SplitNN for Vertically Partitioned Data

A Project Proposal for
The Google Summer of Code

From Abbas Ismail

Mentor:- Adam J Hall

OpenMined

Contents

1. Project Proposal

1.1 General Aspects

In Machine Learning projects, companies need data from other partners to train their models. The model architecture and parameters represent a company asset while the data is the partner's asset. Data privacy is a large challenge for machine learning. The techniques such as Federated Learning and Split Neural Network aim to distribute the training of a neural network across multiple machines for privacy preservation.

Split Neural Network is a technique in which model architecture is distributed among different entities or workers. It only communicates activations and gradients just from the split layer, unlike other methods that share weights/gradients from all the layers. SplitNN attains high resource efficiency for distributed deep learning models.

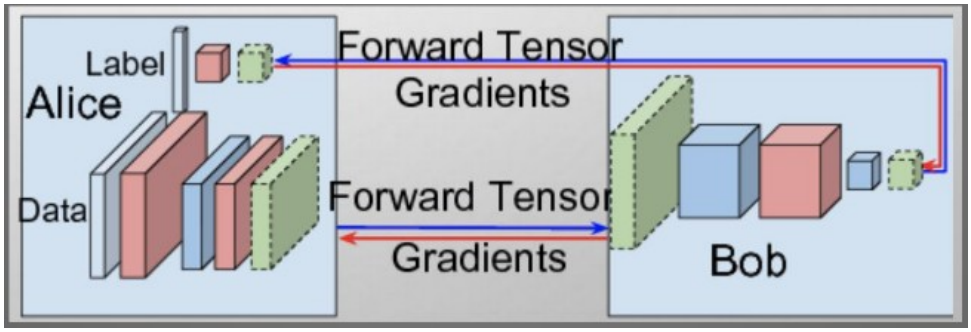

Figure 1: Split Neural Network

1.2 Aim/Objective

This project aims to implement SplitNN configuration for vertically partitioned data. Data is said to be vertically partitioned when several organizations own different attributes or modalities of information for the same set of entities. Partition allows for orgnizations holding different modalities of data to learn distributed models without data sharing. Partitioning scheme is traditionally used to reduce the size of data by splitting and distribute to each client

As of now, in the split_neural_network, there are implementations of SplitNN architectures for the data held by the single entity. During the period of Google Summer of Code, I will be creating the notebooks (in the advanced tutorials section) of implementations for SplitNN for vertically partitioned data with Federated Learning scheme.

1.3 Proposed Methodology

Firstly, we must define the structure of the set of pointers which points towards the set of training data batches at different data holder's locations.

Let say, we have three data holders:- Alice, Bob, Claire.
Each of them has their data managed into a list of batches of the same size.

```
Alice_data->  [alice_batch1, alice_batch2, alice_batch3,...]
Bob_data->    [bob_batch1, bob_batch2, bob_batch3,...]
Claire_data-> [claire_batch1, claire_batch2, claire_batch3,...]
```

At the central server, the pointers to the list of data batches of each client/data holder are structured into a dictionary (say Data_pointer).

"Data_pointer" is a dictionary in which every,
(key, value) = (id of the data holder, a pointer to the list of batches at that data holder) .

```
Data_pointer-> {
                  "alice" : alice_data_ptr,
                  "bob" :  bob_data_ptr,
                  "claire": claire_data_ptr
                }
```

alice_data_ptr :- pointer at the central server, which points to "Alice_data".
bob_data_ptr :- pointer at the central server, which points to "Bob_data".
claire_data_ptr :- pointer at the central server, which points to "Claire_data".

In every training iteration, a batch from all data holders/clients is passed through the model parallelly in a federated manner. The gradients are aggregated and model parameters are updated as per the configurations explained below.

1.4 SplitNN configurations

In this section, several configurations of SplitNN are proposed for vertically partitioned data.

Configuration-I:-

This configuration allows for multiple institutions holding different modalities of patient data to learn distributed models without data sharing. In Fig. 3, showing an example configuration of SplitNN suitable for such multi-modal multi-institutional collaboration. In this configuration, the Server holds the labels for Input Data from each client.

