Communication-Efficient Vertical Federated Inference with Hybrid Federated Neural Networks

RESEARCH PROJECT

Author:
Denise-Phi Khuu
Matr. Nr.: 52354

Supervisors:
Prof. Dr.-Ing Stefan Schulte and Prof. Dr. Olaf Landsiedel

Institute for Data Engineering and Institute for Networked Cyber-Physical Systems

Blohmstraße 15
21079 Hamburg
Germany

Juni 2025

Declaration

I hereby declare that I have written this thesis independently and that I have not made use of any aid other than those acknowledged in this thesis. I further declare that neither this thesis nor any other similar work has been previously submitted to any examination board.

Hamburg, June 10, 2025

Denise-Phi Khuu

Abstract

In Vertical Federated Inference (VFI), multiple clients with vertically separated data intend to collectively use a pre-trained model under the coordination of a server. Traditional inference pipelines require each client to send their raw data, such as images, video, or audio, to a centralized cloud server causing high latency and energy costs. To mitigate these issues, VFI shares responsibility for model executions between a server and multiple clients. To share those responsibilities, high communication frequencies between a server and clients are still often necessary. Since servers and clients usually communicate over wireless networks with fixed bandwidth constraints, data transmission becomes a bottleneck. This highlights the need for communication-efficient VFI. Current research in communication-efficient VFI focuses on compression methods to improve communication costs, while the question of client-side request coordination has remained underexplored. We therefore investigate the following research question: Is VFI always necessary, or can communication cost be reduced by selectively requesting additional support from a server only when necessary? To answer this question, we introduce a new structure called Hybrid Federated Neural Network (HF -NN) that enables a client-side coordination of requests to selectively switch between local and remote models to balance performance and communication costs. Additionally, we show that vertical separations of data can introduce ambiguity between classes for local clients. This can limit the performance of isolated local models.

Contents

Acronyms

ANN Artificial Neural Network 4–6, 8, 9

BNet BranchyNet 8, 10, 21

C-VFL Compressed Vertical Federated Learning 17
CI Collaborative Inference 1, 12, 19
CL Centralized Learning 3, 10, 11, 19, 20

DisL Distributed Learning 10, 11

EF-VFL Error Feedback Compressed Vertical Federated Learning 17
EI Edge Intelligence 1

FI Federated Inference 1, 2, 14, 15
FNN Federated Neural Network 8, 9, 11, 14

GD Gradient Descent 6–8, 10

HF -NN Hybrid Federated Neural Network v, 2, 3, 9, 18, 20, 23, 29, 30, 32, 36, 39, 41
HL Hybrid Learning 10
HNN Hybrid Neural Network 8, 10, 22

i.i.d independently and identically distributed 20

LESS-VFL Local Communication Efficient Group laSSo For Vertical Federated Learning 17

ML Machine Learning 1–4, 10

NN Neural Network 3–5, 8, 18, 20

RTT Round Trip Time 13, 20

SL Split Learning 1, 8, 11
SNN Split Neural Network 8, 9
SotA State-of-the-Art 12

T-VFL Tunable Vertical Federated Learning 17

VFI Vertical Federated Inference v, 1, 17, 29, 39, 41
VFL Vertical Federated Learning 1, 11, 17, 27

Chapter 1

Introduction

In Collaborative Inference (CI), edge and cloud components need to be coordinated to collectively use a pre-trained model [1, 13]. A collaborative setting includes edge devices such as multiple autonomous vehicles and a coordinator such as a traffic service in the cloud [11]. All participants have the common goal of estimating traffic conditions using different sensors [11]. As another example, we can imagine that multiple hospitals want to deploy a diagnostic model collaboratively and each hospital provides information on different medical tests for the same patient [13]. To execute those models, coordinators are necessary to handle communication between those clients and to connect components. Such inter-component inferences are challenging in practice due to static resource limitation and changing network conditions [20]. Ng et al. named, among other things, network variability, node failures, thermal conditions, energy constraints, and workload fluctuations as challenges due to dynamic conditions [20].

