Abstract

In today's world, Neural Network training is processed in cloud and servers with large-scale data by super computers. This technique lets companies to collect data from users by neglecting data privacy. Also, the data in edge devices is becoming more qualified so data size increases. For this reason, the Internet is filled with redundant data. Besides, edge devices are being manufactured with more reinforced computational power every year so that they can compute more complex processes. However, Federated Learning prevents privacy issues by letting edge devices to store their local data and gives opportunity to edge devices by letting them to train neural network model with their local data. Federated Learning is basically a distributed training technique that trains neural network models across edge devices. In this project, Federated Learning is used as centralized architecture by a server and clients. The server operates the process and clients train the neural network model. MNIST and CIFAR-10 are used as dataset, LeNet-5 used as Convolutional Neural Network. I found that model selection to start training, number of communication rounds between client and the server and complexity of dataset play critical role to get better model performance by Federated Learning. Specifically, having an IID vs non-IID classes dataset in federated clients is more crucial than having a balanced vs unbalanced count (corpus size) representation.

Contents

I. Introduction

Federated learning is a distributed machine learning technique to train neural network models across edge devices with small amount of data; instead, training model in cloud or server with centralized, big data corpus. In this project, I implemented an application that clients can interact with a server to be part of federated learning. Server is responsible for receiving, sending, and aggregating neural network models. The clients are only assigned to training corresponding model with their local data.

The second chapter, background, will be explanation of techniques, tools and technologies that are implemented in the project. The third chapter will cover the problems in today's world that federated learning points out and how it solves these problems also the requirements of the application. In fourth chapter, I will explain my approach to the problem. The fifth chapter will be demonstration and discussion of test results. The sixth chapter will be about related works of my project and how they differentiation from my approach. In seventh chapter, there is my conclusion and future works that I am going to try to achieve in the next semester.

II. Background

During the project, I have used DNN techniques, NN libraries, and python tools to solve the problem. Brief descriptions as follows:

TensorFlow: TensorFlow is a library for machine learning. I have used TensorFlow's Keras API to build, train, update, validate and store machine learning models.

SocketIO: SocketIO provides real-time bidirectional communication between clients and a server. It has been utilized on machine learning model transfer between clients and a server.

Asyncio: Asyncio is a built-in library that includes concurrent keywords. It has been used to make the code concurrent.

Scikit-learn: I have used different metrics from Scikit-learn library to validate machine learning models.

Matplotlib: Matplotlib has been used to visualize confusion matrix.

H5py: H5py was used to manipulate hdf5 files which is a format to save architecture and weights of machine learning models.

IID Data: IID stands for independent and identically distributed. In this paper, it was used in where the data was distributed among clients so that each client had data within all the labels with equal number without overlapping.

Non-IID Data: It means not independent and identically distributed. This technique was used in the scenario where clients have different number of labels and different amount of data in their local storage.

Communication Round: It is a process starting from sending global models to clients by a server. Then, clients train the model with local data. Finally, the process ends with aggregation of models from each client by a server.

III. Problem Statement

In today's world, machine learning model training is a process where computation power and data meets on a cloud server with powerful CPU/GPU and massive storage space. Supercomputers in cloud server store data by undermining data privacy; therefore, this situation carries data leakage risks and raises privacy concerns [1]. Also, number of edge devices and their capacities are increasing. This renders transfer of data from edge devices to servers unnecessary since network bandwidths are limited. Some of the commonly used model size reduction techniques such as quantization and weight sharing are lossy in nature, which affects their accuracy.

Besides storage improvement in edge devices, their computation power grows exponentially. Ability of edge devices is limited with only sending and receiving data to cloud server; however, they can do more complex processes rather than data convey.

Federated Learning solves this issue more than any technique. It is a distributed neural network training process. It provides privacy by letting clients to store their own data without sending central server or cloud. Also, it is more scalable solution because it uses client's computation power while training a model instead of collecting all data and train on a supercomputer. Besides, it helps clients to use better personalized models because personal data can be separated easily. However, it is costly to generate a neural network model for each client in cloud server.

In this semester, I started to implement decentralized federated learning application; however, it will be ready on next semester, so I decided to implement a centralized application. Requirements of the application as follows:

Functional Requirement 1.0

ID: 1
Title: Connection
Description: Clients should connect model server and server should be able to connect the server

Functional Requirement 1.1

ID: 2

Title: Training model

Description: Clients should be able to train their model in a communication round

Functional Requirement 1.2

ID: 3

Title: Fetch model

Description: Server should be able to fetch client's model at the end of a communication round

Functional Requirement 1.3

ID: 4

Title: Model distribution

Description: Client should be able to fetch global model after a communication round

Functional Requirement 1.4

ID: 5

Title: Model Architecture

Description: Neural Network Model architecture should be same in the server and the client

Non-Functional Requirement 1.5

ID: 6

Title: Response time

Description: Response time should be less than 120 seconds

Non-Functional Requirement 1.6

ID: 7

Title: Connection Size

Description: Server should let connect only 10 clients

IV. Solution Approach

Part 1: Application

My first approach to this problem was building a server and a client application with using SocketIO to transfer machine learning models between clients and a server. The reason I chose SocketIO was that I needed bidirectional transfer and ease of implementation. In Python, there is a package which is called python-socketIO that uses WebSocket, which is a communication protocol, connection to interact with client and server. Then, I thought using concurrency in some parts of the code would be efficient due to the fact that each client could connect the server and start to train model asynchronously. Thus, I have used asyncio library to use keywords that makes code concurrent.

Firstly, the server starts the first communication round and creates a neural network model for the task with given weight initialization parameter. However, the server never trains the model, only built it. After initialization, it sends the initialized model to the clients. While sending the model, it extracts weights from Keras model for each layer and saves in a Hash Map. In the Hash Map, keys consist of name of layer and values are weights of that corresponding layer. Then, model weights are converted into bytes by pickle because when data is transferred via sockets, actually byte of the data is transferred. In addition to weight, the model architecture is sent to the clients. It is basically a JSON file of Keras configurations that includes layer name, layer type and layer parameters such as kernel size, padding value etc. Each client receives the model architecture and decodes weight of the model from bytes to NumPy array. Before training, they build model with the architecture. Then, they set NumPy arrays as weight in regarding layer. Finally, each client starts training with their local data. They train model with their local epoch and batch size. When the training is finished, each client sends their local model to the server. The server takes all models from clients and aggregates them into one model which is called global model. In my solution, the aggregation is simply by taking average of weights in corresponding layer for all models. This technique called FedAvg [2]. Since the server starts the model at the very beginning, each client uses same model for training, so the model architecture is typical for each client. It prevents the issue of having different model architecture for clients. After aggregation process ends, it validates the new global model which is aggregated model on the test data. Then, the server sends global model to all clients and starts the new communication round. Clients take new global model and replaces with their previous local model by changing weights of each layer.

After that, clients start new training again with their local data and local settings i.e., local epoch size, batch size. This process continues until desired number of communication rounds is reached. At the end of the process, latest global model in the server is distributed to all clients. The major threat to the application is Byzantine Fault Tolerance since involve of harmful devices in training process can decrease the general model performance and let the system fails. Moreover, if the server is crashed for some reason or is captured for bad purpose, whole training process can fail. For this reason, there is a single point of failure issue.

Since I have only one computer, I opened new terminal tabs and run necessary files in them to represent clients and a server. The interaction between client and server is shown in *Figure 1*.

