FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Edge Client Integration and Federation of Machine Learning Models

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Damian Jäger
Matrikelnummer 11776843

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dr.-Ing. Stefan Schulte
Mitwirkung: Thomas Hiessl (Siemens Technology)

Wien, 6. März 2021

Damian Jäger   Stefan Schulte

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Edge Client Integration and Federation of Machine Learning Models

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Damian Jäger
Registration Number 11776843

to the Faculty of Informatics

at the TU Wien

Advisor: Dr.-Ing. Stefan Schulte
Assistance: Thomas Hiessl (Siemens Technology)

Vienna, 6th March, 2021

Damian Jäger Stefan Schulte

Erklärung zur Verfassung der Arbeit

Damian Jäger
Johann-Weber-Straße 58, 1210 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 6. März 2021

Damian Jäger

Kurzfassung

In vielen Bereichen sind Daten verteilt, haben unterschiedliche Besitzer und sind privat, wodurch es sehr schwer ist, sie zentral zu sammeln, um Machine Learning (ML) Modelle zu trainieren. Federated Learning (FL) ermöglicht es, in solchen Szenarien ML Modelle zu trainieren, ohne dass die rohen Daten zwischen Teilnehmern geteilt werden müssen. Dies ist möglich, indem Updates der Modelle der individuellen Clients an einen zentralen FL Server gesendet werden. Obwohl FL für Aufgaben auf Smartphones, wie Vorhersage des nächsten Wortes, gedacht war, sind Anwendungen in verschiedenen Bereichen möglich. Industrial Federated Learning (IFL) ist ein Ansatz, der versucht, Federated Learning an den industriellen Kontext anzupassen.

Diese Arbeit behandelt den Entwurf und die Implementierung des IFL Clients, der eine pythonische API für ein bestehendes IFL System ist, und einer IFL Anwendung, der IFL App, die den IFL Client verwendet, um auf das IFL System zuzugreifen und ML Modelle mit FL zu trainieren. Weiters wird die IFL App mit bestehender Management Software auf Edge Geräten aufgesetzt.

Um das IFL System und den IFL Client zu evaluieren, wird ein Skalierungsexperiment durchgeführt. Dabei wird gemessen, wie die Genauigkeit und die Anzahl an Kommunikationsrunden bis zur Konvergenz durch das Hinzufügen von Clients - und somit Daten - zu einer Federation in FL, beeinflusst wird. Bei diesem Experiment hatten die einzelnen Clients ausreichend Daten, um ein gutes Modell zu erlangen. FL zu verwenden führt zu schnellerer Konvergenz und spart so Rechenaufwand. Dies kann die zusätzlichen Kommunikationskosten von FL wert sein, da Edge Geräte, auf denen IFL Anwendungen laufen, oft nur begrenzte Rechenleistung haben.

Abstract

In various fields, data is distributed, owned by different entities and privacy sensitive, making collecting it centrally to train Machine Learning (ML) models very hard. Federated Learning (FL) enables training a central ML model in such scenarios in a way that does not require any raw data to be shared by participants. This is possible by only sharing updates of each client's models with the central FL server. While FL was initially designed for smartphone-specific tasks, such as next-word predictions, applications in different fields seem feasible. Industrial Federated Learning (IFL) is an approach aiming at tailoring FL to the industrial context.

This thesis aims at designing and implementing the IFL Client, which is a pythonic API for an existing IFL System, as well as an IFL Application, the IFL App, that uses the IFL Client to access the IFL System to train ML models using FL. Further, the IFL App is deployed to edge devices using existing management software.

To evaluate the IFL System and IFL Client, a scaling experiment is performed which measures how accuracy and the number of communication rounds to convergence are affected by adding clients and, therefore, data to a federation in FL. The results show that in the given dataset a single client has enough training data to obtain a good model. However, using FL leads to faster convergence which saves computational cost. This can be worth the communication cost of FL as edge devices running IFL Applications may have limited processing power.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Learning from distributed data with different owners without exposing the data is a goal many research communities have pursued for a long time [16]. Federated Learning (FL) is an approach proposed by McMahan et al. [22] that allows exchanging knowledge between multiple clients to collaboratively train a central Machine Learning (ML) model. In contrast to non-federated approaches, sharing raw data is not necessary, which is possible by decoupling model training from direct access to training data. This way, FL reduces privacy and security risks. If more clients participate in a federation, more training data becomes available which may lead to better models. Bonawitz et al. [9] designed a large-scale FL system which is applied to different ML problems on smartphones. They include next word prediction for a keyboard application, on-device item ranking of search results and content suggestions for on-device keyboards. These examples illustrate the benefits of FL well, as they all possess the following properties: They deal with privacy-sensitive and highly distributed data, which would make it otherwise harder to train a ML model as individual clients do not have sufficient training data to obtain a good model on their own.

Industries have to deal with similar challenges when it comes to training ML models. Due to highly heterogeneous working conditions of machines, additional challenges arise, which Hiessl et al. [14] proposed to tackle with Industrial Federated Learning (IFL), an approach that aims at tailoring FL to the industrial context. IFL is designed to make use of *Edge Computing* by using *edge nodes*, which are computing devices in close geographical proximity to machines [27], as an intermediary between machines and the server. This allows the edge nodes to take the heavy computational load of training ML models while keeping the raw data on devices owned by the operator of machines.

1.2 Aim of the Work

The aim of this work is to design and implement the *IFL Client*, which is an API for using IFL. It is then used to build an application that uses the IFL Client for IFL, called an *IFL Application*. In order to provide a proof of concept and evaluate the result, the IFL App is deployed to Industrial Edge[1] devices. Industrial Edge is an Edge Computing platform from Siemens. Among others, it includes tools for developing applications as well as managing edge devices and applications [2].

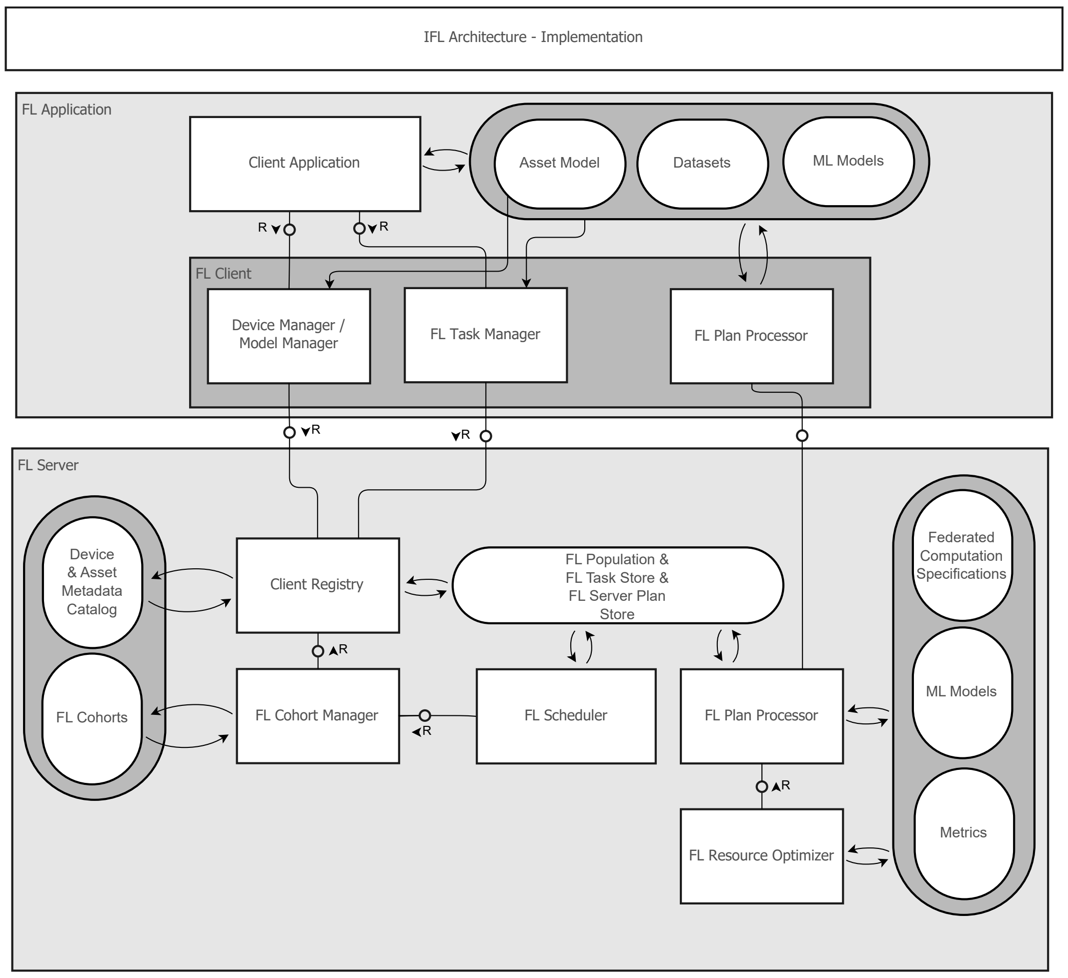

Figure 1.1: IFL Architecture [14]

[1]https://new.siemens.com/global/en/products/automation/topic-areas/industrial-edge/simatic-edge.html

1.2.1 IFL Client

Based on a research prototype of the *IFL System* by Hiessl et al. [14] which contains the *IFL Server* hosted on the Cloud Computing platform Amazon Web Services[2], the IFL Client is implemented. Figure 1.1 shows how application, client and server are related to each other. The IFL Client provides an API for accessing the IFL System and handles the client-side ML model training. It can be integrated into IFL applications which can be deployed to edge nodes. The IFL Client uses the API provided by the IFL Server and provides the following functionality:

- **Asset management:** Assets are the digital representation of machines used in IFL [14]. Assets and organizations can be created, updated, listed and deleted (CRUD).
- **Model management:** CRUD of ML models. This contains both managing ML models in their internal representation of the IFL System as well as the parameters of ML models.
- **IFL Task submission and execution:** IFL Tasks can be submitted to request the execution of FL training with other suitable IFL Clients. The IFL Client also has to be able to execute an IFL Task on request.
- **IFL Run and metrics retrieval:** After successful execution of an IFL Task, the IFL System provides a corresponding *IFL Run* object which the IFL Client should be able to retrieve. The IFL Run contains information on the configuration of the IFL Task as well as model evaluation results.

The IFL Client is a *Python* package that can be used by any IFL Application based on Python to access the IFL System. It should therefore provide an API that is easy to use. This work aims to design and implement the IFL Client and describe its architecture.

1.2.2 IFL App

An IFL Application to provide a proof of concept for both the IFL Client and the IFL System is implemented. This *IFL App* uses the IFL Client to access the IFL System and to federate a ML model for the *MNIST*[3] dataset which contains handwritten digits. It also provides output to track the progress of an IFL Task. To show that the IFL System supports multiple Deep Learning frameworks, two versions of the app, one using PyTorch[4] and one using TensorFlow[5] are implemented. The IFL App is a Python application packed into a Docker[6] image.

