linear layers on top of activation functions of ReLU which is appended by a softmax function.



```python
class SimpleMLP:
    @staticmethod
    def build(shape, classes):
        model = Sequential()
        model.add(Dense(200, input_shape=(shape,)))
        model.add(Activation("relu"))
        model.add(Dense(200))
        model.add(Activation("relu"))
        model.add(Dense(classes))
        model.add(Activation("softmax"))
        return model
```

The neural network looks like the following figure which takes in a input of flattened image of 784(this number comes up because each MNIST image is of dimensions 28*28 so flattening it gives us 784).

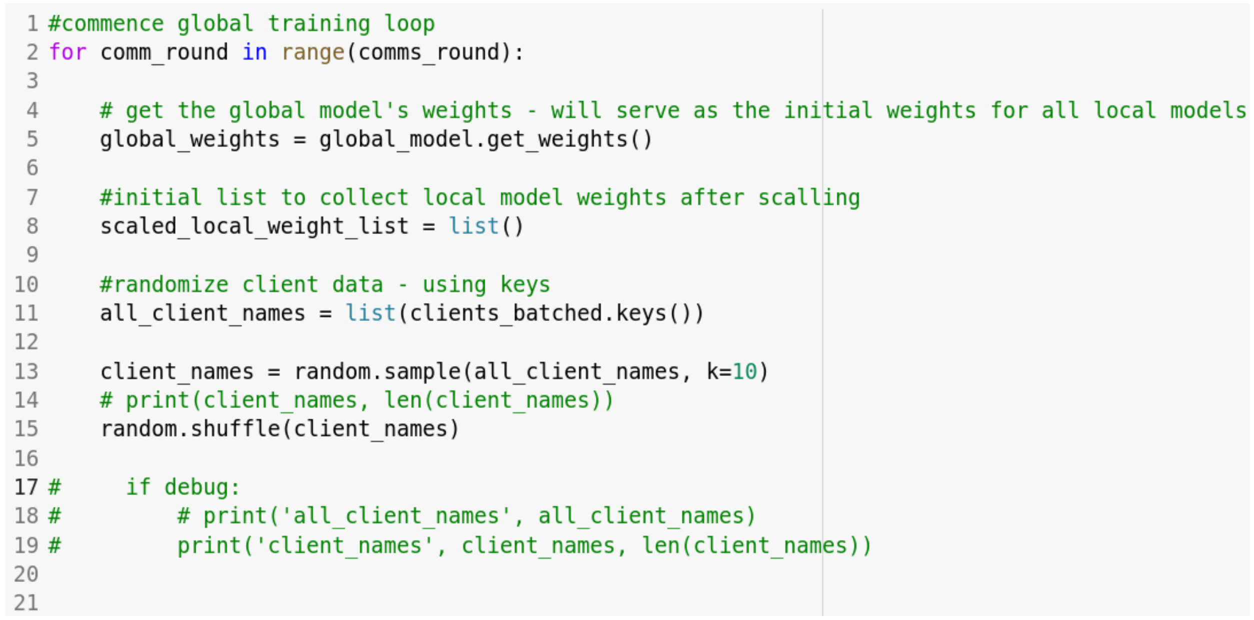


The hyperparameters used for the training are the following. Note: Here, we are using the variable name comms_round which indicates the epochs for any generic deep learning training. The communication rounds refer to how many times a client node interacts with the central node in order to update the global gradients.

```
lr = 0.01
comms_round = 20
loss='categorical_crossentropy'
metrics = ['accuracy']
optimizer = SGD(lr=lr,
                decay=lr / comms_round,
                momentum=0.9
               )
```

Now, we move onto the training loop. This is broken up into a nested for loop where the outer for loop indicates the operations taking place in the server node while the inner for loop takes care of the backpropagation taking place in the client node.