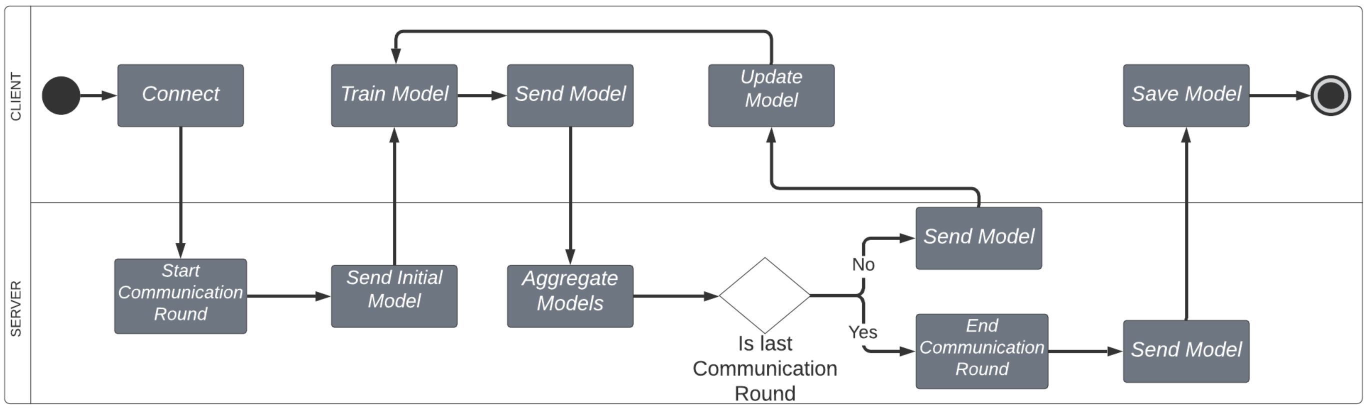

Figure 1

Part 2: Dataset Preparation

To test the model, I chose images as data because they fit perfectly to use case of federated learning. I have used two different image datasets, MNIST [3] and CIFAR-10 [4] respectively. Size of these datasets do not exceed my computer's RAM that's why I chose them.

The MNIST dataset of handwritten digits contains 60.000 images for training set and 10.000 images for testing set. There are 10 unique labels that represents numbers from 0 to 9. Each image has 28x28 pixels and only 1 channel because MNIST's images are white and black. You can see example images in *Figure 2*.

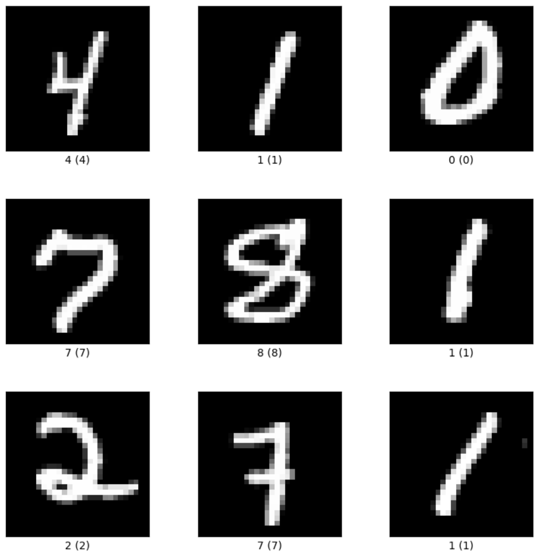

Figure 2

The CIFAR-10 dataset consists of images of 10 objects. It has 50.000 images for training set and 10.000 images for testing set. Each image has 32x32 resolutions and 3 color channels for red, green, and blue colors due to colored images despite MNIST. There are some examples from CIFAR-10 in *Figure-3*.

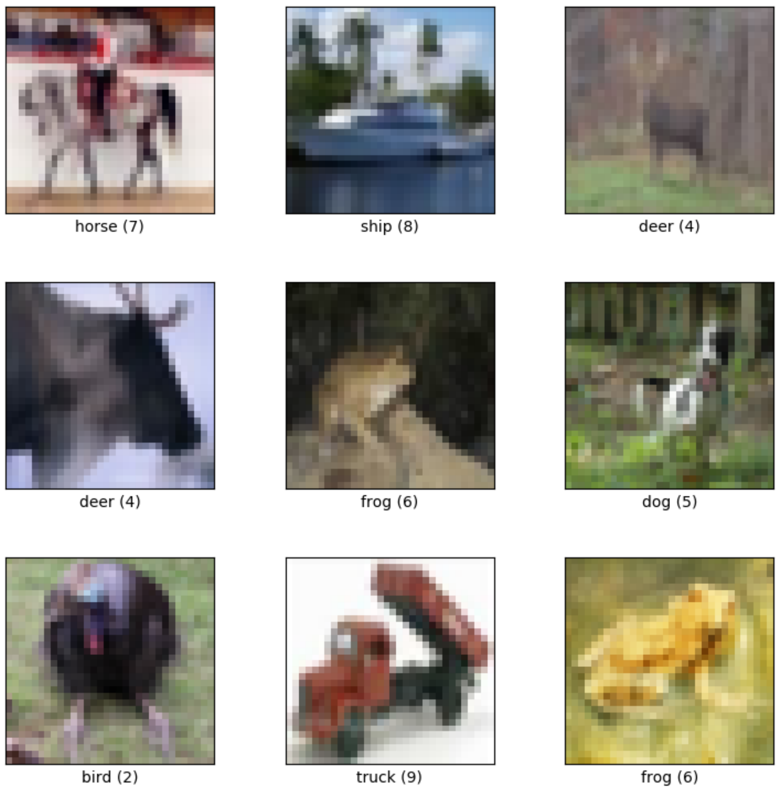

Figure 3

For MNIST and CIFAR-10 datasets, whole training sets were trained in a single client. Then, they were validated with test sets. It was baseline for accuracy perspective. Later, these results will be compared with federated learning's results.

For federated learning, there were two data separation techniques. Firstly, independent and identically distributed data (IID) was used. The data partitioned so that each client had same number of labels. For example, there are 50.000 training images in CIFAR-10 for 5000 images per label. There are 10 clients so each of them acquires 5000 training images for 500 images per label. Thus, clients train their model with only small patriation of the while training dataset. However, the test set is not partitioned, it stayed one piece.

The other technique was non independent and identically distributed data (Non-IID). It is a partition technique and for scenarios like each client has different data distribution and amount of data since data is not distributed equally and some clients can generate more or less data in most of real time applications. In this scenario, each client only possess data with two labels instead of 10.

There are two cases in Non-IID, balanced and unbalanced respectively. Firstly, each client has equal amount of data in the balanced option. For example, in CIFAR-10, client-1 has 2500 data for label 1 and 2500 data for label 2, client-2 has 2500 data for label 4 and 5. Thus, client-1 never sees from label 3 to 10 and client-2 never sees data from the other labels. The other case is unbalanced distribution which stands for cases such as some clients create more data, and some clients create less data. For example, in CIFAR-10, client-1 has 1200 data for label 1 and 3800 data for label 2. So, amount of data is changed; however, the client still has two labels.

Number of data instances for each client in IID, Non-IID balanced & unbalanced techniques are shown Table 1 and Table 2, for MNIST and CIFAR-10 respectively.