Additionally, typical CI requires edges to upload their data, such as images, video, or audio, often through wireless networks to a centralized cloud server, which can lead to high latency and energy cost [26]. Data transmission represents a key performance bottleneck for systems with a high inference frequency or data volume. This has motivated research on shared computational responsibility between edge and cloud components to facilitate offloading of computations to the edge [26]. The distribution of computations and intelligence to the edge is often referred to as Edge Intelligence (EI) [26]. EI can be divided into two subcategories, namely edge training and edge inference. Edge training aims to embed the model training process directly within edge systems, utilizing approaches such as Vertical Federated Learning (VFL) [26] and Split Learning (SL) [23]. In contrast, edge inference focuses on the deployment and execution of pre-trained Machine Learning (ML) models to edge nodes. We call inferences which require the orchestration of computational power in multiple edges Federated Inference (FI) and we refer to settings in which multiple edges have the same target, but own different features as Vertical Federated Inference (VFI) [13]. Current research in communication-efficient VFI focuses on compression methods to improve VFI communication costs, while the topic of client-side coordination of requests has remained underexplored [32]. We therefore investigate the research question: Is VFI always necessary, or can communication cost be reduced by selectively requesting additional support from a server only when necessary?

As an answer to this question, we introduce a new structure called Hybrid Federated Neural Network (HF -NN) that enables a client-side coordination of requests to selectively switch between local and remote models to balance performance and communication costs. This model learns a routing strategy using a proxy that estimates the likelihood that the local model misclassifies and the global model correctly classifies a training sample. We show that only $\sim 30\%$ remote inferences are necessary to achieve the best local performance for the MNIST dataset and only $\sim 20\%$ are required to achieve the best local-only performance for the Fashion MNIST dataset for all clients. Using this model, we can save $\sim 15\%$ on Fashion MNIST and $\sim 10\%$ on MNIST in communication cost, if a reduction of $\Delta acc = -1\%$ accuracy points is acceptable. Our HF -NN V2 reduces the data volume of the local model output by 34.7% and simultaneously achieves a slight increase in average performance of $\Delta acc = 0.15\%$ accuracy points in MNIST compared to a remote model. For Fashion MNIST, we detect a loss of $\Delta acc = -1.25\%$ accuracy points with the same savings demonstrating a trade-off between performance and communication cost savings. Last but not least, we show that vertical separations of data can introduce ambiguity between classes for local clients, i.e., an isolated client cannot distinguish between certain classes and needs access to additional data to make class distinctions. This can limit the performance of isolated local models.

This work is structured as follows. In Chapter 2, we explain the technical background necessary to understand the model architecture. In addition, this chapter introduces the notation and mathematical description of the relevant ML task. In Chapter 3, we introduce recent studies in communication-efficient FI to underline the distinction of our work and to position our contribution. In Chapter 4, we introduce the HF -NN architecture, the training process, and the inference protocol. In Chapter 5, we explain the experimental design including the evaluation metrics and in Chapter 6, we evaluate the models. In Chapter 7, we answer the research question and discuss model limitations and open research questions. In Chapter 8, we conclude our research and introduce possible future directions.

Chapter 2

Technical Background

In this chapter, we introduce the technical background needed to understand the HF - NN model structure and the problem it solves. We start with the introduction of the ML task and the collaborative goal. We continue with different architectural concepts in Neural Network (NN) design and conclude this chapter introducing the used learning method, namely Centralized Learning (CL).

2.1 Multi-View Image Classification

In multi-view image classification, multiple *agents* or *clients* each observe only a subset of the complete input. For image-based tasks, this corresponds to partitioning an image into distinct regions and assigning each region to a different client. The complete input is then reconstructed by concatenating the local observations from all clients.

2.1.1 Classification

In classification problems, we define an input space $X \subseteq \mathbb{R}^{d_X}$, where d_X represents the input dimension, and an output space

- $Y = \{0, 1\}$ for binary classification and
- $Y = \{0, 1, 2, ..., m-1\}$ for multi-label classification with m labels.