[2]https://aws.amazon.com
[3]http://yann.lecun.com/exdb/mnist
[4]https://pytorch.org
[5]https://www.tensorflow.org
[6]https://www.docker.com

1.2.3 Edge Deployment and Integration

The final aim of this work is to deploy the IFL App to Industrial Edge devices to show the interaction between the IFL System and the IFL App on actual Industrial Edge devices. For this, existing management software is used, emulating a real-world use case. A scaling experiment is performed to evaluate the performance of the IFL System and the IFL App.

1.3 Methodology

To achieve the aim of this work, the following methods are used:

1. **Literature review**: The literature review covers background information on ML, FL and Edge Computing. Furthermore, core concepts of IFL are covered.
2. **Design**: In the design phase, the architecture for both IFL Client and IFL App is created. Interests of data scientists and asset owners have to be considered. Furthermore, the interplay and communication between IFL Client and IFL System are described.
3. **Implementation**: As discussed previously, both the IFL Client and the IFL App are implemented in Python. For executing IFL Tasks, the framework *PyGrid*[7] is used to handle the actual ML work.
4. **Evaluation**: The evaluation aims at evaluating how well both IFL Client and IFL System perform. For this, a combination of Industrial Edge devices and virtual machines is used. The main goal is to evaluate how adding more clients to a federation affects model quality.

1.4 Structure of the Thesis

Chapter 2 of this thesis covers the *Background* of the topics Deep Learning, FL, Edge Computing and IFL. Based on these topics, *Related Work* is discussed in Chapter 3. In Chapter 4, the *Design* of the IFL Client and the IFL App are described. Furthermore, the interactions with the IFL System as well as the deployment to Industrial Edge devices are covered. Chapter 5 provides details on the *Implementation* of the IFL Client and IFL App. In Chapter 6, the *Evaluation* of the IFL System and IFL Client is described. Finally, the *Conclusion* (Chapter 7) summarizes the thesis and its results, and gives an outlook at future work.

[7]https://github.com/OpenMined/PyGrid

CHAPTER 2

Background

In this chapter, concepts used in IFL are discussed. First, Deep Learning as well as FL are discussed. Next, Edge Computing is described as it is an integral part of maintaining privacy in IFL. Finally, IFL as an adaption of FL to the industrial context is discussed.

2.1 Deep Learning

LeCun et al. [17] describe, that when using conventional ML techniques, transforming raw data into an internal representation that is suitable for a model to produce good results has been a great challenge for a long time. This transformation is required, as conventional ML techniques struggle with processing natural data. For example, ML systems for image classification require that the pixel values are transformed into a feature vector. *Representation Learning* enables machines to find the internal representation they require automatically from raw data. *Deep Learning* is a form of Representation Learning where multiple layers of representation that build upon each other are used. For example, the first layer transforms the raw input into some more abstract representation, the second layer takes this representation and transforms it into an even more abstract representation and so forth. These transformations can be composited to learn highly complex functions. Deep Learning is capable of finding intricate structures in high-dimensional data. It can be applied to a wide range of problems in many different fields. Examples include image classification, speech recognition and analysing particle accelerator data [17].

Deep Learning is often applied to images where the raw data is an array of pixels. The first layer transforms them into a representation which often contains the occurrence of edges at some locations and orientations. Based on this, the second layer usually finds motifs. The third layer may then detect familiar objects based on the arrangement of motifs in the previous representation. Further layers can then detect objects that the network is intended to find [17].

2.2 Federated Learning

FL is a privacy-preserving distributed Deep Learning approach introduced by McMahan et al. [22]. In FL, a federation of devices (*FL clients*) solve an ML problem and are coordinated by a central *FL server*. Datasets are kept on the client and therefore training data is never sent to the server. Instead, clients train the model locally and only updates to the global model are sent to the server. This allows the global model to be trained without the server needing to access the raw training data.

McMahan et al. [22] describe that FL is best suited for problems where the data is privacy sensitive and training using distributed data is advantageous compared to using proxy data available on the server. They initially suggested using FL for training models with data on mobile devices. This includes models for speech recognition, text entry and selecting good photos. Compared to centralized training on a server, FL offers more privacy, since the data does not have to leave the client. The updates sent to the server should be ephemeral and contain less information than the dataset. However, updates may still contain sensitive information. This problem can be tackled with *Secure Aggregation* which allows the server to aggregate the sum of all updates without being able to access the individual updates [10].

FL enables training a model with data from multiple parties without compromising privacy. This makes it possible to use ML for problems for which individual parties do not want to share their data, but do not have enough training by data themselves. Therefore, FL has the potential to be used in a variety of fields. A notable example is smart healthcare. As medical data is highly sensitive it is only accessible to isolated medical facilities and is hard to gather. Due to this lack of data, creating good ML models is a great challenge. Since FL enables training models without the need to expose medical data, it has the potential to vastly improve smart healthcare [28]. Similarly, owners of industrial machines do not want to expose the data their machines produce, as they might contain information about the way they operate. However, ML can, for example, be used for smart condition monitoring, which aims to determine the state and health of a machine by analyzing sensor data as described by Bangert [7]. FL may be able to improve the quality of such models, as more training data becomes available.

2.2.1 Federated Optimization

The term Federated Optimization was introduced by McMahan et al. [22] and refers to the optimization problem in FL. Compared to other distributed optimization problems it deals with Non-IID (Independent and Identically Distributed) training data. Different clients may also have different amounts of data. In Federated Optimization, a synchronous update scheme where communication is performed in rounds is used. In each round a random C-fraction of K clients is selected and the server sends the current model parameters to them. The clients update their local models using their local datasets and send the updates to the server which updates the global model.

When examining neural networks the objective function is of the form $\min_{w \in \mathbb{R}^d} f(w)$ where w represents the model parameters and $f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w)$. Typically, $f_i(w)$ is the loss for the prediction made for some data with the model parameters w. In that case, $f(w)$ is the average loss of the model with parameters w over n samples [22]. K clients hold the training data where P_k is the set of indices of data points which client k holds and $n_k = |P_k|$. The objective for Federated Optimization proposed by McMahan et al. [22] then is $f(w) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(w)$ where $F_k(w) = \frac{1}{n_k} \sum_{i \in P_k} f_i(w)$.

2.2.2 Federated Averaging

McMahan et al. [22] propose the FederatedAveraging (FedAvg) algorithm for updating the global model based on the local updates. As Stochasic Gradient Descent (SGD) is widely used in Deep Learning, the FedAvg algorithm is based on it. In each round, a C-fraction of all clients is selected. Typically a fixed learning rate η is used. Each client k calculates the average gradient $g_k = \nabla F_k(w_t)$ for the model w_t on its local data. The server can then calculate and apply the aggregated update $w_{t+1} \leftarrow w_t - \eta \sum_{k=1}^{K} \frac{n_k}{n} g_k$ which can also be written as $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$ where $w_{t+1}^k \leftarrow w_t - \eta g_k$. This means that each client takes a step of gradient descent on its local model and the server computes a weighted average of all models. The client can now update its local model iteratively by executing the update $w^k \leftarrow w^k - \eta \nabla F_k(w^k)$ multiple times. The number of training passes each client makes can be controlled by the parameter E. With the parameter B, the batch size for the local updates can be controlled. The entire FederatedAveraging algorithm can be found in Algorithm 2.1.

Taking the average of any ML models can lead to arbitrarily bad models. When models are initialized independently and then trained and averaged, low quality models may be the result. If all models are initialized with the same seed, however, averaging models leads to higher quality models than the individual models in experiments by McMahan et al. [22] on the MNIST training set.

2.2.3 Federated Learning Tasks

Based on the work of McMahan et al. [22], Bonawitz et al. [9] built a production Federated Learning system and introduced the terms *FL task* and *FL population*. An FL population represents a particular learning problem or application and is identified by a globally unique name. FL tasks represent specific computations such as training the model. Once an FL client is ready to run an FL task for an FL population, it contacts the FL server.

In this system, Bonawitz et al. [9] split the rounds of training into three phases: selection, configuration and reporting. In the selection phase, devices signal to the server that they are ready for training. The server then chooses devices that will participate in the round and tells the remaining devices to reconnect at a later time. During configuration, the server sends the FL plan to the clients and they start the local training. In the reporting phase, the server waits for the clients to send their updates and aggregates them. If a sufficient number of clients report their updates, the server updates the global model.

Algorithm 2.1: FederatedAveraging after McMahan et al. [22]

```
1  Server executes:
2      initialize $w_0$
3      for each round $t = 1, 2, \ldots$ do
4          $m \leftarrow \max(C \cdot K, 1)$
5          $S_t \leftarrow$ (random set of $m$ clients)
6          for each client $k \in S_t$ in parallel do
7              $w_{t+1}^K \leftarrow$ ClientUpdate($k, w_t$)
8          end
9          $w_t + 1 \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$
10     end
11
12 ClientUpdate($k$, $w$): //Run on client k
13     $B \leftarrow$ (split $P_k$ into batches of size $B$)
14     for each local epoch $i$ from 1 to $E$ do
15         for batch $b \in B$ do
16             $w \leftarrow w - \eta \nabla \ell(w; b)$
17         end
18     end
19     return $w$ to server
```

2.2.4 Secure Aggregation

Even though updates sent to the server in FL are ephemeral, it might be possible to obtain information about the training data of a client from them [22]. This poses a privacy risk that can be solved using Secure Aggregation. Secure Aggregation is a class of secure multi-party protocols which allows private values held by different parties to be aggregated without exposing the private data to other parties. Bonawitz et al. [10] propose a Secure Aggregation protocol for Federated Learning. It utilizes the fact that the server does not need to know the values of the individual updates, but only the sum.

Each of the n users $u \in U$ has their private value x_u. The server S needs to be able to obtain $\sum_{u \in \widetilde{U}} x_u$ where $\widetilde{U} \subseteq U$ and $|\widetilde{U}| \geq \frac{n}{2}$ but must not be able to access any $u \in U$. Thus, S can learn no more than what is derivable from the aggregate $\sum_{u \in \widetilde{U}} x_u$ about each x_u and the users can not learn anything.

To achieve this, Bonawitz et al. [10] developed a protocol with four rounds. In the first two rounds, clients establish a *shared secret* for masking the aggregated update. For this, a scheme such as Shamir's Secret Sharing [26] can be utilized. Devices dropping out during this round will not be considered in the final aggregate. In the third round, clients send their updates cryptographically masked to the server. Updates uploaded in this step will be part of the final aggregate. If too many clients drop out, the entire aggregation fails. In the final round, clients share their part of the cryptographic information necessary to allow the server to compute the aggregated update. Unless too many clients drop out, this step completes even if not all clients finish it.

This secure aggregation protocol is designed to work, even if part of of the clients drop out. Cost in computation, communication and storage grow quadratically, effectively limiting the number of participants in real-world application. To circumvent this limitation, Bonawitz et al. [9] suggest aggregating intermediate sums until a master aggregator aggregates them into the final result.