Client Number	IID	NON-IID Balanced	NON-IID Unbalanced
Client-1	6000	6000	2015
Client-2	6000	6000	9985
Client-3	6000	6000	7408
Client-4	6000	6000	4592
Client-5	6000	6000	6987
Client-6	6000	6000	5013
Client-7	6000	6000	7865
Client-8	6000	6000	4135
Client-9	6000	6000	8817
Client-10	6000	6000	3183

Table 1: Data distribution for MNIST

Client Number	IID	NON-IID Balanced	NON-IID Unbalanced
Client-1	5000	5000	1322
Client-2	5000	5000	8678
Client-3	5000	5000	2572
Client-4	5000	5000	7428
Client-5	5000	5000	5513
Client-6	5000	5000	4487
Client-7	5000	5000	2042
Client-8	5000	5000	7958
Client-9	5000	5000	5783
Client-10	5000	5000	4217

Table 2: Data distribution for CIFAR-10

The input of training and testing datasets were prepared so that neural network model could use them. Thus, pixel value of images was normalized by divided by 255 because the range of a pixel is 0 to 255. Each pixel value was ranged between 0 and 1 after this process. Besides, they were reshaped to (#instance, pixel, pixel, channel) form so that the model can take them and start training.

For example, for CIFAR-IID, each client has (5000,32,32,3) shaped input data. In addition to input data, output data for both datasets were one hot encoded because Keras models requires output datasets in that form.

Part 3: Neural Network Model & Training

I decided to use Image Classification task because it has been used widely in the academia and real-world applications. I needed small Convolutional Neural Network that works significantly well in my datasets to test federated learning. The model also had to be trained quickly due to my computer's limited resources, so the model had to have a smaller number of trainable weights. Thus, I chose Lenet-5 [3] as the model. Lenet-5 is a CNN model that consists of 7 layers. There are 1 input layer, 2 convolutional layers, 2 subsampling layers and 3 fully connected layers. Firstly, the input layer takes an image from training data. Then, convolutional layer extracts the feature map from given input and pass feature map to the subsampling layer. In my project, I used max pooling as subsampling layer. Max pooling basically takes maximum value in given pool size. Then, there is a one more convolutional layer and max pooling process. After that, last output is flattened into 1 dimension array. Finally, fully connection layers do matrix multiplication and get the final output which is possibility of classes. The visualization of model pipeline can be seen in *Figure 4*.

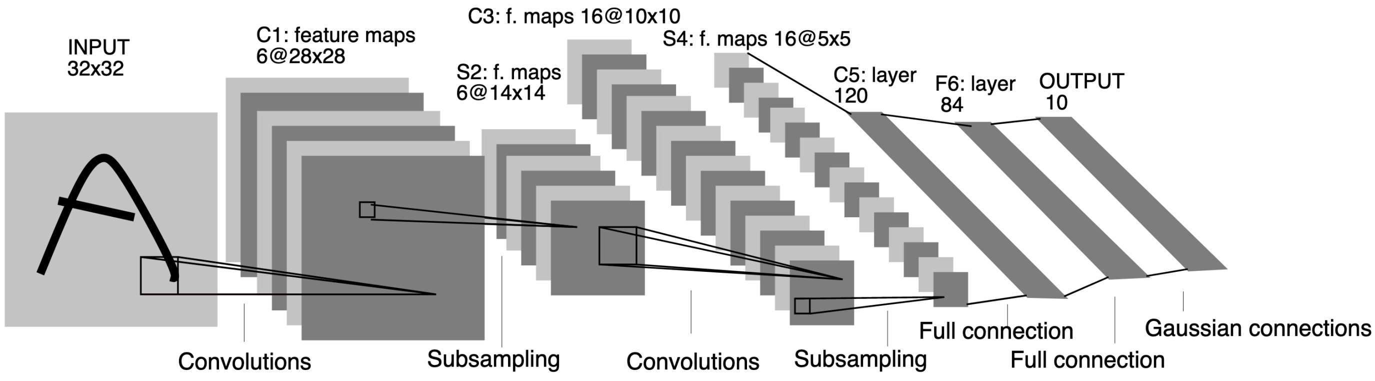

Figure 4

The pipeline was generated with Sequential function of Keras API. Keras has built-in layer implementations, so I didn't have to implement all these layers myself. I created these layers by regarding functions and add them in Sequential function as parameters.

The parameter in the [3] worked well for MNIST; nevertheless, poorly for CIFAR-10 dataset so I did slight changes in layer parameters to get better results for CIFAR-10. Then, I used these layer parameters in both CIFAR-10 and MNIST datasets. New parameters are shown in Table-3.

Layer Name	**Feature Map**	**Kernel/Filter or Units**	**Activation**
Convolution 1	30	3x3	Relu
Max pool 1	-	2x2	
Convolution 2	13	3x3	Relu
Max pool 2	-	2x2	
Fully connected 1	120	-	Relu
Fully connected 2	86	-	Relu
Fully connected 3	10	-	Softmax

Table 3

Financial Feasibility

Currently, I’m using my personal computer without GPU but I’m planning to use more complex and large-scale neural network models and dataset such as CIFAR-100 and Resnet-50 in next semester. So, it can be beneficial to use cloud machine with gpu. Also, I may need to use more clients in next semester so I may need huge RAM capacity. Financial feasibility must be reconsidered depends on the machine in cloud.

Technical Feasibility

Since federated learning is still prototype concept, there are miserable resources that I can benefit for my project, so I have spent too much time to find what I have looked for.

Knowledge and Skill Set

- CS101: Since it is the core of all computer science lectures. I highly utilize this lecture in my project.
- CS102: Object oriented programming components are used all over the project
- CS320: I have learnt how to specify software requirements in this lecture.
- CS321: Functional programming concepts are utilized all over the project.
- CS447: I have learnt network protocols from this lecture. It helped me to understand better how WebSocket’s work.
- CS452: I have learnt how to use validation metrics and visualization tools.
- CS454: I have learnt how neural networks work, it’s concepts and literature behind them and how to construct, train and validate convolutional neural networks.

V. Results and Discussion

The federated learning setting application was tested with MNIST and CIFAR-10 data. There are 10 clients in all cases and batch size is set as 10 for all trainings for fairness [2]. In the first test, one client with the whole dataset (60.000 instances for MNIST and 50.000 for CIFAR) trained the model over 10 epochs, which is called as the *baseline*. This *baseline* is maintained as reference of comparison with federated model accuracies. In *baseline* training, there are 60.000 mini-batch updates for MNIST and 50.000 for CIFAR. Next, IID data were distributed to 10 clients. Each client trained their local data over 10 epochs. Number of communication rounds is set to 10, thus server aggregated models from clients 10 times. Note that, data are not sent from client(s) to server in this federated learning approach, so the network bandwidth saved with respect to a non-federated learning approach is equal to the model size (249 kb) (size of CIFAR-10 = 550 mb and MNIST = 110 mb). When the multiple communication rounds are considered the data transfer savings in real life scenario would be immense. Each client synchronously made 6000 and 5000 mini-batch updates for MNIST and CIFAR respectively in a communication round. Totally 60.000 and 50.000 minibatch updates for 10 communication rounds per client were made. Results are shown in Figure 5 and Figure-6.