The following code cell is the outer for loop. First, we initialize the the global weights and shuffle the client keys in order to apply a regularizing factor while training.



```
#commence global training loop
for comm_round in range(comms_round):

    # get the global model's weights - will serve as the initial weights for all local models
    global_weights = global_model.get_weights()

    #initial list to collect local model weights after scalling
    scaled_local_weight_list = list()

    #randomize client data - using keys
    all_client_names = list(clients_batched.keys())

    client_names = random.sample(all_client_names, k=10)
    # print(client_names, len(client_names))
    random.shuffle(client_names)

#     if debug:
#         # print('all_client_names', all_client_names)
#         print('client_names', client_names, len(client_names))


```

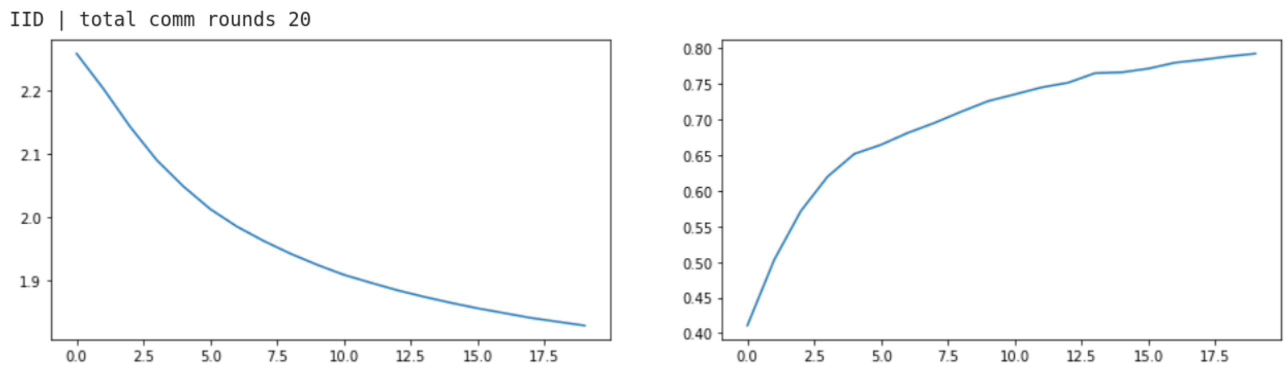
The next code cell is the inner for loop which depicts how the gradients are being updated in the client node. Here, line 49 shows us how the model aggregation (federated averaging) is taking place. We are estimating the weight parameters for each client based on the loss values recorded across every data point they trained with. We have also scaled each of those parameters and sum them all component-wise.

```
    #loop through each client and create new local model
    for client in client_names:
        smlp_local = SimpleMLP()
        local_model = smlp_local.build(build_shape, 10)
        local_model.compile(loss=loss,
                      optimizer=optimizer,
                      metrics=metrics)

        #set local model weight to the weight of the global model
        local_model.set_weights(global_weights)

        #fit local model with client's data
        local_model.fit(clients_batched[client], epochs=1, verbose=0)

        #scale the model weights and add to list
        scaling_factor = 0.1 # weight_scalling_factor(clients_batched, client)
        # print('scaling_factor', scaling_factor)
        scaled_weights = scale_model_weights(local_model.get_weights(), scaling_factor)
        scaled_local_weight_list.append(scaled_weights)

        #clear session to free memory after each communication round
        K.clear_session()

    #to get the average over all the local model, we simply take the sum of the scaled weights
    average_weights = sum_scaled_weights(scaled_local_weight_list)

    #update global model
    global_model.set_weights(average_weights)

    #test global model and print out metrics after each communications round
    for(X_test, Y_test) in test_batched:
        global_acc, global_loss = test_model(X_test, Y_test, global_model, comm_round)
        global_acc_list.append(global_acc)
        global_loss_list.append(global_loss)
```