2.3 Computing on Cloud and Edge

In the past decade, *Cloud Computing* has been a dominant paradigm in the IT discourse according to Satyanarayanan [25]. It offers high scalability combined with removing the need for companies to build data centers by using computing resource from large service providers. With the emergence of the Internet of Things (IoT) and mobile computing, however, *Edge Computing* has become more and more relevant. Edge Computing leverages added computing and storage resources on devices placed close to mobile devices and sensors.

2.3.1 Cloud Computing

According to the National Institute of Standards and Technology (NIST), Cloud Computing is a model that allows ubiquitous and convenient access to a shared pool of computing resources [24]. These resources can be scaled up or down with minimal management effort. Providers of computing resources serve multiple customers. Examples for these providers are Amazon Web Services (AWS)[1], Microsoft Azure[2] and Google Cloud[3]. Resources are shared using a multi-tenant model where resources are dynamically assigned. Customers therefore do not control which hardware is used.

The *Cloud* refers to the data centers operated by the providers of computing resources mentioned before. A major advantage of using the Cloud is the possibility to scale quickly. Services can scale up to handle high load and then quickly scale back down to save money [12].

2.3.2 Edge Computing

In contrast to Cloud Computing, where data processing is done centrally in data centers, Edge Computing, explores using *edge nodes* for performing computations as an intermediary between the Cloud and *edge devices*, according to Varghese et al. [27]. Edge nodes can be devices through which network traffic flows, such as routers, switches, and base stations, or any other computing devices which are placed geographically closer to edge

[1]https://aws.amazon.com
[2]https://azure.microsoft.com
[3]https://cloud.google.com

devices than the Cloud. Edge devices refer to devices on the edge of the internet, such as mobile phones or sensors.

Motivation

Varghese et al. [27] describe that Edge Computing enables computing tasks geographically closer to the edge devices compared to the centralized Cloud Computing approach. This reduces latency and therefore improves the service provided. That way, Edge Computing can make real time applications, which are not running on front-end devices, viable.

Edge devices often have limited resources. In the industrial context, for example, machines may do not have sufficient computing power to train a ML model. An edge node could assume the workload while keeping latency low [27].

Another aspect mentioned by Satyanarayanan [25], is that Edge Computing can reduce the amount of network traffic the Cloud has to handle. This can be achieved by only passing extracted information and metadata for the edge nodes to the Cloud. For example, in IFL sensor data only has to be sent to an edge node. Only the model updates are then passed to the Cloud, which amounts to significantly less network traffic to the Cloud. Edge nodes can also act as a fallback service when Cloud outages occur. In IFL this means that the ML models stored on an edge node are still accessible when the Cloud is available and therefore machines are not affected.

Edge nodes can decide which data is passed on to the Cloud. This way, the owners can enforce a privacy policy for their data. In an industrial setting, for example, machines may pass sensitive data to edge nodes for processing. Then edge nodes can decide, which data to pass on to the Cloud [25].

Challenges

While edge nodes are better equipped to handle computationally intense workloads then edge devices, not overloading them is a challenge according to Varghese et al. [27]. It is important that they maintain high throughput, which might not be possible when they are overloaded. Therefore, it is important to choose the tasks edge nodes have to handle correctly in order to keep the quality of service.

Ensuring the security of public edge nodes is another challenge in Edge Computing. In data centers using virtualization, the risks for both customers and providers are known. Multi-tenancy on edge nodes can only be done securely, if security is central in developing the required technology. Furthermore, the primary function of devices such as routers, switches, and base stations must not be affected [27].

2.4 Industrial Federated Learning

Federated Learning was initially designed with smartphones as clients in mind [22]. Training decentralized models with FL is, however, interesting for machine data as well.

There are different types of machines and they operate under different conditions. This can lead to a lack of similarity of data which could decrease the quality of a model if federated. Hiessl et al. [14] proposed Industrial Federated Learning which aims to tailor FL to the industrial context and its requirements.

In IFL, machines are treated as their digital representation called *assets* which generate data, e.g., through sensors, to be used by FL. In order to differentiate different types of machines, assets are assigned an *asset type*. Each asset type has a number of *aspects* with their corresponding *aspect types* which contains the *variables*. An example would be a concrete machine (asset) which is an engine (asset type). It has sensors collecting data about vibrations at the surface (aspect). Surface vibrations are vibrations (aspect type) and are collected in x, y and z dimensions (variables). For assets, a description of the environment is saved as well.

Assets with a different asset type use a different global model as their data does most likely not have enough similarity. Machines of the same asset type may produce highly heterogeneous data as well based on the environment they are operating in. Therefore, the selection of clients for an FL task is highly relevant. To account for the lack of similarity in data originated from different machines of the same asset type, Hiessl et al. [14] introduced *FL cohorts*. FL clients can submit FL tasks to an FL cohort which has a corresponding global ML model. Clients within a cohort only share their updates with FL tasks in the same cohort. With this approach training of a model with highly heterogeneous data due to different operating conditions is prevented. Tasks can be moved to a different cohort to improve model quality. For this, factors like model accuracy and environmental changes are considered.

IFL is designed to make use of Edge Computing. Machines do not perform ML tasks themselves, but delegate these to edge nodes which are computing devices in close geographical proximity to the machines operated by the owner of the machines. There the data is stored and used in the training of ML models. This approach enables machines to work independently without being affected by the heavy workload caused by training ML models while keeping the data within company bounds and therefore private. Due to the proximity to the edge nodes, latency is kept low.

CHAPTER 3

Related Work

3.1 BrainTorrent

BrainTorrent is a server-less peer-to-peer FL environment proposed by Roy et al. [13] which aims to adapt FL to the context of medical centers. The authors identified some key differences between the environment of FL in its initially proposed form and the medical environment: While FL is designed to be able to scale to millions of devices, medical centers are part of much smaller communities. Furthermore, centers are expected to have good communication infrastructure, eliminating the bottleneck of communication cycles. Lastly, there may not be a trusted server, instead centers want to coordinate with each other directly. This context is similar to the one encountered in IFL, making it an interesting comparison. Unlike IFL, BrainTorrent requires a high level of coordination between all participating parties.

Instead of relying on a central server, Roy et al. [13] decided to take a peer-to-peer approach for BrainTorrent. This enables clients to start update rounds more freely. Furthermore, the central server as a single point of failure is eliminated, making BrainTorrent more reliable than traditional FL.

3.2 Energy Consumption and Accuracy

Magid et al. [21] explored how energy consumption of image classification tasks using ML on edge devices is affected by parameters such as dataset size and image resolution, among others. Furthermore, they looked at the effects these parameters have on processing duration and accuracy. Magid et al. found that lowering the resolution of images significantly reduces energy consumption but does not impair the accuracy drastically. Dataset size, too, has a drastic effect on energy consumption. Reducing energy consumption is relevant in the industrial context as well. However, this is outside the scope of this work.

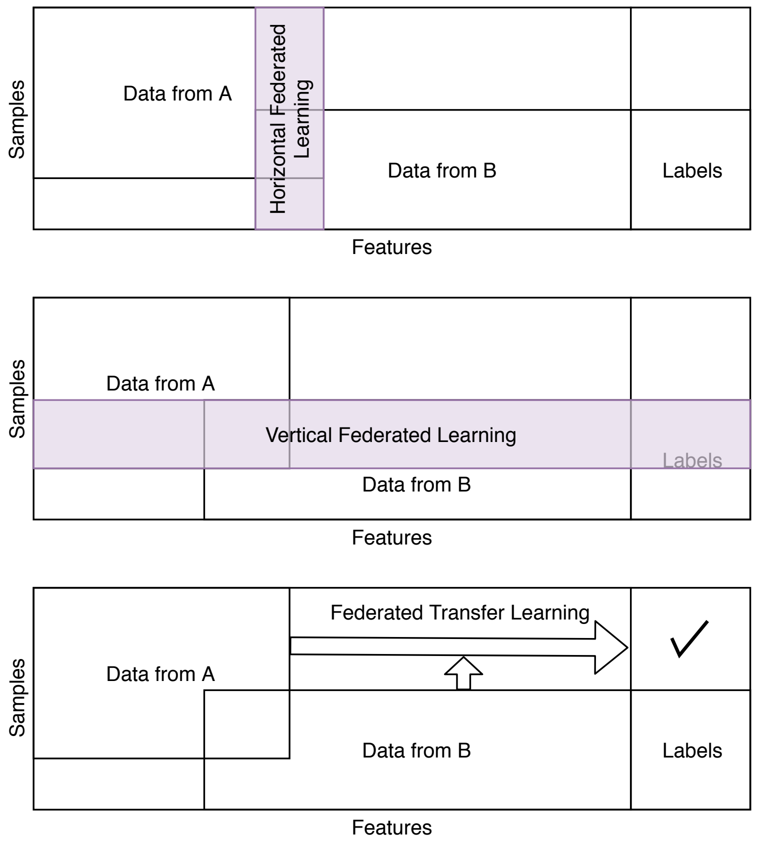

Figure 3.1: Categories of Federated Learning after Yang et al. [28]

3.3 Categorization of Federated Learning

Yang et al. [28] divide FL into the three categories *horizontal FL*, *vertical FL* and *Federated Transfer Learning (FTL)*. Horizontal FL can be used if datasets share the same features but have different samples. The FL framework introduced by McMahan et al. [22] is an example of horizontal FL. If the sample ID space overlaps but the features do not, vertical FL is applicable. It combines feature spaces which leads to a model with more features. An example for vertical FL would be if two companies in a different business field are located in the same area. Due to their different business goals, their models have different feature spaces but they share a large number of clients. Lastly, FTL is best suited for scenarios in which neither sample nor feature spaces have large overlaps. As an example, Yang et al. [28] state a bank in China and an e-commerce company in the United States. Naturally, they neither share many customers nor features. FTL aims to provide solutions for the whole sample and feature space. All three categories are depicted in Figure 3.1.

3.4 Challenges in Federated Learning

FL still faces a lot of challenges, Aledhari et al. [5] list the following, among others: First of all, devices participating in FL can be highly heterogeneous. They may have widely varying capabilities in terms of capacity, processing power and network capacity, which can lead to extremely high communication costs. Moreover, fault tolerance has to be considered, as devices may drop out at any point. For IFL, these issues are not relevant as a solid infrastructure can be expected. Second, the FL framework by McMahan et al. [22] as well as IFL require that all devices agree on and use the same architecture for the ML model. In some cases, for example in the medical context, different parties may want to choose to use a different model architecture based on their environment and capacities. Especially in healthcare, sharing specifics about models can be hard due to privacy concerns. In the industrial context, companies may have similar concerns.

Li et al. [20] describe further challenges in FL: Statistical heterogeneity and privacy concerns. An important challenge in FL is that devices generate varying amounts of data that are non-identically distributed as, for example, usage patterns of smartphones are highly heterogeneous. This may add complexity, as in distributed optimization the assumption of independent and identically distributed data is commonly used. Furthermore, even though FL was designed with privacy in mind, privacy is still a concern. As raw training data does not leave the device, only updates are sent to the server, improving privacy significantly. However, these updates may still contain sensitive information which is then available to the central server. As this is a privacy issue, different approaches try to improve privacy but often at the cost of model performance or efficiency [20]. This trade-off has to be considered when implementing IFL as well.