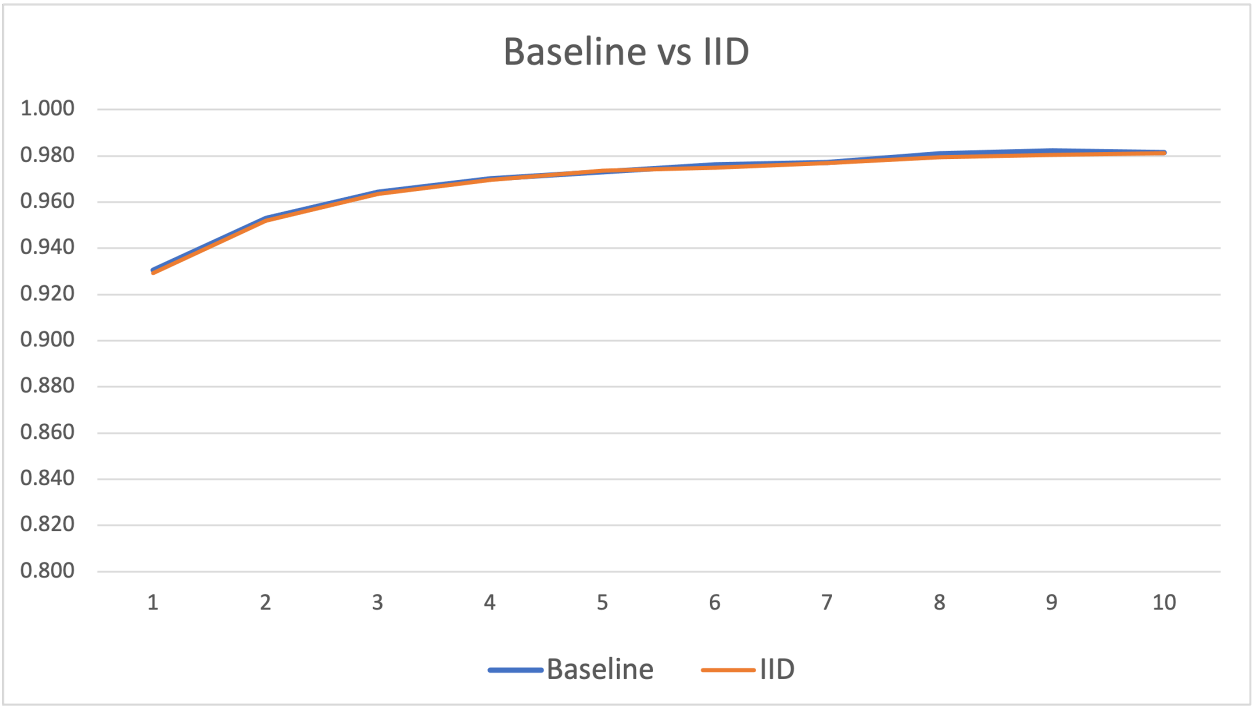

Figure 5: Baseline vs IID by MNIST

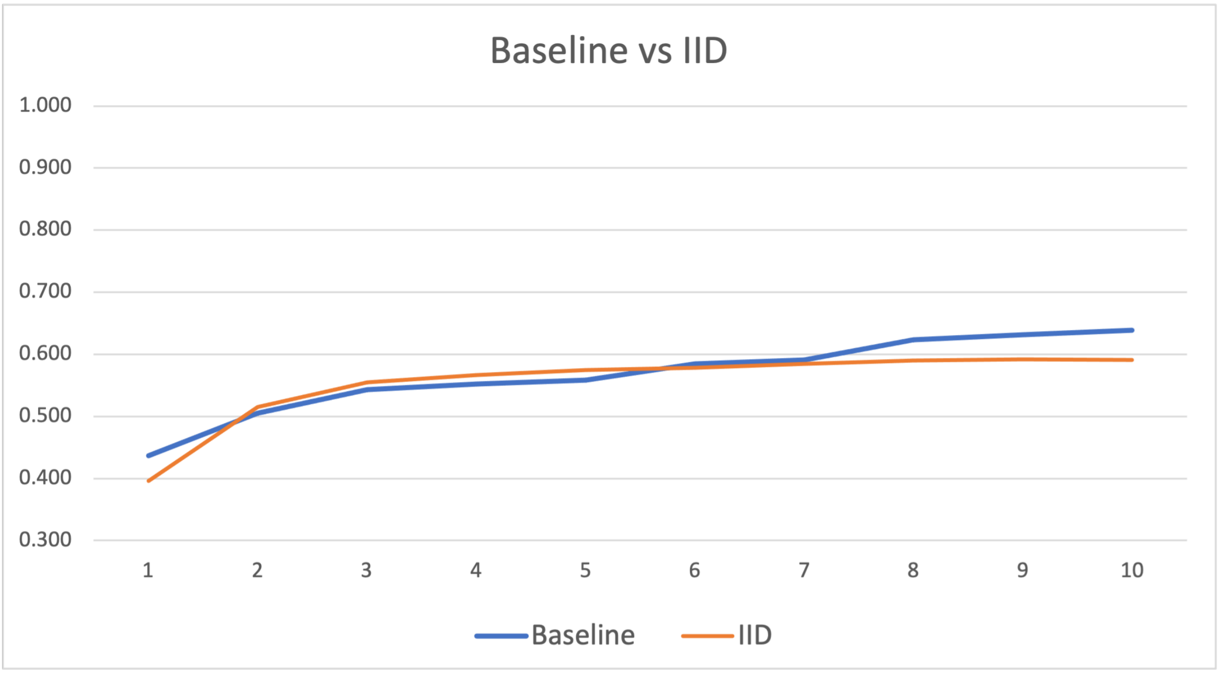

Figure 6: Baseline vs IID by CIFAR-10

It can be concluded that baseline and IID have similar accuracy during training period. Thus, when the data is balanced and distributed as IID, federated learning works with slight accuracy loss. In Figure 5, results are almost identical by MNIST data, but the accuracy difference in Figure 6 is bigger than Figure 5. In the next case, Non-IID data is separated to clients by balance and unbalanced types. Each client trained it with 10 local epochs and communication round was again 10.