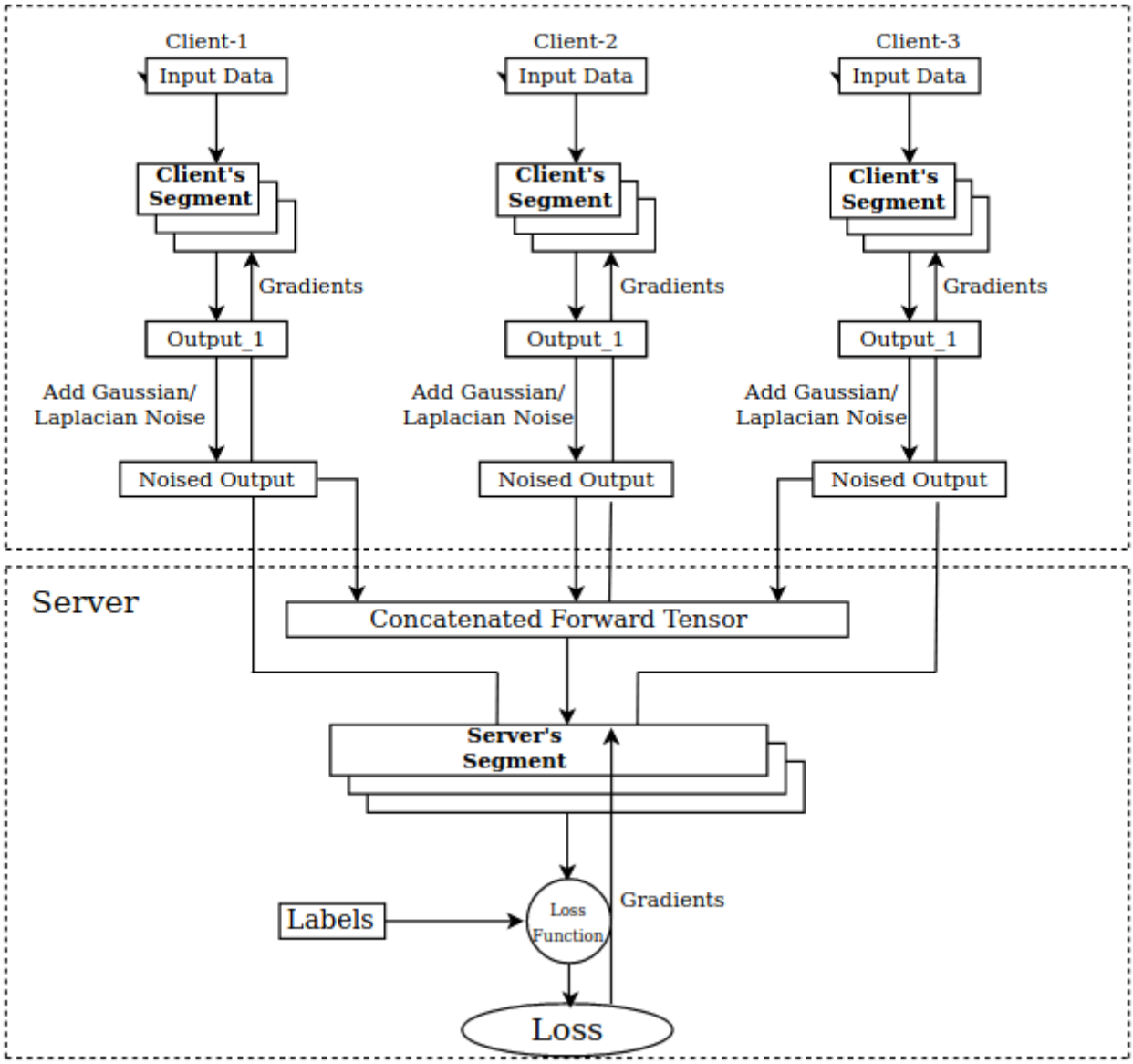

Figure 3a: Split learning for vertically partitioned data

- In this method, each client trains the network upto a certain layer known as the cut layer and sends the output to the server after adding noise to it.
- The server then concatenates the output from all the clients and pass it through the rest of the layers. This completes the forward propagation.
- The server then generates the gradients for the final layer and back-propagates the error until the cut layer. The gradients are then passed over to the client(s).