We refer to the correct answer as *ground truth*. We use y to denote the true class of x and $\hat{y}_F$ as the predicted label of the input x by model F. The objective is to train a model $F : X \to Y$ which maps an input $x \in X$ to the correct target $y \in Y$. The investigated are trained based on supervised learning approaches, which uses a prepared dataset to adjust some parameters to solve the task. Thus, we need a correctly labeled data set $D = \{x_i, y_i\}_{i=1}^{n}$ consisting of input-output pairs.

2.1.2 Multi-View Classification

Let C be the set of clients, where each client $c_i \in C$ owns a local observation $x^{(i)} \in X_i$. The complete input $x \in X$ is defined as the concatenation of the local views:

$$x = \bigoplus_{i=1}^{|C|} x^{(i)},$$

where $\bigoplus$ denotes vector concatenation. The input-output pairs are drawn from a labeled dataset $D = \{(x_j, y_j)\}_{j=1}^{n}$, where n is the dataset size.

The objective is to train a global model $F : X \to Y$ that maps an concatenated input x to the correct label y. This global model may consist of submodels, where each client c_i owns a local submodel F_i responsible for processing X_i.

2.1.3 Multi-View Image Partitioning

In this work, the MNIST [7] and Fashion MNIST [36] datasets are used for the classification task. Both datasets consist of grayscale images of size 28×28, with a single color channel ($d = 1$). To simulate a multi-view setting, each image is divided into four non-overlapping quadrants, where each client receives one quadrant. Thus, each client's view consists of a 14×14 pixel patch, corresponding to 25% of the full image. Since each client observes only part of the input, their data must be aligned across samples to enable joint inference. All clients process synchronized observations corresponding to the same original image.

(a) MNIST **(b)** Fashion MNIST

Figure 2.1: Data separation of MNIST and Fashion MNIST into four quadrants to simulate a multi-view classification problem. Each client receives one quadrant as their local view.

2.2 Artificial Neural Network

Our solution falls into the class of Artificial Neural Network (ANN) models. ANNs are ML structures that imitate the inner workings of biological neurons [9]. They receive a lot of attention because of their versatility and performance in different applications [16]. ANNs are used for example in AlexNet for Image Recognition [16] or in GPT models for Text Generations [25]. We start by introducing the smallest unit of a NN and then focus

on the learning process. Last but not least, we explain important model architectures. In this work, we use the notation introduced by Zemke to describe the networks [34].

2.2.1 Neurons and Layers

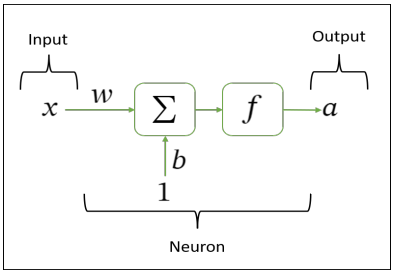


Figure 2.2: Illustration of a general neuron with bias b, weight w, activation function f, input x and neuron output a (Source: [own representation from [14] based on Hagen et al.[9]])

Figure 2.2 shows a visualization of a neuron with weight w, bias b and activation function $f : \mathbb{R} \to \mathbb{R}$. The neuron output a is calculated with the given parameters and input value x as follows:

$$a = f(wx + b) \tag{2.1}$$

If we concatenate and stack multiple neurons into connected layers, we call the resulting model an ANN [9]. The most important layer types for this work are fully connected and convolutional layers.

Fully Connected Layer

The simplest form of NN layers are the fully connected layers [34]. Fully connected layers are defined as

$$a_i := x, \quad z_i := W_i a_i + b_i \quad a_{i+1} := \alpha_i(z_i) \tag{2.2}$$

with x being the layer input, W_i the weights, b_i the biases, and α_i the activation function. They are prone to overfitting due to the amount of trained parameters and therefore another layer was introduced for high-dimensional input dimensions called convolutional neural networks [34, 35].