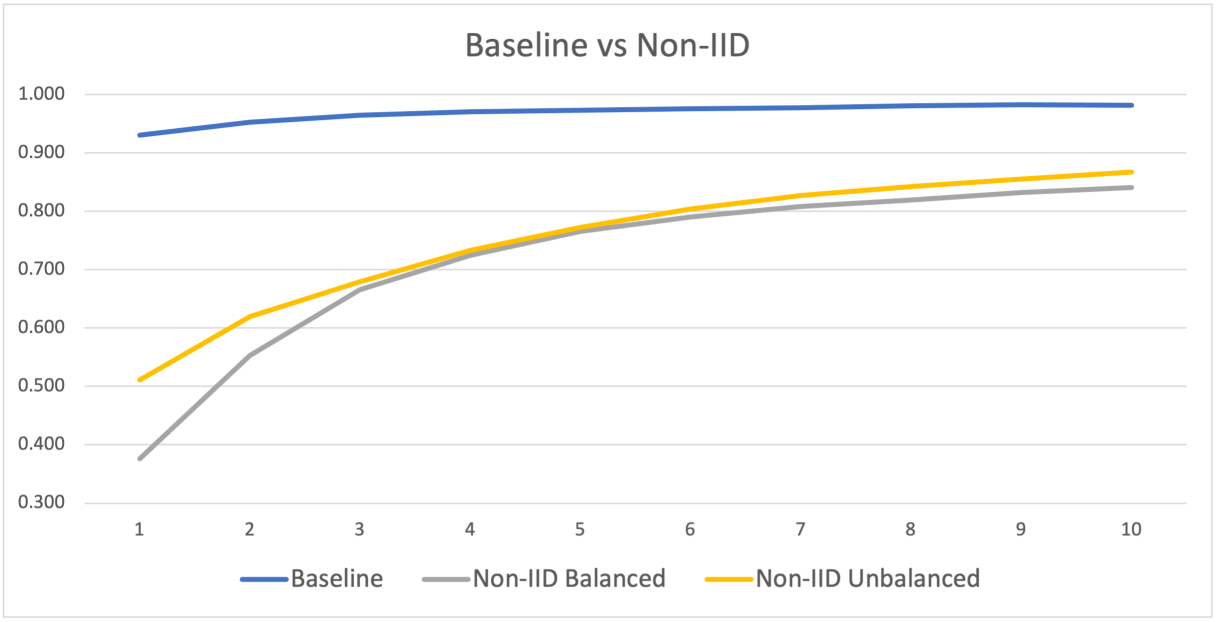

Figure 7: Baseline vs Non-IID by MNIST

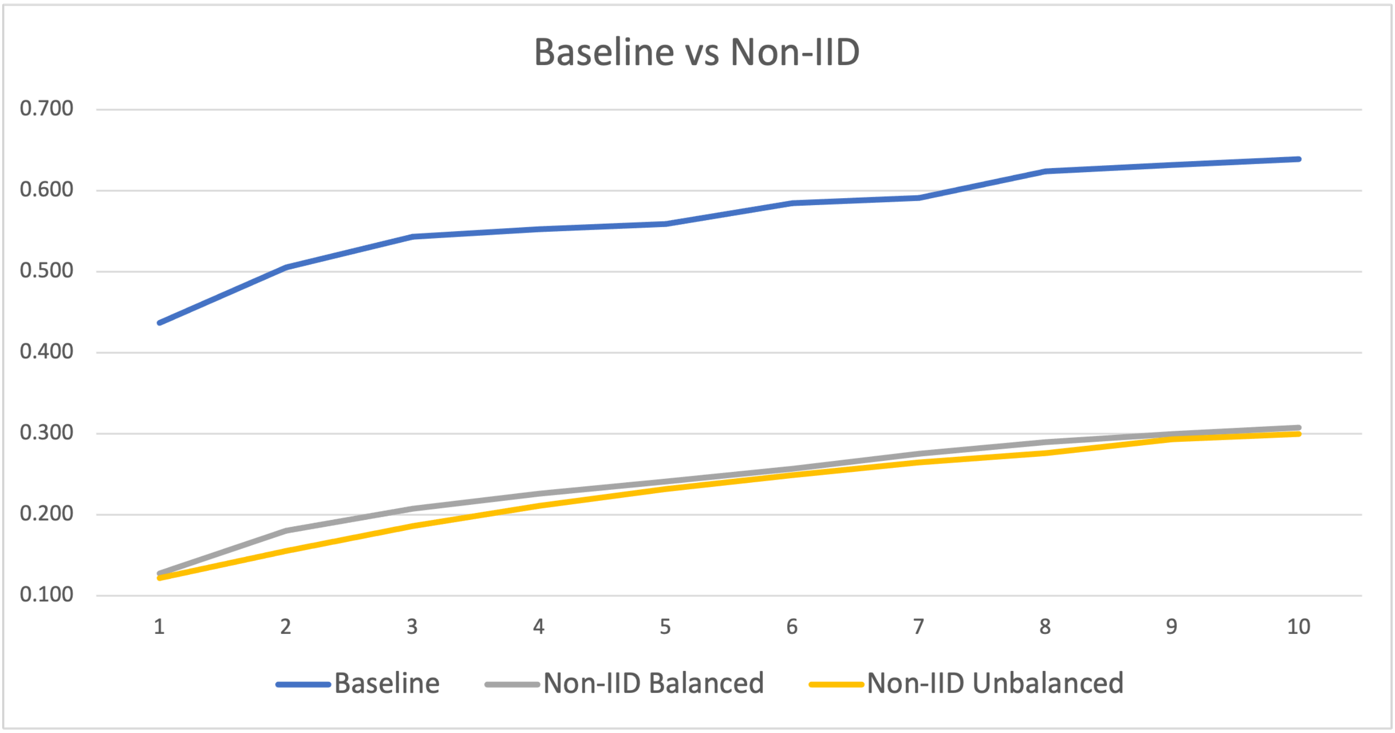

Figure 8: Baseline vs Non-IID by CIFAR-10

In Figure 7 and 8, it is clear that Non-IID performs poorly compared to IID. Because each client has only two labels, but clients still continue to learn between them. Also, unbalanced and balanced performs mostly identical. So, amount of data doesn't cause a huge problem. In next case, Non-IID again is used with 10 communication rounds; however, this time number of local epochs is set 20 instead of prior 10.