- The rest of the back-propagation is completed by the clients. This is continued until the network is trained. In this configuration, the Server holds the labels for Input Data from each client.

Advantages :-

- Simplest SplitNN configuration.
- Easy gradient transfer for back-propagation.
- No compromise with the accuracy.

Disadvantages:-

- Label sharing by the clients.
- Intermediate layer's output is shared.

Configuration-II:-

The configuration-1 involves the sharing of labels although they do not share the raw input data. This can be avoided using a U-shaped configuration that does not requires the label sharing by clients. This configurations contains three model segments which can be further extended.

1. Front Segment(Client)
2. Centre Segment(Server)
3. Back Segment(Client)

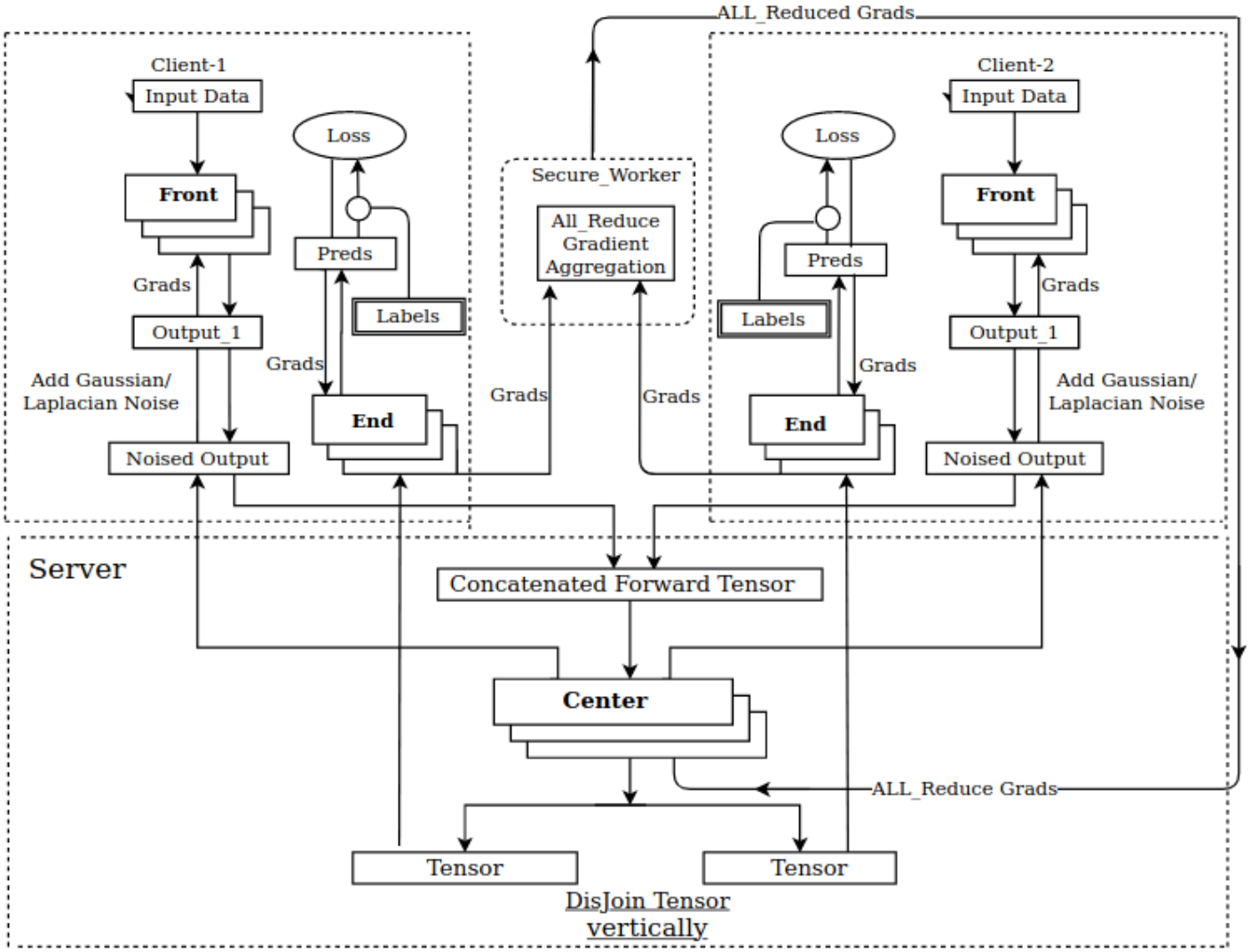

Figure 3b:U-shaped SplitNN for vertically partitioned data

➢ In this configuration, each client trains the network up to a certain layer or the **Front Segment**. These are the starting layers of the network.

➢ The server then concatenates the output from all the clients an pass it through the **Center Segment** (which is the majority layers of the neural network) of the neural network.

➢ The output from the Center Segment of the network goes through a **disjoin operation vertically** concerning each client's input data.

➢ The tensor concerning each client's input is then sent to the client's location. This tensor is then pass through the **End Segment** (containing end layers of the neural network) of the model which in turn generates prediction and calculates the loss.

➢ The client generates the gradients from the end layers and uses them for backpropagation without sharing corresponding labels.

➢ The gradients obtain from each client are sent to a **Secure_Worker** which calculates the **weighted average of the gradients** from different client.