Convolutional Layer

The core of the convolutional layer is a set of trainable filters also known as kernels [34, 35]. The output is computed by convolving a receptive field-sized image patch with those filters as shown in Figure 2.3. Convolutional layers reduce the number of trainable parameter by using filters and regional connected neurons. The convolutional layer is defined as

$$a_i := x, \quad z_i := (a * f)_i + b_i \tag{2.3}$$

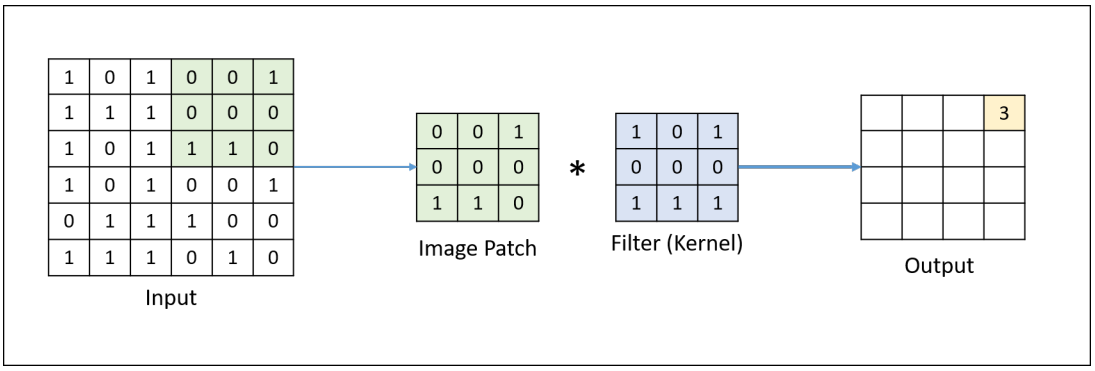


Figure 2.3: A illustration of a convolutional layer with one filter of size 3x3 (Source: [own representation from [14] based on Li et al. [35]])

where x denotes the layer input, b_i the biases and f the convolutional filter.

However, to solve the classification problem with ANNs, the filters, weights, and biases need to be trained to reach a predefined goal. This is done by adjusting the parameters of the neural layers in a self-learning process that is typically based on steepest Gradient Descent (GD) [28].

2.2.2 Gradient Descent

To understand, how this model learns, let us consider a scenario with a classification model F with m different classes. To train the model, a loss function is necessary to define a quantitative measure of the objective.

For the following, we assume that the labels are one-hot encoded and therefore can be viewed as class distributions. A typical loss function used to quantify the distance between the predicted distribution and the correct distribution is called *Cross Entropy Loss* [35]. Let us assume that we use for the last layer the softmax activation function given by

$$p_{i,j} = \frac{e^{a_{i,j}}}{\sum_{k=1}^{m} e^{a_{i,k}}}, \tag{2.4}$$

where $a_{i,j}$ is the logit for class j of sample i and m being the number of classes. This squeezes the values of the last layer from our model between zero and one and introduces probabilistic properties [35]. The average loss between the model outputs and the one-hot-encoded labels can be expressed as the average negative log-likelihood given by

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} -\log p_{i,y_i} \tag{2.5}$$

where N is the size of the training set and y_i is the actual label [35].

Putting both equations together, we derive that Cross Entropy Loss can be expressed as

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}\left[-a_{i,y_i} + \log\left(\sum_{j=1}^{m} e^{a_{i,j}}\right)\right] \tag{2.6}$$

Thus, the better the prediction of our model, the smaller is the distribution distance between the labels and the predicted probabilities [35]. Similarly, for binary classification tasks, it is common to use the sigmoid activation given by

$$p_i = \frac{1}{1 + e^{a_i}} \tag{2.7}$$

as activation for the last layer for the sample i with $p_i \in [0, 1]$ [34].
It is paired with Binary Cross Entropy Loss, which is defined as

$$\mathcal{L}(\mathbf{y_i}, p_i) = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(p_i) + (1 - y_i)\log(1 - p_i)\right) \tag{2.8}$$

where N is the size of the training set and y_i is the actual binary label to quantify the distribution distance [34].