Bhagoji et al. [8] identified *model poisoning* as a threat in FL. Model poisoning means that an adversary causes the global model to wrongly classify a set of chosen inputs with high confidence. For this, the attacker only needs to control a small number of agents. Bhagoji et al. [8] conclude that FL in its original form is highly vulnerable to model poisoning. However, it is more challenging to do it stealthily. Similarly, Bagdasaryan et al. [6] were able to introduce backdoor functionality into the global model without strongly affecting the accuracy on the main task. As metrics for evaluating model quality focus on the accuracy of the model for its main task, they might not detect what else the model has learned. This allows introducing covert backdoor functionality. These security issues are highly relevant in the industrial context as well. However, IFL is only concerned with improving performance, leaving them to future research.

3.5 Applications of Federated Learning

Aledhari et al. [5] describe that FL has gained a lot of attention for its application for mobile devices. One of the most prominent applications is Gboard, a mobile keyboard application by Google. This use case includes next-word prediction and word completion. It features an environment which is highly suitable for FL: User input is privacy sensitive data that is highly distributed and low latency is crucial. Another application of FL on mobile devices is ranking browser-history suggestions. This feature turned out to be hard and expensive to test, as it was rolled out to a large number of Firefox[1] users, risking negative user experience.

In industrial engineering, too, FL has gained popularity according to Li et al. [19]. Due to laws and regulations concerning data privacy protection, FL enables leveraging distributed datasets in this area. Applications include data monitoring and visual inspection tasks. Further, research into using FL for detecting attacks in the communication system of drones and detecting credit card fraud has been conducted. FL has the potential to enable more industries to use ML to become more intelligent. IFL is an approach aiming at improving FL in this area.

Another area of application for FL is healthcare. As individual medical facilities might not have enough training data available, FL is promising for training prediction models on patient data. Possible applications for FL in healthcare range from biomedical imaging analysis to analyzing electronic health records with natural language processing [19]. Further applications include functional magnetic resonance imaging (fMRI) analysis and brain tumor segmentation [5].

3.6 Federated Learning Frameworks

Li et al. [19] state the two open source frameworks TensorFlow Federated (TFF)[2] and Federated AI Technology Enabler (FATE)[3] as the mainstream frameworks for FL. TFF features FedAvg for updating models as well as Secure Aggregation to improve privacy. It is highly scalable and has been used in applications installed on more than ten million devices. FATE is designed for a cross-organizational architecture. It also supports Transfer Learning and other ML algorithms.

Since then, other frameworks such as PySyft[4], CrypTen[5] and LEAF[6] have emerged, which each focus on different aspects such as privacy, mobile devices or multi-tasking. The IFL System uses PySyft, which is mainly focused on privacy [5].

3.7 Industrial Edge Computing Reference Model

Dai et al. [11] describe the industrial edge computing reference model which consists of three layers: Edge Computing controller (control layer), Edge Computing gateway and industrial cloud platform. The control layer contains, for example, robots and sensors which deliver data to the Edge Computing gateway on request via protocols such as

[1]https://www.mozilla.org/en/firefox/
[2]https://www.tensorflow.org/federated
[3]https://fate.fedai.org/
[4]https://github.com/OpenMined/PySyft
[5]https://crypten.ai/
[6]https://leaf.cmu.edu/

RESTful APIs, WebSocket or TCP/UDP. The Edge Computing gateway is responsible for tasks such as data buffering and filtering, real-time monitoring and data analysis. It exchanges data with the industrial cloud platform through encrypted challenges and protocols such as MQTT or any of the aforementioned protocols. The industrial cloud platform can contain a public or private cloud. It provides storage and computing resources to analyze data collected by the lower layers. In the case of the IFL System, IFL Applications are placed in the Edge Computing gateway layer and receive data from sensors in the control layer. They communicate with the IFL Services, which are placed in the cloud platform layer, using RESTful APIs and WebSockets.

3.8 Edge Management Software

Li et al. [18] cover architecture patterns for developing industrial software, in particular industrial edge systems. They present microservice patterns for the stages of industrial software delivery, which are deployment, monitoring, adaptation and testing. These include, for example, patterns for moving services from the cloud to edge devices, load balancing between edge and cloud, and testing microservices on edge devices without side effects. The work at hand uses existing management software for deploying the IFL App to edge nodes and is therefore not concerned with the implementation of such systems.

3.9 Conclusions

Since FL is a rather recent approach, there still are many open research challenges. As privacy and security are important promises of FL these topics have been studied intensely. Solutions typically lead to having to make a trade-off between privacy and model performance or efficiency. As different applications of FL feature different requirements, adaptions for different contexts have been developed. For example, IFL targets the industrial context and BrainTorrent the context of medical centers. The industrial edge computing reference model describes how an Edge Computing service can be structured. This work covers a design for an application in the Edge Computing gateway layer.

CHAPTER 4

Design

This chapter discusses the design of the IFL Client and the IFL App. To understand the interactions with the IFL System, the IFL System is described as well. Finally, the deployment of the IFL App to Industrial Edge devices is discussed.

4.1 IFL System

The IFL System is a prototype based on the work of by Hiessl et al. [14]. It is hosted on AWS and provides the server-side services for managing data related to IFL as well as for executing FL tasks with cohorts. The IFL System consists of various services which are available through REST APIs.

4.1.1 PyGrid

PyGrid is a peer-to-peer network for training ML models using the secure Deep Learning framework PySyft[1] which supports FL and is based on the ML framework PyTorch[2]. It is composed of a network, nodes and workers. The network is used to manage instructions related to executing FL and routes them to grid nodes. Grid nodes store models as well as the private data uploaded to them. They also manage workers, to which they issue instructions to compute data [4].

The IFL System makes use of PyGrid, to which it delegates communication and processing for the FL component of executing IFL Tasks. IFL Applications spawn or use existing PyGrid nodes (from now on referred to as grid nodes) which connect to a PyGrid network (grid network) provided by the IFL System. Ideally, grid nodes run on edge nodes, as this ensures that the data, IFL Applications send to grid nodes for training, is not sent to the cloud.

[1]https://github.com/OpenMined/PySyft/
[2]https://pytorch.org/

4.1.2 Services

The IFL System offers the following services, each of which provides a REST API, to the IFL Client:

IFL Registry The IFL Registry (registry) can be used to obtain, create, modify and delete organizations, assets, asset types, aspects, aspect types, variables, ML models and datasets. It is responsible for persisting this data. Furthermore, IFL Tasks can be submitted to the registry for execution.

IFL Plan Processor The IFL Plan Processor (plan processor) is responsible for the execution of IFL Tasks. It also provides feedback on the status of IFL Runs.

IFL Grid Network Manager The IFL Grid Network Manager (grid network manager) offers functionality for spawning and destroying grid nodes in the cloud to IFL Applications. This can be used if it is not possible to spawn the grid node on the edge. It therefore violates the idea of using Edge Computing for IFL. Due to limitations in the current version of PyGrid, however, it is necessary to use grid nodes running in the cloud if no edge node with a publicly reachable IP address can be spawned on the edge.

4.1.3 Server Plans

Once a sufficient number of tasks have been submitted to the registry, the plan processor generates a *server plan*. A server plan holds information about the execution of tasks, such as the *IFL Population*, which contains the participating tasks and cohorts, and a description. Furthermore, it contains the URL of the grid network used.

4.1.4 IFL Algorithms

The IFL System supports several algorithms for updating the global models during task execution. Clients can choose between the following when submitting a task:

- **Central Learning:** Central Learning collects the training data from each client and updates the global model by centrally training on the server. Therefore, it is not an FL approach. However, its results serve as a benchmark to compare different algorithms, as it represents a near best-case scenario for obtaining good models.
- **Sequential FL:** When using Sequential FL the model is sequentially updated by each participating client within every round. This means, that the model is sent to a client and updated there. The resulting model is then sent to the next client and so forth.
- **Federated Averaging:** Federated Averaging uses the FedAvg algorithm proposed by McMahan et al. [22] to update the model.
- **Cohort-based Federated Averaging:** Cohort-based Federated Averaging is based on the idea of forming FL cohorts introduced by Hiessl et al. [14] combined with FedAvg. First, a cohorting algorithm forms cohorts from the participating

tasks. To update the ML model, the FedAvg algorithm is then used within the cohorts.

4.2 IFL Client

As the services provided by the IFL System are offered as REST APIs, the IFL Client is developed to provide a pythonic API. Furthermore, it provides additional utility that enables executing IFL Tasks through the IFL System. The IFL Client is a Python library that can be used to communicate with and use the IFL System. Figure 4.1 shows the primary use case where it serves as a link between an edge node connected to assets and the IFL System. In the following, actors using the IFL Client, the domain model and architecture as well as the most important workflows are described.

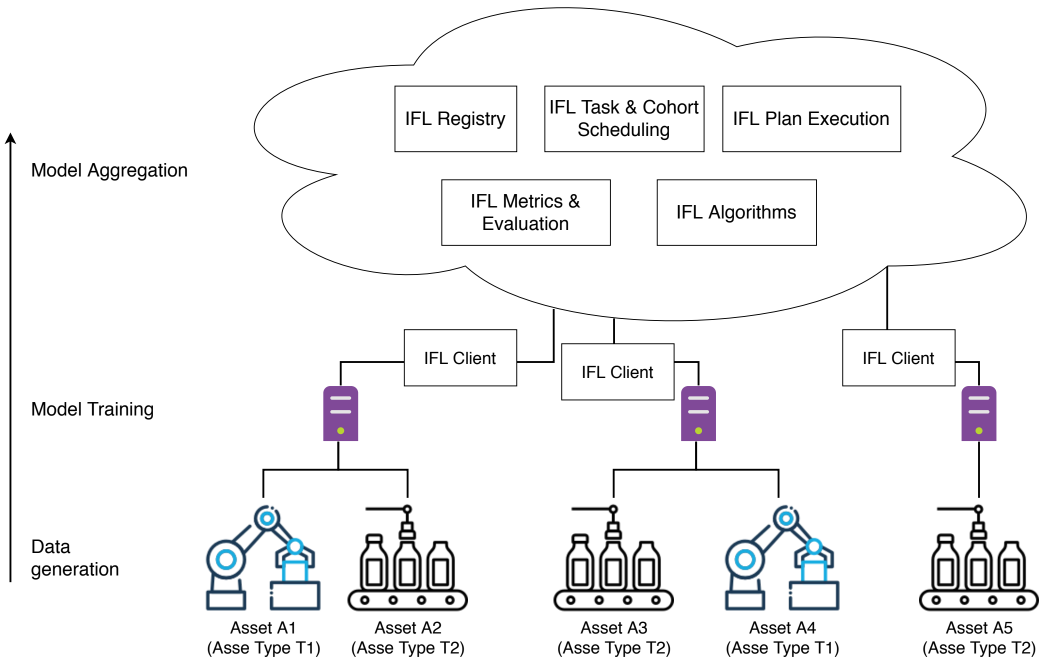

Figure 4.1: IFL System Functional Blocks

4.2.1 Actors

The IFL Client has to be able to serve the needs of machine builders, data scientists, asset owners and machine operators. As it is a Python library, it can be used either in programs and automated systems or directly in Python for manual use, depending on the needs of the actor.