All_Reduce Gradient Aggregation :-

$$\begin{aligned}\frac{\partial \text{Loss}}{\partial w} &= \frac{\partial \left[\frac{1}{n}\sum_{i=1}^{n} f(x_i, y_i)\right]}{\partial w} \\ &= \frac{1}{n}\sum_{i=1}^{n}\frac{\partial f(x_i, y_i)}{\partial w} \\ &= \frac{m_1}{n}\frac{\partial \left[\frac{1}{m_1}\sum_{i=1}^{m_1} f(x_i, y_i)\right]}{\partial w} + \frac{m_2}{n}\frac{\partial \left[\frac{1}{m_2}\sum_{i=m_1+1}^{m_1+m_2} f(x_i, y_i)\right]}{\partial w} + \cdots + \frac{m_k}{n}\frac{\partial \left[\frac{1}{m_k}\sum_{i=m_{k-1}+1}^{m_{k-1}+m_k} f(x_i, y_i)\right]}{\partial w} \\ &= \frac{m_1}{n}\frac{\partial l_1}{\partial w} + \frac{m_2}{n}\frac{\partial l_2}{\partial w} + \cdots + \frac{m_k}{n}\frac{\partial l_k}{\partial w}\end{aligned}$$

where,

w is the parameters of the model,

n is the total number of data points in the dataset,

$\mathbb{k}$ is the total number of clients,

$\partial \mathbb{L}oss/\partial w$ is the true gradient of the big batch of size n,

$\partial l_{\mathbb{k}}/\partial w$ is the gradient of a batch at client "$\mathbb{k}$",

xi and yi are the features and labels of data point i,

$m_{\mathbb{k}}$ the number of data points at client $\mathbb{k}$, ($m1+m2+\cdots+m\mathbb{k} = n$)

Advantages :-

- Label sharing not involved.
- No compromise with the accuracy.

Disadvantages:-

- Complex gradient transfer process.
- An additional worker is involved (Secure_Worker).
- Intermediate layer's output is shared.

Configuration-III:-

This configuration is a form of *standard federated learning scheme*. The neural network is divided into different segments and these segments are sent to each client.

- The model at the server is divided into different segments. These segments are moved one by one to each client.
- At the central server, the pointers to the list of input and output of of each model segment at every client/data holder are structured into two dictionaries (say input_ptr and output_ptr).