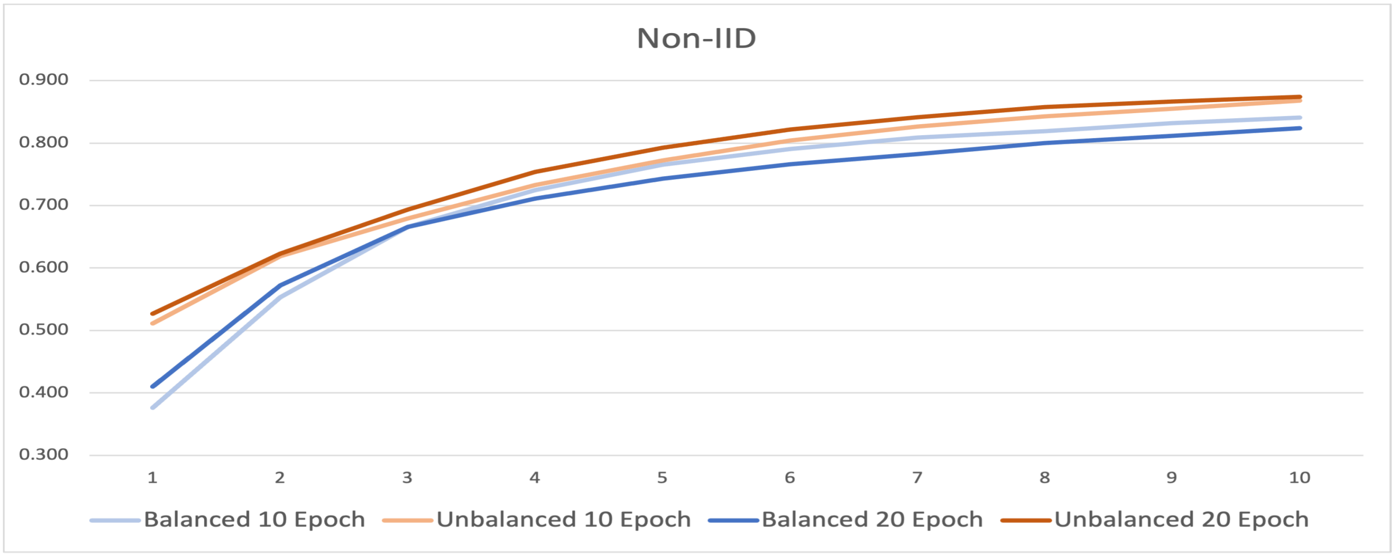

Figure 9: Non-IID by MNIST

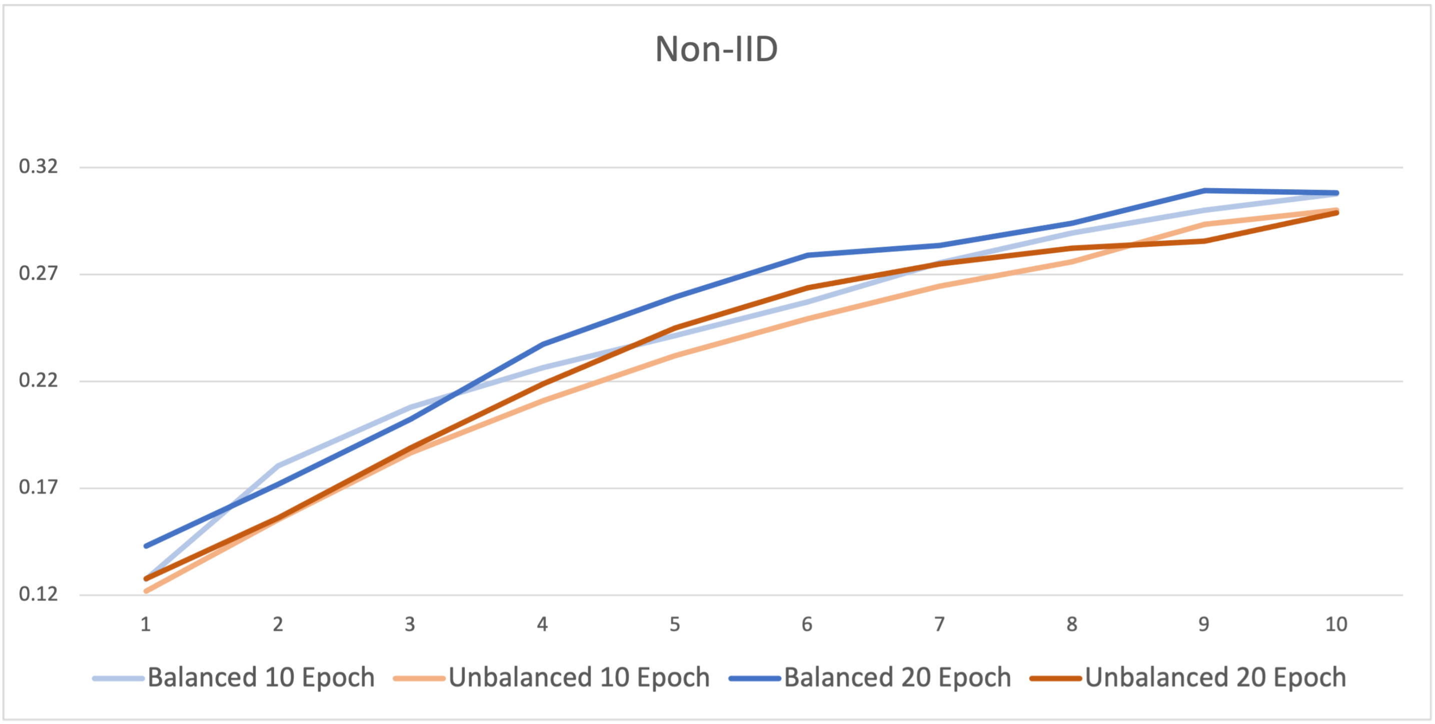

Figure 10: Non-IID by CIFAR-10

In Figure 9 and 10, bold lines represent results of 20 epoch trained models and blue colored lines represents balanced, red colored ones for unbalanced. In MNIST, unbalanced works better however works poorly in CIFAR-10. 20 epoch trained models works similar to 10 epoch trained models. There is no big accuracy difference between them. In last case, 20 local epoch is replaced with 10 local epochs back and 10 communication round is updated to 20. In Figure 11-12, it is clear to say that bigger communication round number rise the accuracy. There is a huge gap between 10 CR and 20 CR in Figure 11, but this gap is less in Figure 12.