We arrive at the following result when running the for loop for 20 communication rounds(global epochs).

```
comm_round: 0 | global_acc: 41.071% | global_loss: 2.2585582733154297
comm_round: 1 | global_acc: 50.262% | global_loss: 2.2031443119049072
comm_round: 2 | global_acc: 57.143% | global_loss: 2.142970085144043
comm_round: 3 | global_acc: 61.976% | global_loss: 2.089998245239258
comm_round: 4 | global_acc: 65.167% | global_loss: 2.0482840538024902
comm_round: 5 | global_acc: 66.452% | global_loss: 2.012636184692383
comm_round: 6 | global_acc: 68.119% | global_loss: 1.984912633895874
comm_round: 7 | global_acc: 69.524% | global_loss: 1.962205410003662
comm_round: 8 | global_acc: 71.095% | global_loss: 1.942123532295227
comm_round: 9 | global_acc: 72.571% | global_loss: 1.9244935512542725
comm_round: 10 | global_acc: 73.524% | global_loss: 1.908677101135254
comm_round: 11 | global_acc: 74.500% | global_loss: 1.8962006568908691
comm_round: 12 | global_acc: 75.167% | global_loss: 1.8842144012451172
comm_round: 13 | global_acc: 76.500% | global_loss: 1.8738600015640259
comm_round: 14 | global_acc: 76.619% | global_loss: 1.8643903732299805
comm_round: 15 | global_acc: 77.143% | global_loss: 1.855697751045227
comm_round: 16 | global_acc: 77.976% | global_loss: 1.8481093645095825
comm_round: 17 | global_acc: 78.381% | global_loss: 1.8405101299285889
comm_round: 18 | global_acc: 78.857% | global_loss: 1.834467887878418
comm_round: 19 | global_acc: 79.238% | global_loss: 1.8284481763839722
```

Through the results logged, we get a loss function which keeps on decreasing as shown on the left while the accuracy keeps increasing shown on the right figure.



RESULTS & CONCLUSIONS

In this project, a comprehensive study of UAVs was done. We have presented an elaborate summary of literature review undertaken, followed by why there is a recent shift in the focus towards federated learning and then proceeding onto doing a federated learning simulation on MNIST dataset.

REFERENCES

1. Jouhari, Mohammed & Al-Ali, Abdulla & Baccour, Emna & Mohamed, Amr & Erbad, Aiman & Guizani, Mohsen & Hamdi, Mounir. (2021). "*Distributed CNN Inference on Resource-Constrained UAVs for Surveillance Systems: Design and Optimization*".

2. Wang, Siqi & Ananthanarayanan, Gayathri & Zeng, Yifan & Goel, Neeraj & Pathania, Anuj & Mitra, Tulika. (2019). "*High-Throughput CNN Inference on Embedded ARM big.LITTLE Multi-Core Processors*". IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. PP. 1-1. 10.1109/TCAD.2019.2944584.

3. Dong, Chao & Shen, Yun & Qu, Yuben & Wu, Qihui & wu, Fan & Chen, Guihai. (2020). "*UAVs as a Service: Boosting Edge Intelligence for Air-Ground Integrated Networks*".

4. J. Ren, G. Yu and G. Ding, "*Accelerating DNN Training in Wireless Federated Edge Learning Systems*", in IEEE Journal on Selected Areas in Communications, vol. 39, no. 1, pp. 219-232, Jan. 2021, doi: 10.1109/JSAC.2020.3036971.

5. Bo Yang, Xuelin Cao, Chau Yuen and Lijun Qian. (2020). "*Offloading Optimization in Edge Computing for Deep Learning Enabled Target Tracking by Internet-of-UAVs*". IEEE Internet of Things Journal.

6. Svetlana Minakova, Erqian Tang and Todor Stefanov. (2020). "*Combining Task- and Data-Level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs*".

7. Hong Xing, Osvaldo Simeone and Suzhi Bi. (2020). "*Decentralized Federated Learning via SGD over Wireless D2D Networks*".

8. Mingzhe Chen, Zhaohui Yang, Walid Saad, Changchuan Yin, H. Vincent Poor, and Shuguang Cui. (2021). *“A Joint Learning and Communications Framework for Federated Learning Over Wireless Networks”*. IEEE Transactions on Wireless Communications, Vol. 20, No. 1, Jan 2021.

9. Muhammad Asad, Ahmed Moustafa, and Takayuki Ito. *“Federated Learning Versus Classical Machine Learning: A Convergence Comparison”*.

10. https://www.tensorflow.org/federated/federated_learning