Given these loss functions, we want to find an algorithm to locate a minimum numerically [28]. GD is usually used to find such a minimum. The algorithm starts at a random point in the function using random initialization [28]. We need to adjust the parameters so that it progresses towards the minimum. The negative of the gradient points in the direction of the steepest descent, i.e., the direction in which the function decreases most rapidly [28]. Thus, if we take a step in that direction, we can get closer to the minimum. This step is commonly known as the learning rate η [9, 28]. As seen in Algorithm 4.1, to find the minimum of L, we adjust the parameters W in the direction of the negative gradient with respect to a loss function L and the batch $B_j \subseteq D$ [28]. The meta parameter *epoch* defines the number of times we go through all batches to train the model. For large datasets, it is more efficient to use smaller *batches* for each parameter update instead of the whole dataset D and we refer to the process as mini-batch gradient descent[28].

Algorithm 2.1: Mini-Batch Gradient Descent Algorithm for Model Training

Require: a training set $D = \{x_i, y_i\}_1^n$ divided into disjoint batches $B = \{B_j \mid B_j \subseteq D, \bigcup_j B_j = D\}$

```
function Train()
    Initialize a CNN model F with random parameters W_0,
    a loss function L, learning rate η and number of epochs E
    for epoch e ∈ E do
        for batch B_j ∈ B do
            W_e ←W_{e−1} − η∇L(W_{e−1}, B_j)        ▷ parameter updates
        end for
    end for
    return W_E
end function
```

Backpropagation

As mentioned in section 2.2.2, we need the gradient $\nabla L(W_{e-1}, B_j)$ for the parameter adjustment. The calculation of derivatives is usually based on backpropagation [9, 28]. Backpropagation uses the chain rule to recursively calculate the gradient. The chain rule defines how the composition of functions can be sectioned and chained to calculate the partial derivatives of the original function [9, 28].

By combining backpropagation with GD with an appropriate loss function, we can train numerous ANNs with different layers making it flexible and versatile [2, 16, 25, 29].

Adam Algorithm

GD has two groups of problems due to non-convexity and high dimensionality [34]. Furthermore, GD can be confronted with unwanted oscillations resulting in slow convergence in ravine-shaped functions, i.e. sections where the surface curves sharply in one direction and only slightly in the other [28, 34]. In addition, a suitable learning rate is necessary and it is often advantageous to adjust the learning rate while training the model. If the learning rate is too high, there is a risk of fluctuating over the lowest points and if it is too small, it can lead to slow convergence [28, 34]. To overcome these difficulties, there are two approaches to adjust the algorithm, namely momentum and adapting the learning rate [28, 34]. With momentum, we control how much of the previous computed gradient is added to the current adjustment of the parameters. Thus, oscillating adjustments cancel each other out, whereas contributions in the same direction sum up. This can speed up the learning process [34]. Empirical experiments indicate that it is better to use high learning rates at the beginning of GD and decrease the learning rate when the function approaches a minimum [34]. RMSprop tracks the applied adjustments and decreases the learning rate based on monotonically increasing accumulated squared gradients, continuously [28, 34].

Adam combines RMSprop with momentum. With decay parameters β_1 and β_2, we control the influence of momenta on gradients and accumulated squared gradients on learning rates [28, 34]. We use Adam Optimizers in our training process.

2.2.3 Neural Network Architecture

Due to its flexibility, different ways to stack neuron layers emerged to handle different tasks and deployment methods. In this section, we introduce different architectural concepts, namely Split Neural Network (SNN), Federated Neural Network (FNN) and Hybrid Neural Network (HNN) and BranchyNet (BNet). Those concepts explain how the layers need to be structured to enable different deployment methods and tasks.

Split Neural Network

SNNs were introduced by Poirot et al. paired with Split Learning (SL) as a way to train NNs without sharing raw data with a centralized server [23]. SNNs can increase data privacy and reduce communication cost if the dimension of the sent activation is smaller than the input dimension. In SNN, the model is structured such that the network layers can be deployed separately (See Figure 2.4). As seen in Figure 2.4 the first neural layers

are deployed on the client and the last layers, which infer the target, are deployed on the server.