Machine builders: Machine builders create the assets for machines they sell to customers. Additionally, they assign the respective aspect types and asset types to the assets.

Data scientists: Data scientists are responsible for managing ML models and datasets. They provide the structure of the ML model as well as metadata about datasets. Furthermore, data scientists can use the IFL Client to obtain, analyze and improve ML models.

Asset owners: Asset owners need to be able to manage their organization as well as their assets. Additionally, they can use the IFL Client in their IFL Applications to submit and execute IFL Tasks. For this, they use data generated by their sensors to participate in the FL process. The IFL Client allows them to implement IFL Applications with reduced effort as it provides the required functionality to use the IFL System.

Machine operators: Machine operators use the IFL Client in their IFL Applications to obtain ML models, that are continually being improved through IFL. They profit from better operation of their machines by using these models.

4.2.2 Domain Model

For creating a domain model, the terminology used by Hiessl et al. [14] is used. Figure 4.2 depicts the models the IFL Client has to handle as well as their relationships to each other. In the following, they are described in more detail.

Each machine is represented by an *Asset* identified by an *asset_identifier* for which metadata such as *name*, *location* and a description of the operating environment is stored. Each asset belongs to an *Organization*, which is represented by its *name*, *industry* and *location*. A location consists of a *street_address*, *postal_code*, *city*, *country* as well as the exact position given in *longitude* and *latitude*. Assets are assigned an *AssetType* identified by an *asset_type_identifier* for which a *name* and a *description* is stored. It can be assigned multiple aspects and variables. An *Aspect* has a *name* and is assigned at least one *Aspect Type* which contains a *name*, *description*, variables and a *temporal_category* which can be static or dynamic. Each *Variable* consists of a *name*, *unit*, *data_type* and a *default_value*.

An FL task is represented as an *IflTask* (IFL Task) which has a *name*, *description*, *ifl_algorithm*, *federation_criteria* and a *cohort_strategy*. The ifl_algorithm specifies which IFL algorithm the task uses. IFL Tasks also store an *ifl_run_id* which is the id of the corresponding *IflRun* (IFL Run) object, and are associated with the *MlModel* that is used for training as well as the *asset* that has submitted the task. Each MlModel represents an ML model whose URL is stored in its *base_model_url*. It also has a *dataset* associated with it, a *name* and *description*, a *type*, which contains the format the ML model is stored in, and a *train_test_split*, which defines how much of the dataset should be used for training and testing, respectively. The *training_parameter* contain additional instructions for the training of the ML model, their format can vary between different IFL algorithms. Each dataset consists of a *name*, *description*, and *size* and is assigned an Aspect Type. An IFL Run contains the results of the execution of IFL Tasks and is

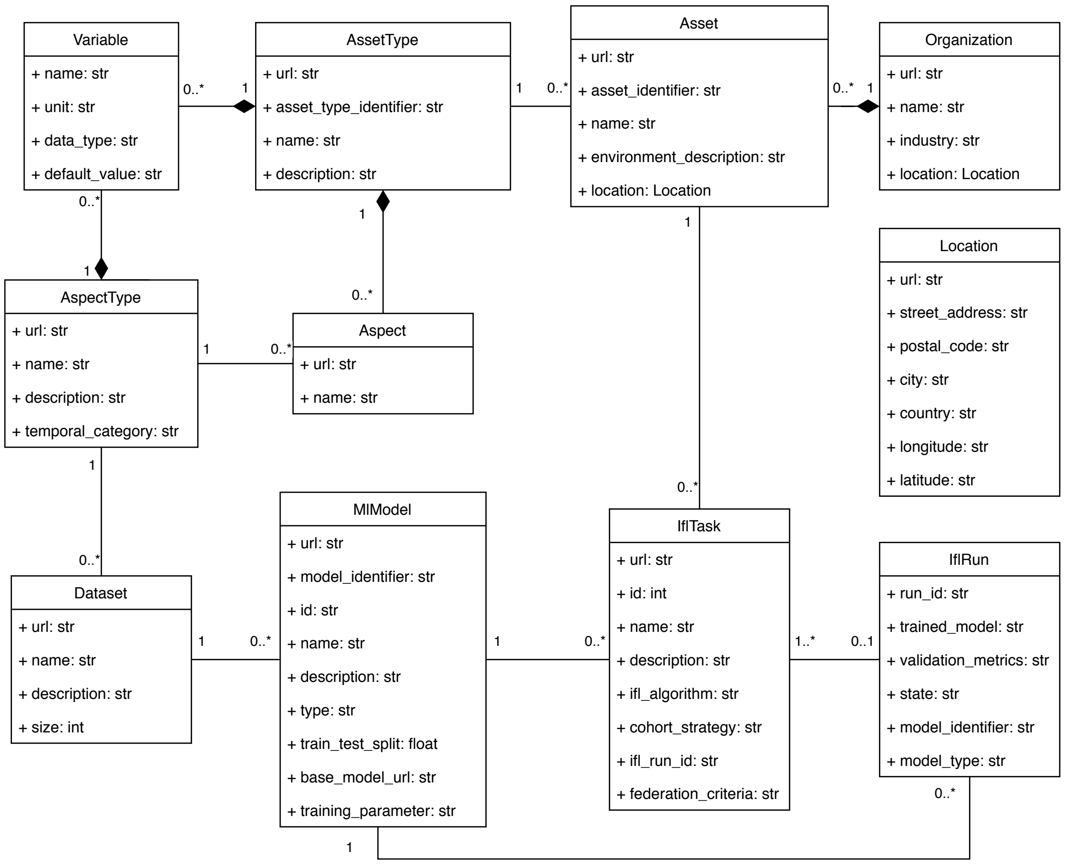

Figure 4.2: Domain Model

identified by a *run_id*. Whether it was successful is stored in its *state*. IFL Runs also have a URL to the updated model, stored in *trained_model*, and *validation_metrics*, which contain detailed information on the execution of the run. They also store information about the ML model used, in particular the *model_type* and *model_identifier*.

Each model, other than IFL Runs, also contains a *url* that is assigned by the IFL Registry and points to the corresponding instance in the registry.

4.2.3 Architecture

This section describes the components of the IFL Client and which services they provide. Figure 4.3 shows the parts of the library and how they are related to each other. While FLClient, Asset- and ModelManager, FLTaskManager and FLPlanProcessor interact directly with the user of the IFL Client, IFLRegistry-, IFLPlanProcessor- and IFLGridNodeManagerApi only provide services to other components of the IFL Client. In the following, each component is described.

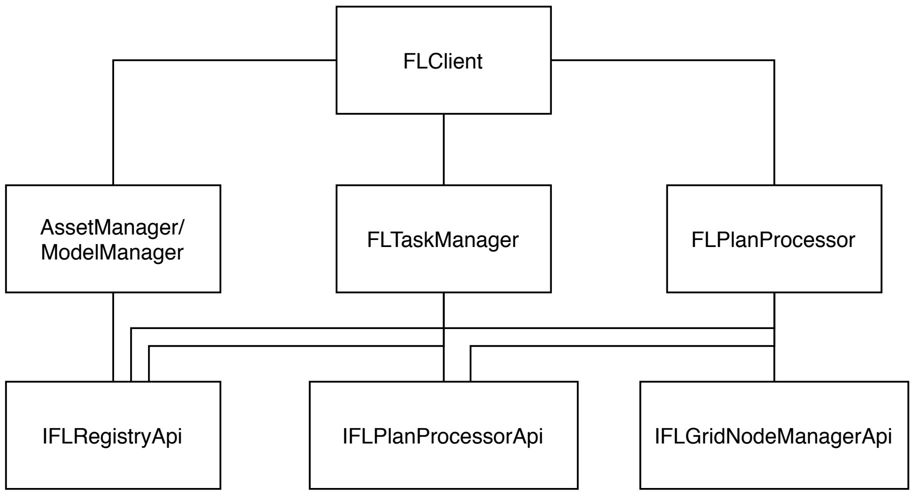

Figure 4.3: IFL Client Architecture

FLClient

The FLClient serves as an entry point into the IFL Client. It is responsible for managing the URLs of the IFL Registry, IFL Plan Processor and IFL Grid Network Manager the client uses. The FLClient can be used to obtain instances of AssetManager, ModelManager, FLTaskManager and FLPlanProcessor.

Asset- and ModelManager

The AssetManager and ModelManager provide a way to use the IFL Registry with Python objects. The AssetManager can be used to create, read, update and delete Organizations, Assets and AssetTypes, while the ModelManager provides the same functionality for MlModels, Datasets and AspectTypes. Furthermore, ML models can be imported and exported through the ModelManager.

FLTaskManager

The FLTaskManager provides functionality for submitting tasks as well as accessing additional information of tasks. IFL Tasks can be submitted and existing ones can be read. In order to track the progress of an IFL Task, the current progress can be queried to acquire temporary validation metrics. Furthermore, functionality for waiting for a task to complete is provided. Once an IFL Task has completed, its corresponding IFL Run can be accessed through the FLTaskManager.

FLPlanProcessor

The FLPlanProcessor contains the necessary utility to execute IFL Tasks and to manage PyGrid nodes. When executing a task, either an existing node can be used, or a

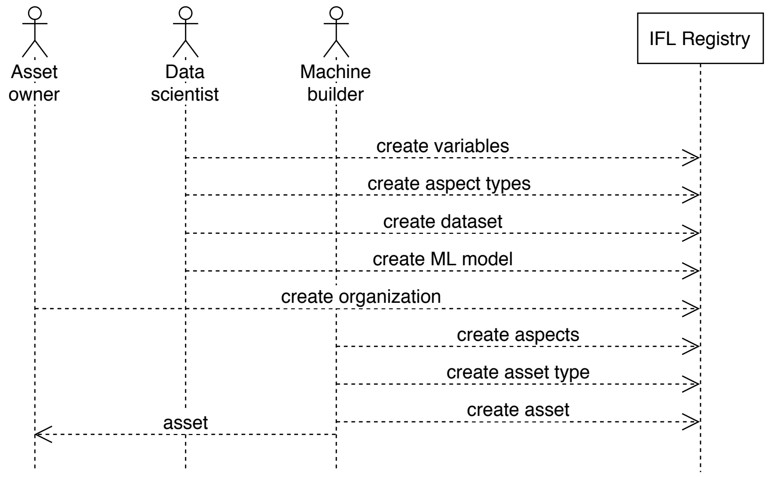

Figure 4.4: Model and Asset Registration

new node can be spawned either using a local Docker[3] installation or in the cloud by accessing the grid network manager. Grid nodes can be started and stopped through the FLPlanProcessor in either case. For the execution of tasks, the FLPlanProcessor provides utility for getting the address of the PyGrid network used for the task, registering datasets on grid nodes and marking tasks as ready.