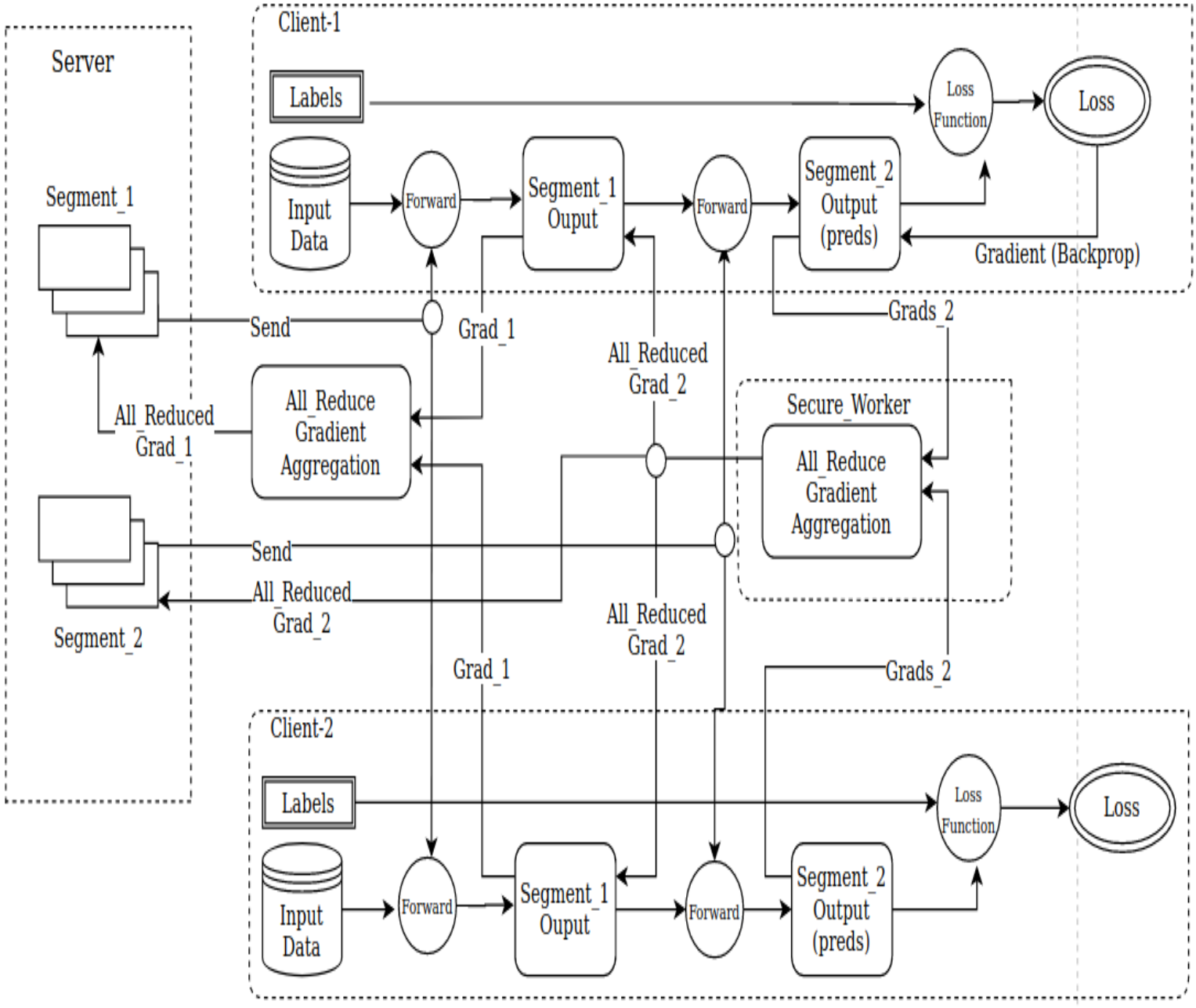

Figure 3c: SplitNN for standard federated learning for vertically partitioned data

- **”Input_ptr” is a dictionary in which every,**
  (key, value) = (id of the data holder, pointer to the list of inputs for each segment at that data holder)

Input_pointer-> {
“alice” : [input for segment 1 at alice, input for segment 2 at alice],
“bob” : [input for segment 1 at bob, input for segment 2 at bob].
“claire” : [input for segment 1 at claire, input for segment 2 at claire],
}