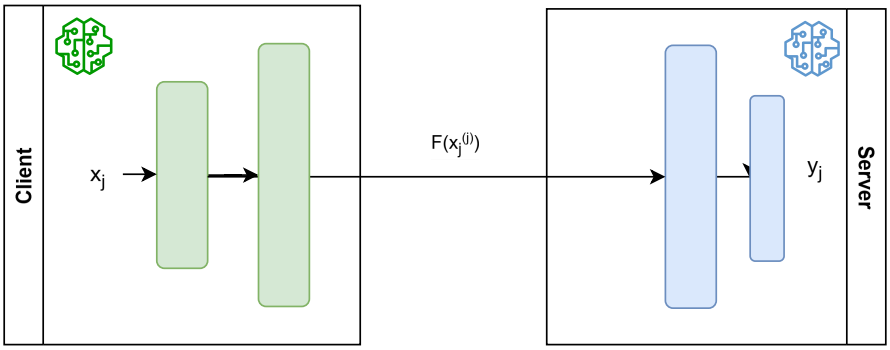


Figure 2.4: Split Neural Network for hybrid deployment of model

Federated Neural Network

FNNs are needed if multiple clients intend to collaboratively use a trained model without sharing their own raw data. It is used in scenarios in which data are generated in vertically separated silos, but due to bandwidth constraints or security reasons the raw data cannot be shared with a centralized server. To use ANNs in distributed settings, ANNs are structured such that submodels can be deployed to each client individually (See Figure 2.5). Based on SNN, Split FNN emerged as a subclass of FNNs. Split FNNs split the model such that each client receive their own initial layers. For inference, each client computes the output of their first layers and sends them to a server. The server has the responsibility to concatenate the intermediate activations to infer the target using their own trainable weights (See Figure 2.5). This concept is incorporated into HF -NN architectures to enable shared deployments between multiple clients and a server in the cloud.

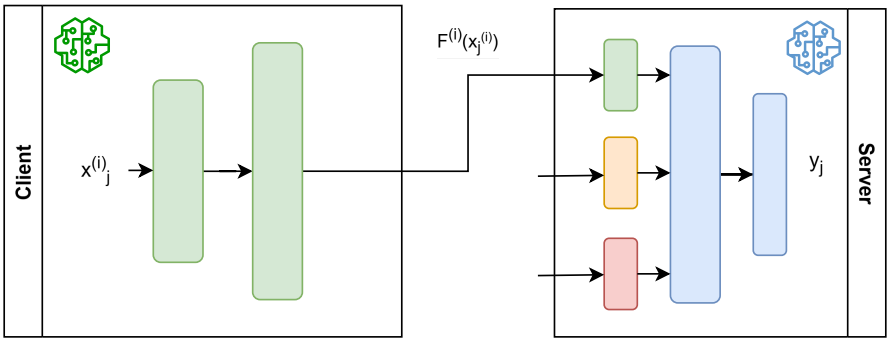


Figure 2.5: Split Federated Neural Network for hybrid deployment

Hybrid Neural Network

HNNs paired with Hybrid Learning (HL) were introduced by Kag et al. to reduce the communication costs and energy consumption of cloud-only deployments [2]. The main idea of HNN is to save communication costs by selectively routing hard-to-predict samples to the cloud while using the local predictor for easier examples [2]. The suggested hybrid model consists of a cheap-base (b) and routing (r) component running on a micro-controller and an expensive global model component (g) running on the cloud (See Figure 2.6). Local and global predictors are trained to solve the same target objective. In contrast, the router has the objective of finding samples that are falsely predicted by the local model and correctly predicted by the server [2]. This router objective is used as a proxy to distinguish hard from easy examples [2].

BranchyNet

BNet by Teerapittayanon were proposed to enable faster inference through early exit [29]. The general structure of BNet includes multiple branches that each consist of one or more layers that lead to exit points. Furthermore, a BNet is trained to solve a joint optimization problem based on the weighted sum of multiple loss functions with respect to branch exits.