IFLRegistry-, IFLPlanProcessor- and IFLGridNodeManagerApi

The IFLRegistryApi, IFLPlanProcessorApi and IFLGridNodeManagerApi provide an abstraction of the APIs offered by the services of the IFL System IFL Registry, IFL Plan Processor and IFL Grid Node Manager, respectively. They are responsible for parsing Python objects, dispatching calls to the APIs and parsing the responses.

4.2.4 Model Creation

Model creation is handled by a data scientist familiar with the ML problem at hand. At first, the variables representing the data collected by machines have to be determined and created in the IFL Registry as part of an aspect type. Afterwards, a dataset for the previously created aspect type is created. Lastly, the MlModel is added to the registry and an initial ML model is uploaded by the data scientist. Figure 4.4 shows the steps for registering an MlModel by the data scientist followed by the necessary steps for asset registration.

4.2.5 Asset Registration

For the registration of an asset, the asset owner has to have the organization registered in the IFL Registry. Furthermore, the relevant aspect types need to have been created by

[3]https://www.docker.com/

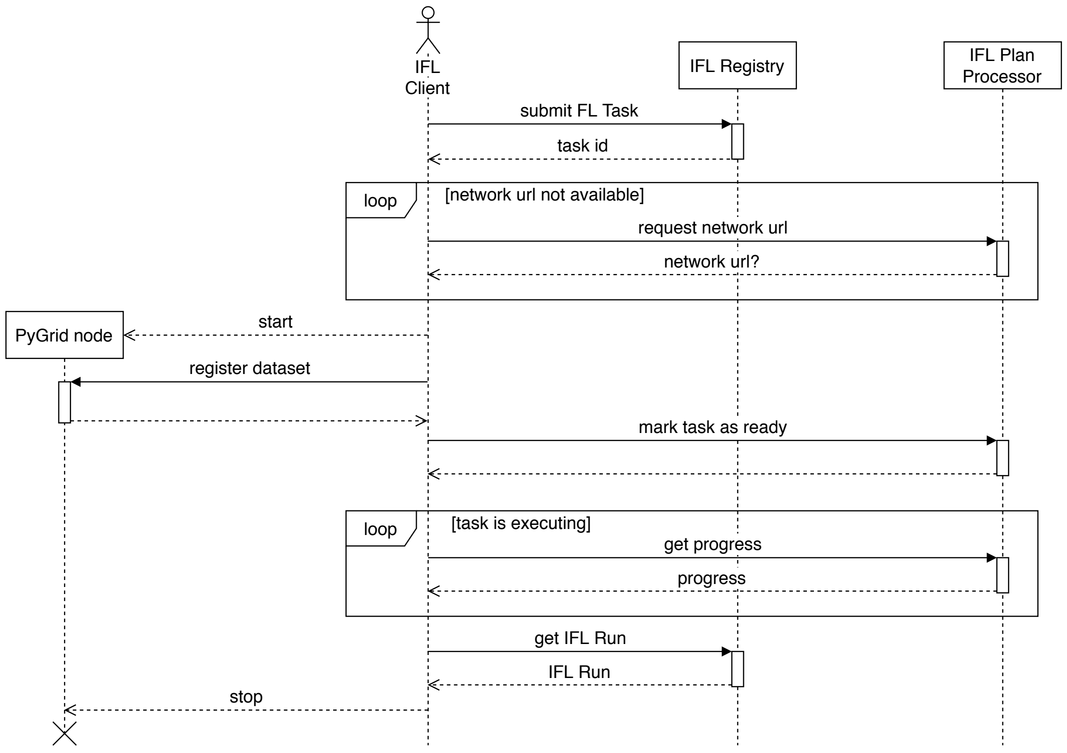

Figure 4.5: Task Execution

a data scientist. The machine builder can then create aspects and register an asset type with the necessary aspects and variables assigned. Finally, the machine builder registers the asset of this asset type to the registry and provides it to the asset owner.

4.2.6 IFL Task Execution

In order for a client to participate in the execution of an IFL Task, it has to indicate that it wants to join a federation by submitting a task to the IFL Registry. Once a sufficient amount of similar tasks, depending on the selected cohort strategy and IFL algorithm, has joined, the IFL Plan Processor indicates, that a server plan has been generated and provides the URL of a grid network for participating clients to connect to. Clients spawn their respective grid nodes, which connect to the grid network and register their training data to them. Once that is done, they tell the IFL Plan Processor that they are ready for training. The server waits for some amount of time for a sufficient number of clients to be ready before starting the training process. If not enough clients are ready in time, the server plan fails. Otherwise, the IFL Plan Processor starts executing training rounds. During execution, the client can query the progress of the training and receive intermediate validating metrics. As soon as the training is complete, an IFL Run is available from the IFL Registry. It contains a link to the updated model, which clients can use for inference. The client may choose to stop the grid node it started. Figure 4.5

shows the sequence of a successful IFL Task execution from the perspective of the IFL Client.

When submitting an IFL Task, the client can pass *federation criteria* to control properties of the federation. These include a minimum number of clients that have to join for the task to be executed.

4.3 IFL App

The IFL App is an IFL Application that uses the IFL Client to access the IFL System. It is able to execute IFL Tasks and use the updated model for inference. Furthermore, the IFL App provides a web interface for managing IFL Tasks and IFL Runs.

4.3.1 Architecture

The IFL App consists of the aforementioned web interface, data and logging services, and the *TaskExecutor*. This section describes them in greater detail. Figure 4.6 visualizes the parts of the IFL App and their dependencies to each other.

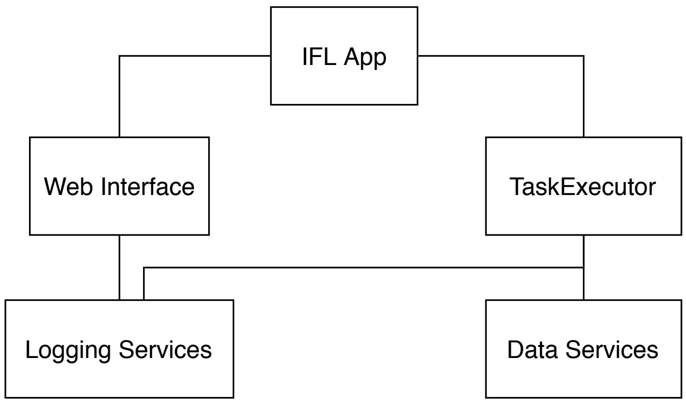

Figure 4.6: IFL App Architecture

Web Interface

The web interface provides a user interface for controlling the IFL App. It allows setting the id of the asset the current instance of the application represents, i.e. setting the identity of the machine within the IFL System. Further, IFL Tasks can be managed through the web interface. New tasks can be submitted and previously submitted tasks can be monitored. For this, the progress of a running task, which contains information on accuracy, loss and dataset size for each round, is output. For completed tasks, a link to the associated IFL Run is provided. For IFL Runs, the validation metrics are displayed, which contain the final progress of each task as well as the average accuracy and loss before and after the execution of the corresponding server plan. Furthermore, a graph containing the accuracy for each task and each round is shown.

With the help of the web interface, machine operators can keep track of the IFL Tasks executed by the IFL App as well as of the improvements in performance of the ML models achieved by IFL.

Data Services

The data services are responsible for providing the test and training data for executing IFL Tasks as well as for exporting the resulting ML models. They are also used for loading locally saved ML models which resulted from task execution.

TaskExecutor

The TaskExecutor is responsible for actually executing tasks and is invoked once a task has been submitted by the IFL App. For this, it uses the IFL Client, in particular the FLPlanProcessor and the FLTaskManager. The FLPlanProcessor is used to start and stop grid nodes in the cloud as well as for executing tasks and the FLTaskManager is used to obtain IFL Runs. During task execution, the TaskExecutor periodically requests the progress of the task from the FLTaskManager.

Logging Services

The logging services are responsible for handling writing log output. This includes writing to the standard Python logging API as well as writing log output to the databus of Industrial Edge devices. This enables analyzing and monitoring the IFL App through monitoring tools other than the web interface.

4.3.2 Task Submission and Execution

The execution of an IFL Task can be triggered through the web interface or by sending a HTTP POST request to the IFL App. For this, the id of an MlModel and a name and description for the task have to be provided. Additionally, federation criteria can be submitted. An IFL Task with the provided information is then submitted to the IFL Registry through the FLTaskManager. A TaskExecutor for this task is then started and executes the task using the IFL Client as described in Section 4.2.6 with a grid node running in the cloud, which is started with the grid network manager. The training data is provided by the data services. During the whole process, the logging services are used to provide output on the state and progress of the execution.

4.3.3 Inference

Once a task has been completed, the corresponding IFL Run is available and contains the updated ML model. This model can be used for inference through the IFL App. To show that the IFL System can be used with different ML framework, the IFL App

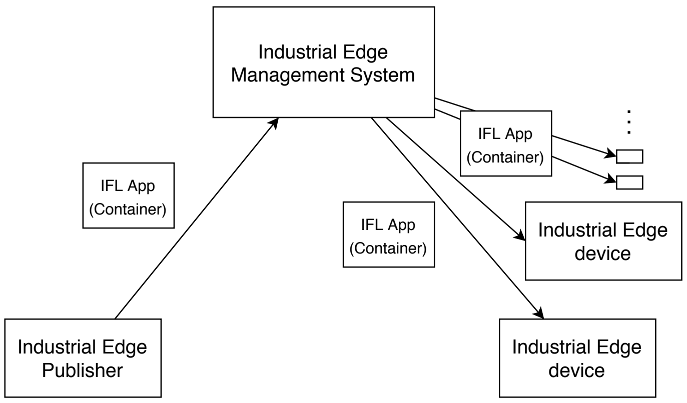

Figure 4.7: Industrial Edge Deployment

supports inference with the two widely-used Deep Learning frameworks PyTorch[4] and Tensorflow[5].

To start inference, the id of the IFL Run whose model should be used and a tensor containing the data on which inference should be executed, have to be provided. The data services are then responsible for obtaining the correct ML model. It is then parsed to the respective Deep Learning framework and the resulting predictions are returned.

4.4 Industrial Edge Deployment

This section covers the deployment of the IFL App to Industrial Edge devices using existing management software. For this, *Siemens Industrial Edge* is used.

4.4.1 Siemens Industrial Edge

Siemens Industrial Edge is an Edge Computing platform developed by Siemens[6] for the industrial context. It is designed to be used for tasks such as analyzing high-frequency data from machines, optimizing machine processes and condition monitoring. Siemens Industrial Edge consists of three parts: the *Industrial Edge Management System*, *Edge devices* and *Edge apps*. The Edge Management System serves as a central infrastructure to manage connected Edge devices. Users can install software, in particular Edge apps, on Edge devices and the system takes care of distribution. Edge apps are either provided by Siemens or can be individually developed. For this, the open source software Docker[7], which provides virtualization, runs on Edge devices. The IFL App is one such Edge app [1][3].