”Output_ptr” is a dictionary in which every,
(key, value) = (id of the data holder, pointer to the list of outputs of each segment at that data holder)

Output_pointer-> {
“alice” : [Output of segment 1 at alice, Output of segment 2 at alice],
“bob” : [Output of segment 1 at bob, Output of segment 2 at bob],
“claire” : [Output of segment 1 at claire, Output of segment 2 at claire],
}

- Forward Propagation

```
#iterating above procedure for each model segment from segment no.0
for idx in range(0, len(self.models)):

    #-->ith forward pass for each dataholder's batch
    for owner in self.data_owners:

        #-->copy the last output to current input and detaching with it.
        current_segment_input = self.output_ptr[owner.id][idx-1].detach().requires_grad_()
        self.input_ptr[owner.id].append(current_segment_input)

        #--->sending model to each data_holder(alice, bob, claire)
        #-->[Fedederated Learning]
        self.models[idx].move(owner)

        #-->forward pass and append to the output_ptr list of the current owner
        current_segment_output = self.models[idx](self.input_ptr[owner.id][idx])
        self.output_ptr[owner.id].append(current_segment_output)
```

➢ Backward Propagation

```
def backward(self):

"""backpropogating for each model segment and
    copying the gradients from the next model segment's leaf"""
    for i in range(len(self.models)-2, -1, -1):

"""applying allreduce with all gradients from each model segment leaf
at local worker(or at Secure_Worker) therefore initializing it with zero"""
        grad_allreduce = 0

        #-->collecting gradients at each model segement leaf
        for owner in self.data_owners:

            grad_in = self.input_ptr[owner.id][i+1].grad.copy()

            #-->Copy the gradient and get for adding it with the grad_allreduce
            grad_allreduce += grad_in.copy().get()

        #-->normalize grads
        grad_allreduce = grad_allreduce/len(self.data_owners)

        #-->send the all reduced grad to each dataholder and
        #-->call .backward() with all_reduced grad
        for owner in self.data_owners:
            grad_new = grad_allreduce.copy().send(owner)
            self.output_ptr[owner.id][i].backward(grad_new, retain_graph=True)
```

Advantages :-

- Full data privacy is maintained as not even the intermediate outputs are not shared by the clients.

Disadvantages:-

- Complex gradient flow for back-propagation.
- Model privacy is not maintained.
- Compromise with the accuracy.

1.5 Distribution of data-sets(MNIST, CIFAR, etc.)

It is important to distribute the popular datasets among workers for the testing of the configurations explained above.

But the class for these configurations involves the dictionary inputs where each **(key, value) = (id of the data holder, a pointer to the list of batches at that data holder).**

PyTorch has **`torch.utils.data.DataLoader`** class which loads the dataset for training by converting the dataset as a pair of tensors of data and labels of a specified batch_size.

Train_Loader → [[data_batch_1, label_batch_1],
[data_batch_2, label_batch-2],
[data_batch_3, label_batch-3], ...]

"Train_Loader" can not be used for the training of SplitNN with Vertically Partitioned Data. Therefore, I will be creating a **`Distributed_DataLoader`** class which is used to distribute the datasets like MNIST, CIFAR, etc. accessible with **`torchvision.dataset`** and convert them into a list of dictionaries having a batch from all the clients as the values and id of the client as the keys.

Distributed_Train_Loader → [
({"alice" : ptr_batch_1, "bob" : ptr_batch_1 }, {"alice": ptr_1abel_1, "bob": ptr_label_1}),
({"alice" : ptr_batch_2, "bob" : ptr_batch_2 }, {"alice": ptr_1abel_2, "bob": ptr_label_2}),
({"alice" : ptr_batch_3, "bob":ptr_batch_3 }, {"alice":ptr_1abel_3, "bob": ptr_label_3})...]