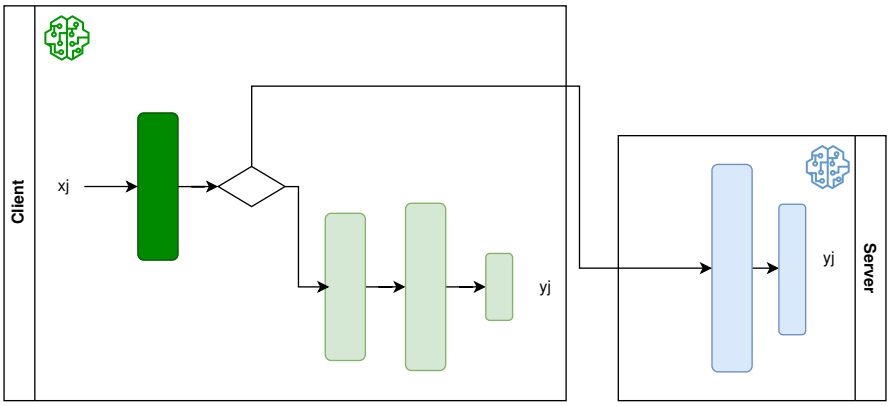


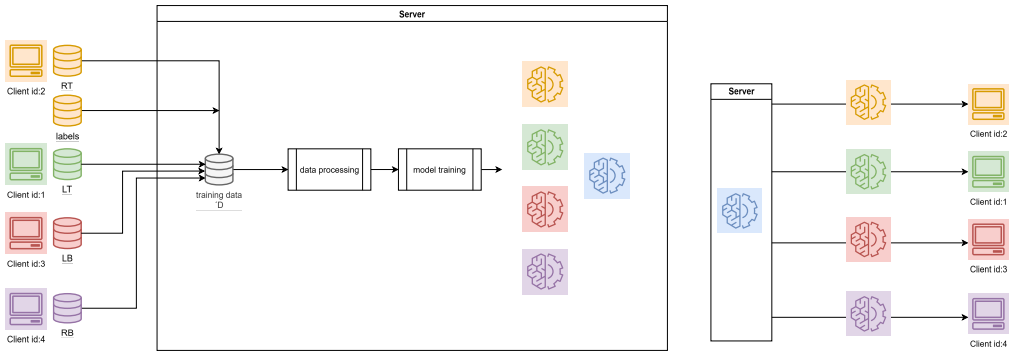
Figure 2.6: Hybrid Deployment Neural Network with (dark green) router r, (light green) base model b and (blue) expensive global model g

2.3 Centralized Learning

Independently, from the model structure, to train the model, the data need to be aligned and the labels need to be available to enable GD paired with backpropagation. In practical ML settings, the necessary data is produced in hundreds to millions of different user devices [15]. Training a model in a distributed setting is what we call Distributed Learning (DisL). A conventional approach is to collect the data in a centralized data center and then use the aggregated data inside a data pipeline to train the model [1]. This approach is generally called CL.

Thus, CL needs a centralized server to coordinate the training process and the data flow of multiple user devices. We refer to those user devices as *clients* or *edges*. To enable supervised learning, we therefore need clients that provide the labels for the training set and clients who provide additional features. Consider a CL system with a server and $|C|$ different clients $c_1, c_2 \dots c_{|C|} \in C$ with respective data sets $D^{(1)}, D^{(2)}, \dots, D^{(|C|)}$ of size n. They want to collaboratively train a classifier model $F : X \to Y$ mapping aligned images to their corresponding labels by minimizing a loss function L as described previously in Algorithm 4.1. A server creates an aggregated dataset $D = D^{(1)} \cup D^{(2)} ... \cup D^{(|C|)}$ after data collection. We use the symbol $W_{(F,e)}$ to represent the model parameters for epoch e and model F.