[4]https://pytorch.org/
[5]https://www.tensorflow.org/
[6]https://new.siemens.com/
[7]https://www.docker.com/

4.4.2 Deployment

To deploy the IFL App to Edge devices, the devices have to be registered in the Industrial Edge Management System first. Then, a project in the system has to be created for the app. Using the *Industrial Edge Publisher*, a program that allows packaging and uploading Edge apps to the Industrial Edge Management System, the IFL App is made available in the Edge Management System for distribution. From there, Edge devices, on which the IFL App should be installed, can be selected. The Edge Management System then takes care of the distribution and installation. Figure 4.7 shows how the IFL App is passed from the developer using the Edge Publisher to the Edge Management System and to Edge devices in its packaged state.

CHAPTER 5

Implementation

This chapter briefly covers implementation details of the IFL Client and IFL App as well as the most important technologies used.

5.1 Technologies

Both IFL Client and IFL App are developed in Python 3.8[1]. Pip[2] is used as a package installer and to manage dependencies. The IFL Client uses the older versions 1.4.0 and 0.2.9 for PyTorch[3] and PySyft[4], respectively, for compatibility with PyGrid. The IFL App uses Flask[5] for providing the web interface and endpoints.

5.2 IFL Client API

The IFL Client is available through the class *FLClient*, which is shown in Figure 5.1. It is initialized with the base URLs of the IFL Registry, IFL Plan Processor and IFL Grid Network Manager to be used. From the FLClient, the user can obtain instances of the AssetManager, ModelManager, FLTaskManager and FLPlanProcessor to use their functionality as described in Section 4.2.3.

5.3 IFL App Endpoints

The IFL App provides the following endpoints, which are used by the web interface but can be used by other applications as well:

[1]https://www.python.org/downloads/release/python-386/
[2]https://pip.pypa.io/en/stable/
[3]https://pytorch.org/
[4]https://github.com/OpenMined/PySyft
[5]https://flask.palletsprojects.com/en/1.1.x/

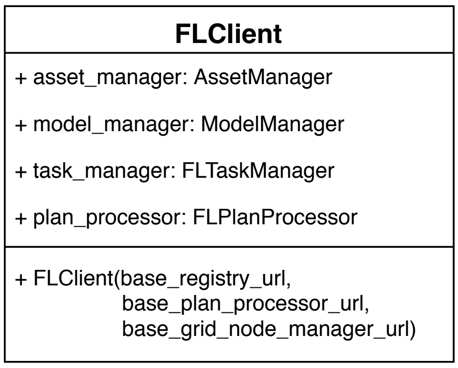
FLClient
+ asset_manager: AssetManager
+ model_manager: ModelManager
+ task_manager: FLTaskManager
+ plan_processor: FLPlanProcessor
+ FLClient(base_registry_url, base_plan_processor_url, base_grid_node_manager_url)

Figure 5.1: FLClient Class Diagram

Set asset: This endpoint can be used to set the id of the assets to be used by this Edge device in the future.

Execute task: To execute a task, this endpoint can be called. For this, the id of an MlModel, a name and a description of the task have to be submitted. Optionally, federation criteria can be submitted to control some aspects of the federation for the task, as discussed in Section 4.3.2. The endpoint then submits the task and starts a TaskExecutor on a new thread to execute it.

Inference: To perform inference on models resulting from an IFL Run, the IFL App offers two endpoints: one for inference with PyTorch and one with Tensorflow. They take a tensor in JSON format and perform inference on it. The endpoints return a JSON document containing the framework used for inference as well as the results. An example for a response can be seen in Listing 5.1.

Listing 5.1: Inference Response

```
{
  "framework": "pytorch",
  "predictions": [
    [
      -4.799382209777832,
      -5.252931118011475,
      -5.099876880645752,
      -3.368135452270508,
      -6.1609015464782715,
      -5.038578510284424,
      -7.690005779266357,
      -0.13186393678188324,
      -4.5497050285339355,
      -2.997725009918213
    ]
  ]
}
```

CHAPTER 6

Evaluation

This chapter covers the evaluation of the IFL Client and IFL System using a scaling experiment. First, the scaling experiment is described, then the results are discussed.

6.1 Scaling Experiment

To evaluate the IFL Client and the IFL System, a scaling experiment is performed. It aims at comparing the performance of models individual clients can achieve compared to models created using FedAvg with different numbers of clients.

6.1.1 Dataset

The MNIST[1] dataset is used to evaluate the performance of the IFL Client and IFL System. It contains 60,000 examples of handwritten digits and is widely used for evaluating ML and FL approaches, for example by McMahen et al. [22] and Roy et al. [13]. Further, it is an image classification problem, which is a class of problems that can be encountered in the industrial context. Examples include detecting object contamination in industrial food packaging [23] as well as quality and process control [15].

6.1.2 Methodology

The dataset is shuffled and split into four parts of equal size, each of which belongs to an instance of the IFL App. For every evaluation run, 10-fold cross validation is used and the results of each fold are averaged.

For the base case, a single instance of the IFL App is used to obtain the accuracy a single client can achieve with only its training data. Then, a second IFL App instance is

[1] http://yann.lecun.com/exdb/mnist

added and both Central Learning and FedAvg are applied. Finally, the number of clients is doubled again by adding two more instances of the IFL App to both Central Learning and FedAvg. This procedure enables analyzing the performance gains FedAvg delivers by training on more data without having to expose the data compared to the performance a single client can achieve. Furthermore, the differences in performance when doubling the number of clients once versus doubling them twice can be observed. The results of Central Learning can be used for comparison as a near best-case scenario.

Effectively, performance differences of the following scenarios are being looked at:

- A company has a single machine and trains a model on its data (one instance, Central Learning).
- Two/four companies have one machine each and share their data with each other to train a model (two/four instances, Central Learning).
- Two/four companies have one machine each and do not want to share their data. They collaborate to train a model using FL (two/four instances, FedAvg).

This allows evaluating the benefits companies can expect from collaborating with each other through FL without having to share their privacy sensitive data.

6.1.3 Metrics

The metrics used for evaluation are the number of communication rounds until the accuracy of the model converges and the best accuracy achieved. The accuracy for a round is calculated as an average of the achieved accuracies of all folds of all clients on their respective test sets. The highest accuracy of all rounds is then used as the best accuracy. The accuracy enables evaluating the quality of the model that is achieved in a given scenario. The number of communication rounds until convergence indicates how many communication rounds it takes until no more improvement of the accuracy occurs with additional rounds. As communication costs rise with each communication round in FL, a low number of rounds is desirable.

6.1.4 Setup

As the collected metrics are not influenced by the performance of the machine the clients run on, such as time-critical metrics, the IFL Apps are run in Docker containers on a PC instead of on Industrial Edge devices. The IFL System is hosted in the cloud and grid nodes are spawned in the cloud as well. Again, this does not affect the collected metrics. However, it does not make use of Edge Computing during the evaluation runs. Running grid nodes in the cloud is currently necessary due to the technical reasons described in Section 4.1.2.

The Deep Learning model used is the convolutional neural network McMahen et al. [22] use for evaluation on the MNIST dataset. It features two 5x5 convolution layers, each

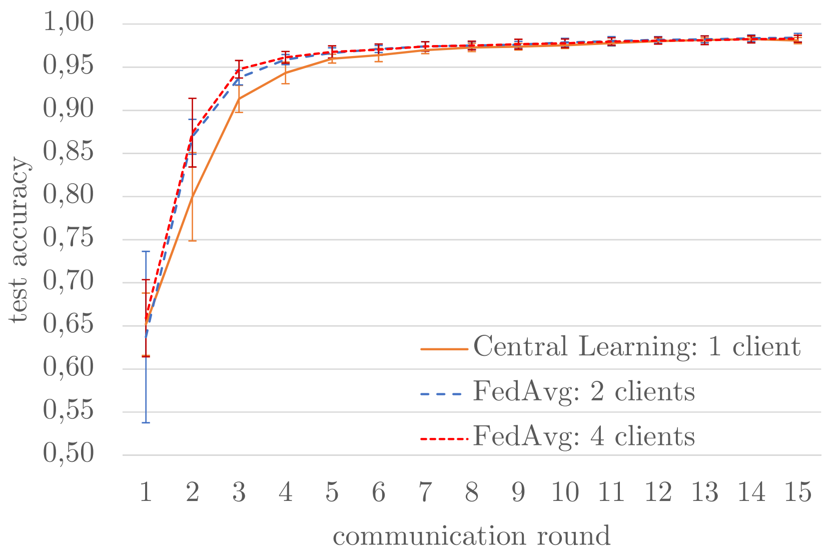

Figure 6.1: Central Learning with a single client versus FedAvg with 2 and 4 clients

followed by 2x2 max pooling, a fully connected layer with 512 units with ReLU activation and another fully connected layer with 10 units and Softmax activation. A learning rate of 0.001, batch size of 128 and the Adam optimizer were experimentally determined to give good results and used. Each evaluation run consists of 15 communication rounds.

6.1.5 Results

A single client using Central Learning was able to achieve an accuracy of 98.2% on the test set after 14 communication rounds, two clients achieved an accuracy of 98.5% after 13 communication rounds and four clients converged at an accuracy of 98.2% after eight communication rounds. For FedAvg with two and four clients, an accuracy of 98.4% and 98.3% respectively, were reached after 15 communication rounds. A table of the average accuracy and standard deviation for each round in each scenario can be found in Figure 6.3.

Figure 6.1 shows how Central Learning with a single client compares to FedAvg with two and four clients. FedAvg with two clients converged significantly faster than Central Learning with a single client. Adding another two clients to FedAvg gave nearly no advantage.

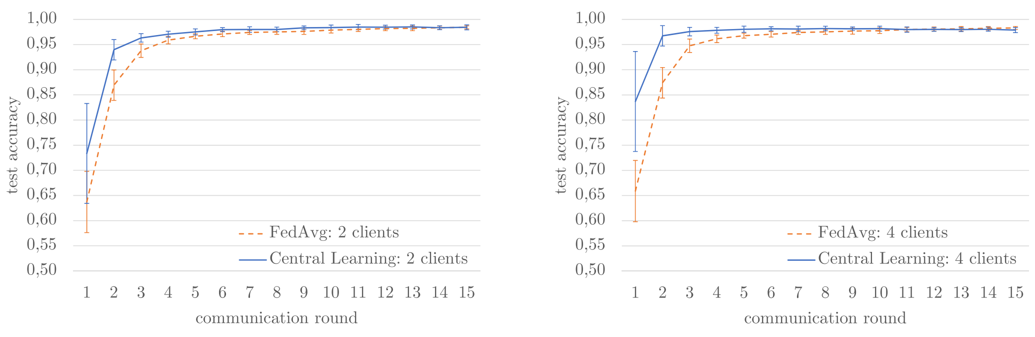

Figure 6.2: Central Learning versus FedAvg with 2 and 4 clients

6.2 Discussion

Surprisingly, all evaluation runs reached similar accuracies and adding clients did not improve the achieved accuracy. This could be due to the size of the MNIST dataset and the rather low complexity of the problem. The data of a single client was sufficient to reach good results leading to a low impact of number of clients on accuracy. A significantly higher number of clients with accordingly smaller datasets would likely lead to less accuracy for a single client and a similar accuracy for Central Learning and FedAvg with multiple clients.