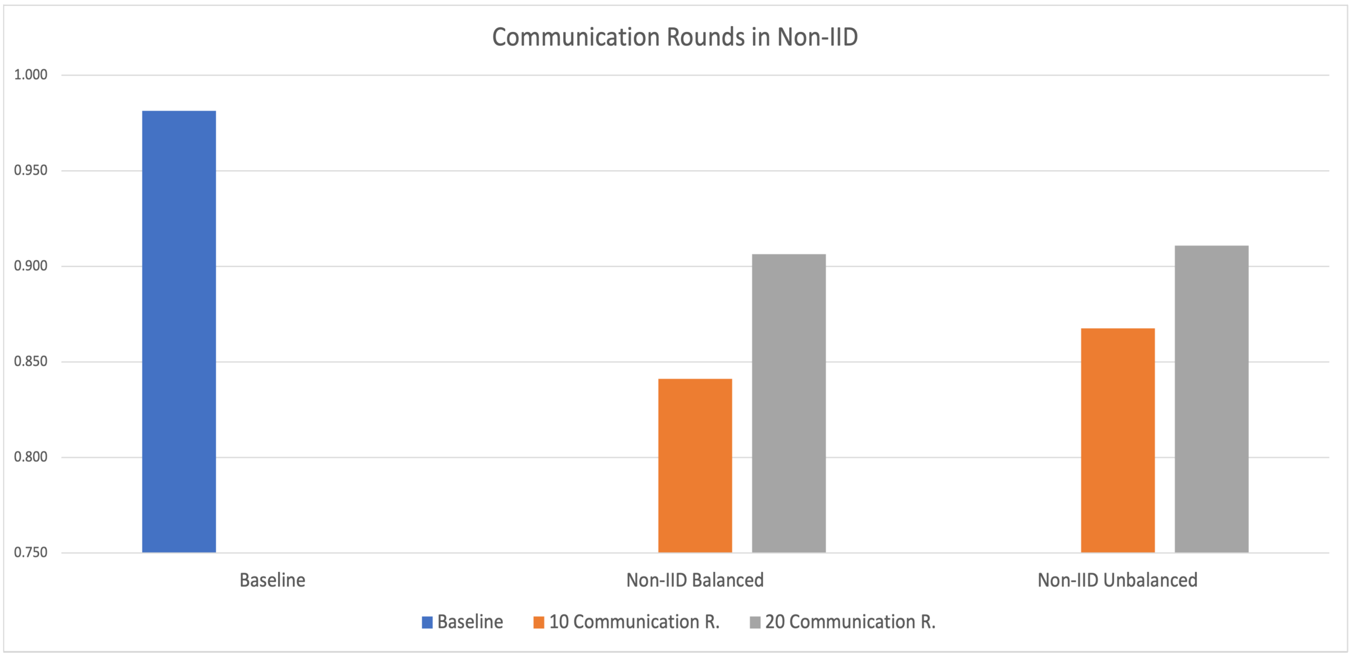

Figure 11: Communication Round Comparison by MNIST

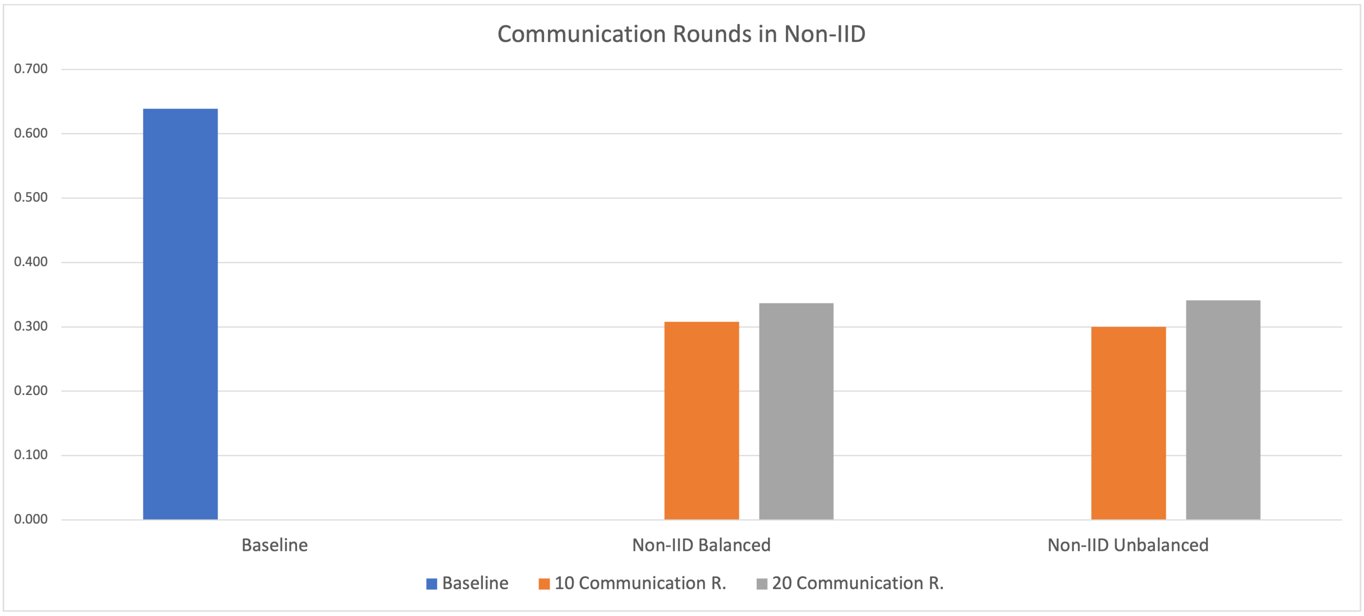

Figure 12: Communication Round Comparison by CIFAR-10

VI. Related Work

Since Federated Learning is new concept, new techniques are being introduced almost every day. However, there is a paper [2] that is considered fundamental paper by federated learning researchers. In this paper, FedAvg, an aggregation technique, is used as baseline for many papers. That's why I chose this paper as my baseline and used FedAvg and also datasets that were presented in this paper. Besides, [5] studied how decentralized federated learning is used in blockchain via smart contracts. After federated learning among clients in decentralized platform, they used Ethereum smart contract to store the global model, so they prevent single point of failure. [6] introduced federated learning algorithm for client personalization. Clients only send their base layers instead of whole model to the server because base layers are more general and top layers are more personalized layers. In server, base layers are aggregated and send back to clients. This allows clients to personalize model to its own behavior. However, this lets server not to do any decision making due to lack of layers. [7] introduced new aggregation technique which is called FedDist that performs better than FedAvg. FedDist is a coordinate-wise approach that gets rid of diverge weights and add new neuron if threshold is reached. [8] published new datasets to test federated learning baseline. The new datasets are used widely by federated learning researchers.

VII. Conclusion and Future Work

In this project, I created an application to measure how federated learning works with different scenarios. To implement the application, I have used python language and WebSocket protocols in network layer to provide bidirectional communication between a server and clients. I split two different datasets into regarding scenarios which are IID, Non-IID with balanced and unbalanced. Moreover, I created a convolutional neural network to train data and optimize parameters. Briefly, it can be said that neural network model selection for initially, number of communication rounds and complexity of dataset play huge role in performance of federated learning and these parameters are needed to be optimized. Specifically, having an IID vs non-IID classes dataset in federated clients is more crucial than having a balanced vs unbalanced count (corpus size) representation.

Federated learning helps people to own their personal data securely. Moreover, it encourages to use all event-free devices to be used. By doing that, it can reduce to redundant GPU sale in the world. Even power of blockchain, people may earn money while being part of federated learning with their free devices. Federated learning can be enormously efficient for usage in hospitals since patient information cannot be shared among hospitals due to privacy. However, machine learning models in healthcare can be shared among hospitals to acquire better model [9].

In next semester, I will convey my application from centralized to decentralized system because centralized application has single point of failure issue; however, decentralized system can solve it. I have already implemented many codes for decentralized system with Golang by using libp2p which is a framework for p2p systems. Also, I will search the interaction with smart contracts in blockchain system to store global model efficiently as it is mentioned in [5]. Besides, I will use new datasets not only for image classification but also for NLP application. For example, there is a Shakespeare dataset [8] that is used for next character prediction, that might be realtime use case of federated learning. Furthermore, I will use tests with more clients with a powerful computer than my personal computer. I will use more powerful aggregation algorithm than FedAvg.

VIII. Acknowledgements

I am deeply thankful to my supervisor İsmail Arı for his support and advice during the semester.

IX. References

[1] Zhu, H., Xu, J., Liu, S., & Jin, Y. (2021). Federated Learning on Non-IID Data: A Survey. *arXiv preprint arXiv:2106.06843*.

[2] McMahan, B., Moore, E., Ramage, D., Hampson, S., & y Arcas, B. A. (2017, April). Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics* (pp. 1273-1282). PMLR.

[3] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324.

[4] Krizhevsky, A., & Hinton, G. (2009). Learning multiple layers of features from tiny images

[5] Korkmaz, C., Kocas, H. E., Uysal, A., Masry, A., Ozkasap, O., & Akgun, B. (2020, November). Chain fl: Decentralized federated machine learning via blockchain. In *2020 Second International Conference on Blockchain Computing and Applications (BCCA)* (pp. 140-146). IEEE.

[6] Arivazhagan, M. G., Aggarwal, V., Singh, A. K., & Choudhary, S. (2019). Federated learning with personalization layers. *arXiv preprint arXiv:1912.00818*.

[7] Ek, S., Portet, F., Lalanda, P., & Vega, G. (2021, March). A federated learning aggregation algorithm for pervasive computing: Evaluation and comparison. In *19th IEEE International Conference on Pervasive Computing and Communications PerCom 2021*.

[8] Caldas, S., Duddu, S. M. K., Wu, P., Li, T., Konečný, J., McMahan, H. B., ... & Talwalkar, A. (2018). Leaf: A benchmark for federated settings. *arXiv preprint arXiv:1812.01097*.

[9] Nguyen, D. C., Pham, Q. V., Pathirana, P. N., Ding, M., Seneviratne, A., Lin, Z., ... & Hwang, W. J. (2021). Federated Learning for Smart Healthcare: A Survey. *arXiv preprint arXiv:2111.08834*.

