COMPUTER SCIENCE AND DATA ANALYTICS

Course: Guided Research I

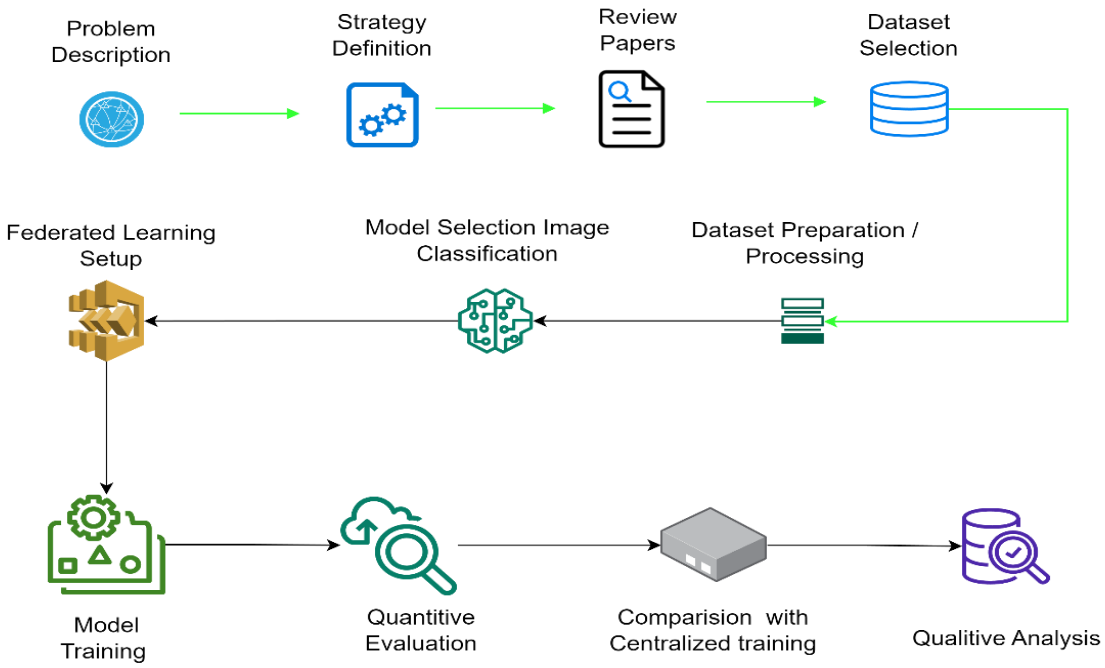
Midterm Report

Title: Federated Machine Learning Implementation on Image Classification

Student: Ali Asgarov

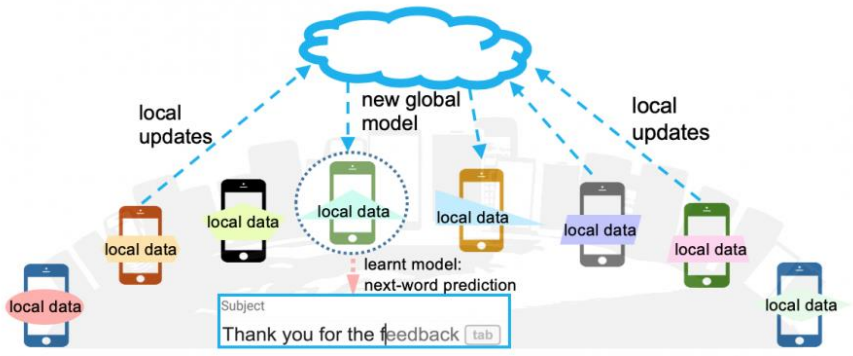
Instructors: Prof. Dr. Stephen Kaisler, Assoc.Prof Jamaladdin Hasanov

The roadmap for the project



1. Problem Description (Done)

Federated Learning (FL) is a concept that emerges as mobile devices, including phones, tablets, and smartwatches, have become essential computing devices for many individuals. These devices store vast amounts of valuable and private data due to their advanced user interactions and powerful sensors. Utilizing models trained on such data could greatly enhance the functionality and capabilities of intelligent applications. However, due to the sensitive nature of this data, there are associated risks and responsibilities. This is where the concept of Federated Learning becomes relevant. This project aims to explore federated learning for image classification while preserving privacy. The goal is to develop a federated learning algorithm tested on image classification tasks and evaluate its performance compared to centralized training approaches.