Using FedAvg with multiple clients converges faster than if every client trains on its own data as Figure 6.1 shows. This was expected, since the data is IID and in each round more training data is available. As computational resources at the Edge are often limited and fewer communication rounds lead to less computational effort, using FedAvg is more efficient than letting each client train their own model. As the experiment shows, this is the case even if each client has enough data to obtain a good model by itself. Using FL does, however, introduce communication costs.

When comparing Central Learning and FedAvg for an equal number of clients, Central Learning converges in less communication rounds than FedAvg as Figure 6.2 shows. However, FedAvg reaches a similar accuracy to Central Learning.

	Central Learning: 1 client		Central Learning: 2 clients		FedAvg: 2 clients		Central Learning: 4 clients		FedAvg: 4 clients	
round	avg. accuracy	sd	avg. accuracy	sd	avg. accuracy	sd	avg. accuracy	sd	avg. accuracy	sd
1	0,6519	0,0361	0,7336	0,0993	0,6371	0,0609	0,8369	0,1217	0,6590	0,0447
2	0,7998	0,0512	0,9397	0,0202	0,8693	0,0303	0,9674	0,0082	0,8741	0,0398
3	0,9133	0,0157	0,9633	0,0084	0,9379	0,0134	0,9757	0,0061	0,9476	0,0102
4	0,9434	0,0126	0,9706	0,0059	0,9588	0,0075	0,9783	0,0052	0,9614	0,0068
5	0,9599	0,0051	0,9751	0,0061	0,9664	0,0049	0,9805	0,0050	0,9677	0,0070
6	0,9641	0,0077	0,9796	0,0041	0,9711	0,0054	0,9813	0,0039	0,9704	0,0065
7	0,9697	0,0039	0,9799	0,0055	0,9741	0,0040	0,9810	0,0045	0,9741	0,0054
8	0,9729	0,0049	0,9801	0,0046	0,9752	0,0049	0,9819	0,0041	0,9752	0,0051
9	0,9738	0,0039	0,9832	0,0037	0,9762	0,0060	0,9814	0,0051	0,9767	0,0057
10	0,9753	0,0036	0,9839	0,0049	0,9786	0,0056	0,9816	0,0041	0,9776	0,0050
11	0,9778	0,0033	0,9848	0,0053	0,9802	0,0042	0,9798	0,0058	0,9798	0,0043
12	0,9801	0,0032	0,9846	0,0039	0,9816	0,0040	0,9801	0,0052	0,9808	0,0041
13	0,9815	0,0026	0,9851	0,0040	0,9822	0,0046	0,9798	0,0051	0,9812	0,0050
14	0,9825	0,0026	0,9834	0,0039	0,9834	0,0034	0,9805	0,0041	0,9824	0,0042
15	0,9809	0,0036	0,9842	0,0050	0,9842	0,0030	0,9787	0,0057	0,9830	0,0041

Figure 6.3: Average accuracy and standard deviation for each round in each evaluation scenario

CHAPTER 7

Conclusions and Future Work

FL enables training ML models on decentralized data without sharing the raw training data. This makes it possible to collaboratively improve ML models on privacy sensitive data which has many potential applications in different areas ranging from biomedical imaging analysis to next-word prediction on smartphones. IFL tailors FL to the industrial context. Based on the IFL System, this work discussed the design and implementation of the IFL Client as an interface to the IFL System and the IFL App as a proof of concept for the IFL Client and the IFL System. Further, the deployment of the IFL App to Industrial Edge devices is described.

The experiments show that the computational effort of training a model of each client can be reduced by using FL as the model of a federation converges faster than a model trained on a single client. This is the case, even if every single client has enough training data to obtain a good model by itself. Using FL does, however, incur communication cost, making it a trade-off between computational and communication cost in such scenarios. In the experiments, FL achieved almost identical accuracy to Central Learning. The primary benefit of FL, however, is improving accuracy when individual clients have insufficient training data.

7.1 Future Work

This work focuses on the design of the IFL Client and IFL App as well as how they communicate with each other and with the IFL System. It does, however, not consider how the IFL App communicates with machines. Designing a communication protocol for this and expanding the data services to provide up-to-date data for training is considered future work.

The IFL App is designed to represent a single machine per instance. As Industrial Edge devices may have to be shared between multiple machines, expanding the IFL App to support managing multiple machines can be considered.

As IFL focuses on improving performance, security aspects are not considered at this point. Future work could investigate whether attackers can influence the cohort building process.

List of Figures

List of Algorithms

Listings

Acronyms

AWS Amazon Web Services. 9, 19

CRUD create, read, update and delete. 3

FATE Federated AI Technology Enabler. 16

FedAvg FederatedAveraging. 7, 16, 20, 21, 35–38

FL Federated Learning. vii, ix, 1, 4–6, 13–17, 19, 20, 22, 35, 36, 38, 41

fMRI functional magnetic resonance imaging. 16

FTL Federated Transfer Learning. 14

HTTP Hypertext Transfer Protocol. 28

IFL Industrial Federated Learning. vii, ix, 1–5, 11, 13, 15–17, 19, 22, 28, 41, 42

IID Independent and Identically Distributed. 38

IoT Internet of Things. 9

JSON JavaScript Object Notation. 32

ML Machine Learning. vii, ix, 1, 3–6, 10, 11, 13, 15, 16, 19, 21, 24, 25, 28, 29, 35, 41

MQTT Message Queuing Telemetry Transport. 17

NIST National Institute of Standards and Technology. 9

REST Representational State Transfer. 17, 19–21

SGD Stochasic Gradient Descent. 7

TCP Transmission Control Protocol. 17

TFF TensorFlow Federated. 16

UDP User Datagram Protocol. 17

URL Uniform Resource Locator. 20, 23, 24, 26, 31

Bibliography

[1] Industrial Edge for production machines. https://new.siemens.com/global/en/products/automation/topic-areas/industrial-edge/simatic-edge.html. Accessed: 2020-12-07.

[2] Industrial Edge, the SIEMENS Edge Computing Platform - Industrial Edge. https://documentation.mindsphere.io/resources/html/industrial-edge/en-US/user-docu/industrialedge.html. Accessed: 2020-11-08.

[3] Siemens Industrial Edge takes the benefits of the Cloud to field level. https://press.siemens.com/global/en/pressrelease/siemens-industrial-edge-takes-benefits-cloud-field-level. Accessed: 2020-12-07.

[4] OpenMined/PyGrid. https://github.com/OpenMined/PyGrid, Nov. 2020. Accessed: 2020-11-22.

[5] M. Aledhari, R. Razzak, R. M. Parizi, and F. Saeed. Federated Learning: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Access*, 8:140699–140725, 2020. Conference Name: IEEE Access.

[6] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov. How to backdoor federated learning. In *International Conference on Artificial Intelligence and Statistics*, pages 2938–2948. PMLR, 2020.

[7] P. Bangert. Smart Condition Monitoring Using Machine Learning. In *SPE Intelligent Oil and Gas Symposium*, Abu Dhabi, UAE, 2017. Society of Petroleum Engineers.

[8] A. N. Bhagoji, S. Chakraborty, P. Mittal, and S. Calo. Analyzing federated learning through an adversarial lens. In *International Conference on Machine Learning*, pages 634–643. PMLR, 2019.

[9] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, H. B. McMahan, T. Van Overveldt, D. Petrou, D. Ramage, and J. Roselander. Towards Federated Learning at Scale: System Design. *arXiv:1902.01046 [cs, stat]*, Mar. 2019. arXiv: 1902.01046.

[10] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical Secure Aggregation for Federated Learning on User-Held Data. *arXiv:1611.04482 [cs, stat]*, Nov. 2016. arXiv: 1611.04482.

[11] W. Dai, H. Nishi, V. Vyatkin, V. Huang, and Y. Shi. Industrial Edge Computing: Enabling Embedded Intelligence. *IEEE Industrial Electronics Magazine*, 13:48–56, Dec. 2019.

[12] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[13] A. Guha Roy, S. Siddiqui, S. Pölsterl, N. Navab, and C. Wachinger. Braintorrent: A peer-to-peer environment for decentralized federated learning. *arXiv e-prints*, pages arXiv–1905, 2019.

[14] T. Hiessl, D. Schall, J. Kemnitz, and S. Schulte. Industrial Federated Learning – Requirements and System Design. *arXiv:2005.06850 [cs]*, May 2020. arXiv: 2005.06850.

[15] H. Jia, F. J. Xi, A. Ghasempoor, and A. Dawoud. A tolerance method for industrial image-based inspection. *The International Journal of Advanced Manufacturing Technology*, 43(11-12):1223–1234, 2009.

[16] E. b. P. Kairouz and H. B. McMahan. Advances and Open Problems in Federated Learning. *Foundations and Trends® in Machine Learning*, 14(1), Mar. 2021. Publisher: Now Publishers, Inc.

[17] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[18] F. Li, J. Fröhlich, D. Schall, M. Lachenmayr, C. Stückjürgen, S. Meixner, and F. Buschmann. Microservice Patterns for the Life Cycle of Industrial Edge Software. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–11, Irsee Germany, July 2018. ACM.

[19] L. Li, Y. Fan, M. Tse, and K.-Y. Lin. A review of applications in federated learning. *Computers & Industrial Engineering*, 149:106854, 2020.

[20] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith. Federated Learning: Challenges, Methods, and Future Directions. *IEEE Signal Processing Magazine*, 37(3):50–60, May 2020. Conference Name: IEEE Signal Processing Magazine.

[21] S. A. Magid, F. Petrini, and B. Dezfouli. Image classification on iot edge devices: profiling and modeling. *Cluster Computing*, pages 1–19, 2019.

[22] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[23] L. D. Medus, M. Saban, J. V. Francés-Víllora, M. Bataller-Mompeán, and A. Rosado-Muñoz. Hyperspectral image classification using cnn: application to industrial food packaging. *Food Control*, page 107962, 2021.

[24] P. Mell, T. Grance, et al. The nist definition of cloud computing. 2011.

[25] M. Satyanarayanan. The Emergence of Edge Computing. *Computer*, 50(1):30–39, Jan. 2017.

[26] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, Nov. 1979.

[27] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos. Challenges and Opportunities in Edge Computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 20–26, Nov. 2016.

[28] Q. Yang, Y. Liu, T. Chen, and Y. Tong. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(2):1–19, 2019.