Appendix

A. *Server Implementation*

```python
class Server:
    def __init__(self, req_nodes=10, comunication_rounds=20):
        self.sio = socketio.AsyncServer(async_mode="aiohttp",ping_timeout=120)
        self.app = web.Application()
        self.sio.attach(self.app)
        self.register_handles()

        self.connected_nodes = list()
        self.pending_nodes = list()
        self.req_nodes = req_nodes
        self.training_room = "training_room"
        self.average_weights = dict()

        _, _, self.X_test, self.y_test = get_data('server','mnist','non_iid','unbalanced')
        self.global_model = get_model_mnist()
        self.max_rounds = comunication_rounds
        self.round = 0
        self.pool = ThreadPoolExecutor(max_workers=4)

    def register_handles(self):
        self.sio.on("connect", self.connect)
        self.sio.on("fl_update", self.fl_update)

    async def connect(self, sid, environ):
        self.connected_nodes.append(sid)
        self.sio.enter_room(sid, self.training_room)

        async def start_training_callback():
            if len(self.connected_nodes) == self.req_nodes:
                print("Connected to ", self.req_nodes, " starting training")
                await self.start_round()
            else:
                print("Waiting to connect to ", self.req_nodes - len(self.connected_nodes),
                      " more nodes to start training")

        await self.sio.emit(
            "connection_received",
            room=sid,
            callback=start_training_callback,
        )

    def run_server(self, host="0.0.0.0", port=5000):
        web.run_app(self.app, host=host, port=port)

    def evaluate(self):
        y_pred = self.global_model.predict(self.X_test)
        print('Accuracy: ',accuracy_score(self.y_test,np.argmax(y_pred, axis=1)))
```

```python
async def fl_update(self, sid, data):
    for layer in data.keys():
        temp_weight = decode(data[layer])
        if len(self.average_weights[layer]) == 2 :
            self.average_weights[layer][0] += (temp_weight[0] / len(self.connected_nodes))
            self.average_weights[layer][1] += (temp_weight[1] / len(self.connected_nodes))
        else:
            self.average_weights[layer].append(temp_weight[0] / len(self.connected_nodes))
            self.average_weights[layer].append(temp_weight[1] / len(self.connected_nodes))
    self.pending_nodes.remove(sid)
    if not self.pending_nodes:
        loop = asyncio.get_event_loop()
        asyncio.ensure_future(self.async_consume(loop))

def apply_updates(self):
    print("Applying updates to global model")
    for layer in self.global_model.layers:
        if layer.trainable_weights:
            layer.set_weights(self.average_weights[layer.name])
    self.evaluate()

def async_consume(self, loop):
    yield from loop.run_in_executor(self.pool, self.apply_updates)
    loop.create_task(self.end_round())

async def start_round(self):
    print(f'Starting round {self.round+1}')
    self.pending_nodes = self.connected_nodes.copy()
    for layer in self.global_model.layers:
        if layer.trainable_weights:
            self.average_weights[layer.name] = []
    await self.sio.emit(
        "start_training",
        data={
            "model_architecture": self.global_model.to_json(),
            "model_weights": encode_layer(self.global_model.get_weights()),
        },
        room=self.training_room,
    )

async def end_round(self):
    print("Ending round")
    self.round += 1
    if self.round < self.max_rounds:
        await self.start_round()
    else:
        await self.end_session()
```

```python
async def end_session(self):
    print("Ending session")
    await self.sio.emit(
        "end_session",
        room=self.training_room,
        data={
            "model_weights": encode_layer(self.global_model.get_weights()),
        },
    )

async def disconnect(self, sid):
    self.connected_nodes.remove(sid)
    self.sio.leave_room(sid, room=self.training_room)
```

B. *Client Implementation*

```python
class Client:
    def __init__(self, address, client, epochs=10):
        self.client = client
        self.server = address
        self.sio = socketio.Client()
        self.register_handles()

        self.X_train, self.y_train, self.X_test, self.y_test = get_data(self.client,'mnist','non_iid','unbalanced')
        self.model = None
        self.epochs = epochs

    def connect(self):
        self.sio.connect(url=self.server)

    def register_handles(self):
        self.sio.on("connection_received", self.connection_received)
        self.sio.on("start_training", self.start_training)

    def connection_received(self):
        print(f"Server at {self.server} returned success")

    def start_training(self, global_model):
        self.model = model_from_json(global_model["model_architecture"])
        self.model.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accuracy'])
        self.model.set_weights(decode(global_model["model_weights"]))
        print("Starting training")
        self.model.fit(self.X_train,self.y_train,epochs=self.epochs,batch_size=10)
        y_pred = self.model.predict(self.X_test)
        print('Accuracy: ',accuracy_score(self.y_test,np.argmax(y_pred, axis=1)))
        print(confusion_matrix(self.y_test,  np.argmax(y_pred, axis=1)))
        self.send_updates()

    def send_updates(self):
        model_weights = dict()
        for layer in self.model.layers:
            if layer.trainable_weights:
                model_weights[layer.name] = encode_layer(layer.get_weights())

        self.sio.emit("fl_update",data=model_weights)

    def disconnect(self):
        return

    def end_session(self, data):
        model_weights = decode(data['model_weights'])
        self.model.set_weights(model_weights)
```

C. *Model Creation*

```python
def get_model_general(data):
    pixel = 32
    rgb = 3
    if data == 'mnist':
        pixel=28
        rgb=1
    os.environ['TF_DETERMINISTIC_OPS'] = '1'
    SEED = 1
    random.seed(SEED)
    np.random.seed(SEED)
    tf.random.set_seed(SEED)
    model = Sequential()
    model.add(Conv2D(6, kernel_size=5, strides=1,
        activation='tanh',input_shape=(pixel, pixel, rgb)))
    model.add(AveragePooling2D())
    model.add(Conv2D(16, kernel_size=5, strides=1,  activation='tanh'))
    model.add(AveragePooling2D())
    model.add(Flatten())
    model.add(Dense(120, activation='tanh'))
    model.add(Dense(84, activation='tanh'))
    model.add(Dense(10, activation='softmax'))
    return model
```

D. *Data Preperation*

```
data = 'mnist'
X = pd.read_csv(f'../data/{data}/train.csv',index_col=0)
y = X['label']
X.drop('label',inplace=True,axis=1)

#iid_balanced
rand_array = np.array(y.index)
np.random.shuffle(rand_array)
clients = [[] for i in range(10)]
for i in range(10):
    clients[i] = list(rand_array[i*5000:(i+1)*5000])
save_as_csv(clients,X,y,data,"iid_balanced")

#non-iid_balanced
first = []
second = []
temp_array = np.array_split(y.sort_values(),20)
for i in range(len(temp_array)):
    if i % 2 == 0:
        first.append(temp_array[i])
    else:
        second.append(temp_array[i])

index_first = 0
index_second = 0
clients = [[] for i in range(10)]
for i in range(10):
    if i % 2 == 0:
        print(set(np.concatenate((first[index_first], first[index_first+1]), axis=0)))
        clients[i] = list(np.concatenate((first[index_first].index, first[index_first+1].index), axis=0))
        index_first+=2
    else:
        print(set(np.concatenate((second[index_second], second[index_second+1]), axis=0)))
        clients[i] = list(np.concatenate((second[index_second].index, second[index_second+1].index), axis=0))
        index_second+=2
save_as_csv(clients,X,y,data,"non_iid_balanced")


#non-iid unbalanced
temp_array = np.array_split(y.sort_values(),10)
clients = [[] for i in range(10)]
for i in range(0,10,2):
    cut = np.random.randint(y.shape[0] / 10, size=(1, 1))[0][0]
    cut2 = np.random.randint(y.shape[0] / 10, size=(1, 1))[0][0]
    clients[i]=list(np.concatenate(((temp_array[i][:cut]).index , (temp_array[i+1][:cut2]).index), axis=0))
    clients[i+1]=list(np.concatenate((((temp_array[i][cut:]).index , (temp_array[i+1][cut2:]).index), axis=0))
save_as_csv(clients,X,y,data,"non_iid_unbalanced")


def save_as_csv(clients,X,y,data,type):
    for i in range(10):
        print(len(clients[i]))
        y[clients[i]].reset_index(drop=True).to_csv(f'../data/{data}/{type}/client'+str(i+1)+'_y_train.csv')
        X.iloc[clients[i]].reset_index(drop=True).to_csv(f'../data/{data}/{type}/client'+str(i+1)+'_X_train.csv')
```