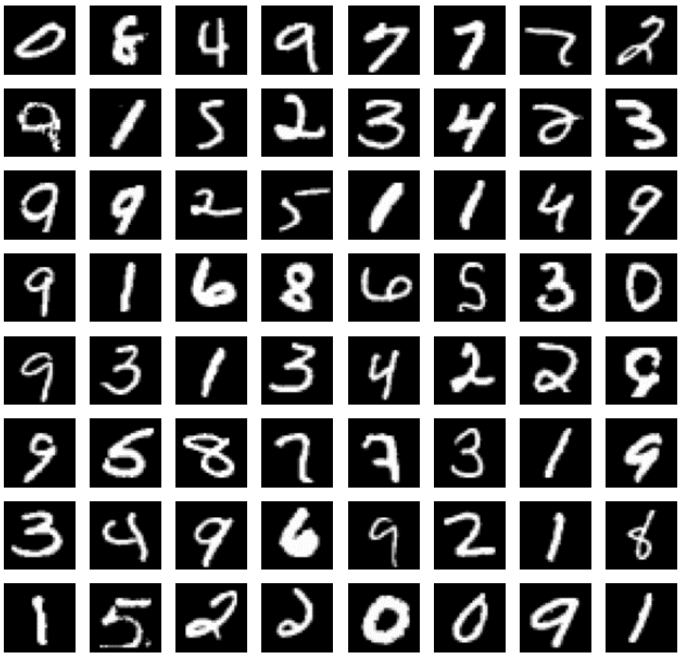
In Federated Learning (FL), every client independently trains its own model in a decentralized manner. This means that the model training process occurs separately on each client's device. Only the learned model parameters are transmitted to a trusted central server, where they are combined and used to update the aggregated main model. The updated aggregated model is then returned to the individual clients, and this iterative process continues.

2. Strategy Definition (Done)

I have mentioned the multiple frameworks available [1], such as PyTorch and TensorFlow Federated API, and I have already chosen **PyTorch** for conducting my project and have already started and done multiple steps that will be mentioned in the further sections. This decision based on various factors, including compatibility issues with the TensorFlow Federated Learning framework, especially with the latest Python version (e.g., Python 3.10).

3. Dataset Selection (Done)

The implementation of the described Federated Learning approach is being conducted using the **MNIST** dataset. The **MNIST** dataset consists of grayscale images with dimensions of 28x28 pixels, depicting handwritten numbers ranging from 0 to 9.



4. **Dataset Preparation (Done)**

a. In the **Report 1** I have mentioned that the MNIST data set does not contain each label equally. Therefore, to fulfill the IID requirement, the dataset has been grouped, shuffled, and then distributed so that each node contains an equal number of each label.

```
def shuffle_and_select_labels(data, seed, amount):
    df = pd.DataFrame(data, columns=["labels"])
    df["index"] = np.arange(len(df))
    selected_labels = {}
    np.random.seed(seed)
    shuffled_indices = np.random.permutation(len(df))
    for i in range(10):
        label_name = "label_" + str(i)
        label_info = df[df["labels"] == i]
        selected_indices = shuffled_indices[label_info.index][:amount]
        selected_data = df.loc[selected_indices, ["labels", "index"]]
        selected_labels[label_name] = selected_data
    return selected_labels
```

The function follows these steps:

The function creates a DataFrame from the input data and generates shuffled indices using a random seed. For each label, it filters the DataFrame to retrieve the corresponding rows. It selects shuffled indices and limits the selection to the specified sample amount. The selected samples, along with their labels and indices, are added to a dictionary. Finally, the function returns the dictionary containing the selected samples for each label.

b. After doing the shuffling and selection, I needed to have such a function which divides the indexes in each node with an equal number of each label.

```
def get_subsamples_indices(label_dict, number_of_samples, amount):
    sample_dict = {}
    batch_size = math.floor(amount / number_of_samples)
    for i in range(number_of_samples):
        sample_name = "sample" + str(i)
        sample_data = []
        for j in range(10):
            label_name = "label_" + str(j)
            label_info = label_dict[label_name]
            start_index = i * batch_size
            end_index = (i + 1) * batch_size
            sample_data.append(label_info[start_index:end_index])
        sample_data = pd.concat(sample_data, axis=0).reset_index(drop=True)
        sample_dict[sample_name] = sample_data
    return sample_dict
```

The function initializes an empty sample_dict dictionary to store subsample data. It calculates batch_size based on the specified amount of samples and subsamples. For each subsample, it creates a subsample name and an empty sample_data list. For each label, it selects a portion of label data based on batch_size. The label data is appended to sample_data. After processing all labels, sample_data is concatenated into a subsample DataFrame. The DataFrame's index is reset, and it's added to sample_dict with the subsample name as the key.

After processing all subsamples, the function returns the sample_dict dictionary containing the created subsamples.

c. Then distributing of x and y data to nodes in dictionary comes into play.

```
def create_iid_subsamples(sample_dict, x_data, y_data, x_name, y_name):
    x_data_dict = {}
    y_data_dict = {}
    for i in range(len(sample_dict)):
        xname = x_name + str(i)
        yname = y_name + str(i)
        sample_name = "sample" + str(i)
        sample_indices = sample_dict[sample_name]["i"].values
        sorted_indices = np.sort(sample_indices)
        x_info = x_data[sorted_indices, :]
        x_data_dict[xname] = x_info
        y_info = y_data[sorted_indices]
        y_data_dict[yname] = y_info
    return x_data_dict, y_data_dict
```

The function performs the following steps:

It initializes empty dictionaries to store the subsampled data and labels. For each subsample in the input dictionary, it retrieves the corresponding indices. The indices are used to extract the subsampled data and labels from the original data. The subsampled data is added to the data dictionary with a unique name. Similarly, the subsampled labels are added to the label dictionary with a unique name. After processing all subsamples, the function returns the dictionaries containing the subsampled data and labels.

5. Model Architecture (Done)

In the Report 1 I have mentioned about designing a suitable deep learning model architecture for image classification and determining the hyperparameters and optimization algorithm to be used. A simple 3-layer model has been created for the classification process.

```
class Net2nn(nn.Module):
    def __init__(self):
        super(Net2nn, self).__init__()
        self.fc1 = nn.Linear(784, 200)
        self.fc2 = nn.Linear(200, 200)
        self.fc3 = nn.Linear(200, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

The Net2nn model is a three-layer neural network. It takes an input of size 784 and consists of two hidden layers, each with 200 units, followed by an output layer with 10 units. The model applies the rectified linear unit (ReLU) activation function after each hidden layer, except for the output layer.

I have train and validation functions for training and validation of the model.

```
def train(model, train_loader, criterion, optimizer):
    model.train()
    train_loss = 0.0
    correct = 0

    for data, target in train_loader:
        output = model(data)
        loss = criterion(output, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        prediction = output.argmax(dim=1, keepdim=True)
        correct += prediction.eq(target.view_as(prediction)).sum().item()


    return train_loss / len(train_loader), correct/len(train_loader.dataset)
```

```python
def validation(model, test_loader, criterion):
    model.eval()
    test_loss = 0.0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            output = model(data)

            test_loss += criterion(output, target).item()
            prediction = output.argmax(dim=1, keepdim=True)
            correct += prediction.eq(target.view_as(prediction)).sum().item()

    test_loss /= len(test_loader)
    correct /= len(test_loader.dataset)

    return (test_loss, correct)
```

The train function performs training on the model, calculates the training loss, and computes the training accuracy. The validation function evaluates the model on validation/test data, calculates the validation/test loss, and computes the accuracy.

I have examined what would the performance of the centralized model be if the data were not distributed to nodes at all?