Every iteration of Distributed_Train_Loader will return a tuple of dictionaries.

```
#owners among which data is to be distributed
data_owners = (alice, bob)

#instantiate class
Distributed_Train_Loader = Distributed_DataLoader(dataset= trainset,
                        data_holders= *data_owners,
                        batch_size=batch_size, shuffle=True)
"""
  Args:
    dataset :- the data to be distributed(torchvision.datasetMINST/CIFAR etc.)
    data_holders:- owners among which data is to be distributed
    batch_size:-- (int) size of batch
    shuffle :- bool
"""
#Training Loop
for i in range(len(Distributed_Train_Loader)):

  data_dict, label_dict = Distributed_Train_Loader.get_next_dict()
 """
    data_dict :  {"alice" : ptr_alice_data_batch_i, "bob" :ptr_bob_data_batch_i}
    label_dict:  {"alice":  ptr_alice_1abel_i, "bob": ptr_bob_label_i}
 """
  pass
```

1.6 Deliverable(s)

- Implementation of Distributed_DataLoader class in Distribute_Dataset.py.
- Notebook for SplitNN configuration-1 (Normal and Extended).
- Notebook for SplitNN configuration-2 (Normal and Extended).
- Notebook for SplitNN configuration-3.
- Report.md for comparison between the configurations I, II, and III regarding resource efficiency, accuracy, data and model privacy.

1.7 Project Timeline/ Roadmap :

Before June 1 (before coding officially begins):-

- Discuss my ideas with mentor and get feedback from them and update the idea accordingly.
- Create Distributed_DataLoader class in Distribute_Dataset.py.
- Create Notebook for SplitNN Configuration-1 with single and multi-split (Normal and Extended) network for vertically partitioned MNIST and CIFAR .
- Update report for configuration-1 regarding accuracy and efficiency in Report.md

1 June – 8 June (after coding officially begins)

- Create Notebook for SplitNN Configuration-2 with single and multi-split (Normal and Extended) network for vertically partitioned MNIST and CIFAR .
- Update Report.md for configuration-2 regrading accuracy and efficiency.
- **Phase-1 Evaluation**

9 June – 15-June

- Create Notebook for SplitNN Configuration-3 with for vertically partitioned MNIST and CIFAR .
- Update Report.md for configuration-2 regrading accuracy and efficiency.
- **Phase-2 Evaluation**

16-June - 29 June

- Create a table for comparison between the configurations.
- Get a review from mentor and implement changes accordingly.
- **Final Evaluation**

From May 5 – July 5 I have my Summer vacations so I'll be able to spend 6-8 hours daily. However, after July 20, I'll be able to spend no more than 5 hours daily as my next Semester begins.

2. About Me :

Who am I? What am I studying?

I am Abbas Ismail, a 3rd year undergraduate student at Birla Institute of Technology, Mesra. I have been working on Deep Learning and with PyTorch framework for around 1 year. My work includes data analytics challenge solving, freelancing and hackathons projects.

Why am I right for this Project?

I am familiar with the PySyft. I had created deep learning models with Federated learning and SplitNN techniques using PySyft library. Apart from this, I have completed the Secure and Private AI udacity course and have understood well concepts like Federated Learning, Secure-Multi party computation, homomorphic encryption, etc.

Past Projects and Developments

My software development has revolved mostly around Deep Learning and related frameworks. Some projects that I've worked on include:

Some technologies which I have used: Python, C++, Tensorflow, PyTorch, OpenCV.

- PC Security System:
  - Real-time security system which notifies admin through a text message if someone unauthorized is accessing your PC.
  - Used OpenCV and Tensorflow for real time face recognition.
  - Used Twilioi API for sending text message to admin's mobile .

  Code can be found here

- Compact Neural Network:
  - Creates any type of Neural network as per the given parameter
  - Interface in IPython Notebooks for easy input of parameters

  Code can be found here

- Malaria Detection:
  - This model detects whether the cell is uninfected or parasitized with pytorch.
  - This model is trained on dataset having of infected and uninfected total of 27,558 images.

  Code can be found here

- Plant Disease Classification
  - This model classifies the type of disease plant is suffering through.

  Code can found here

Preferred Method Of Contact:

OpenMined's Slack handle:- Abbas Ismail

GitHub : @abbas5253

Email: abbas.tel2342@gmail.com

Skype: abbas.tel2342@outlook.com

Phone : +91-7974236343

Address : 106, Raj residency, manik bagh, Indore(M.P), India.