```
------ Centralized Model ------
epoch:   1 | train accuracy:  0.8752 | test accuracy:  0.9530
epoch:   2 | train accuracy:  0.9560 | test accuracy:  0.9596
epoch:   3 | train accuracy:  0.9706 | test accuracy:  0.9699
epoch:   4 | train accuracy:  0.9784 | test accuracy:  0.9686
epoch:   5 | train accuracy:  0.9825 | test accuracy:  0.9718
epoch:   6 | train accuracy:  0.9869 | test accuracy:  0.9780
epoch:   7 | train accuracy:  0.9893 | test accuracy:  0.9785
epoch:   8 | train accuracy:  0.9921 | test accuracy:  0.9777
epoch:   9 | train accuracy:  0.9939 | test accuracy:  0.9765
epoch:  10 | train accuracy:  0.9947 | test accuracy:  0.9787
------ Training finished ------
```

In this example, the current model is kept intentionally simple to focus on comparing its performance with a centralized model trained on all training data. Although various improvements can enhance model performance, such as utilizing more complex models, increasing the number of epochs, or conducting hyperparameter tuning, the primary objective here is to evaluate the federated learning approach's effectiveness. By combining the parameters of locally trained models on their respective data and comparing them with a centralized model, valuable insights into the potential of federated learning can be gained.

Now I am examining what will be the performance while applying federated learning techniques.

6. Federated Learning Setup (In Progress).

This function creates a model, optimizer and loss function for each node.

```
def model_optimizer_criterion_dict(number_of_samples):
    model_dict = dict()
    optimizer_dict= dict()
    criterion_dict = dict()

    for i in range(number_of_samples):
        model_name="model"+str(i)
        model_info=Net2nn()
        model_dict.update({model_name : model_info })

        optimizer_name="optimizer"+str(i)
        optimizer_info = torch.optim.SGD(model_info.parameters(),
lr=learning_rate, momentum=momentum)
        optimizer_dict.update({optimizer_name : optimizer_info })

        criterion_name = "criterion"+str(i)
        criterion_info = nn.CrossEntropyLoss()
        criterion_dict.update({criterion_name : criterion_info})

    return model_dict, optimizer_dict, criterion_dict
```

The **model_optimizer_criterion_dict** function generates dictionaries of models, optimizers, and loss criteria based on the specified number of samples. It creates unique names for each sample and initializes corresponding instances for models, optimizers, and loss criteria. The dictionaries are then returned with the generated objects stored for each sample.

I am on the Federated learning setup and model training step of my roadmap.

Then the next steps are working on the functions for:

- Taking the average of the weights in individual nodes
- Sending the averaged weights of individual nodes to the main model and sets them as the new weights of the main model
- Sending the parameters of the main model to the individual nodes.
- Training individual local models in individual nodes.
- Comparing the accuracy of the main model and the local model running on each node.

In my research project, as I have stated in report 3 I will analyze the performance of the federated learning algorithm for image classification. I'll measure classification metrics (accuracy, precision, recall) as the dependent variable. The independent variables include the number of participating nodes, training rounds, learning rate, and dataset size. By conducting experiments and collecting data, I'll explore the relationship between these variables and the algorithm's performance. Statistical analysis, including hypothesis testing, will be used to draw conclusions based on the data. The DMAIC cycle will guide the analysis process, and the report will provide detailed explanations of the findings.