In this work, we use CL to train the model F including the submodels $F^{(i)}$ for each client i. SL and VFL are usually preferred for DisL due to their data privacy properties [1]. Secure SL and VFL would need additional components to align the samples and encrypt the sent data safely [18], but this is out-of-scope for this work. We assume that the clients are allowed to send their training data to the server and that the alignment is already completed. Because in CL it is possible to clean and rearrange the dataset in processing pipelines, we will not consider the possibility of missing features in the training set. Figure 2.7a visualizes a typical CL pipeline with data collection, processing and model training. After model training, the submodels are distributed to the right clients, respectively (See Figure 2.7b). If the model is trained and the submodels are distributed to the clients, the system can infer targets on unseen data without sending raw data to the server.



(a) CL with data collection, data processing and model training pipeline for FNNs. Client RT provides next to the image data additionally the sample labels.

(b) Submodel Distribution after CL

Figure 2.7: Overview of Centralized Learning

2.4 Collaborative Inference

Previously, we have discussed, how the model can be structured and trained towards a predefined goal using CL. Depending on how the model is deployed, there are different protocols for inference, that the participants need to follow, to compute the target. In the next section, we explain which protocol is suitable for which deployment method to

enable CI. For the next section, we assume that every inference starts with an *active* client, who is interested in the target of x_j where j is the sample index. We call clients that only provide additional features *passive* [18].

On-Cloud Inference

For CI, it is common practice to deploy State-of-the-Art (SotA) models in the cloud [17], considering that the cloud provides sufficient resources for training and inference that are usually not available on edge devices. Figure 2.8 illustrates the following protocol:

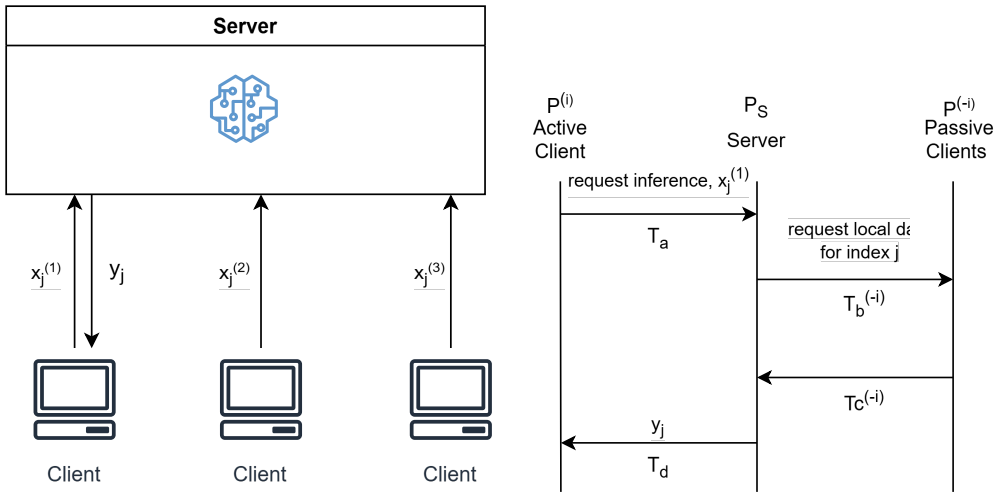
Protocol 2.1. On-Cloud Inference

Inputs: Active client sends index j and input $x_j^{(i)}$.

Protocol:

1. Active client $\rightarrow$ server: $(j, x_j^{(1)})$.
2. Server $\rightarrow$ passive clients: "send your $x_j^{(i)}$".
3. passive clients $\rightarrow$ server: Send local data $x_j^{(i)}$
4. Server computes $x_j = \bigoplus_i x_j^{(i)}$, then $\hat{y}_j = F(x_j)$.
5. Server $\rightarrow$ active client: send $\hat{y}_j$.

Although this deployment model usually has the highest accuracy because it uses the SotA model, it is usually coupled with a higher latency due to slow network connections or high input dimensions of the clients and higher computational cost due to the larger model size [2, 20].



(a) On-Cloud deployment of a global model

(b) Communication Flow of On-Cloud Inference

Figure 2.8: On-Cloud Inference with the active client x_1

Round Trip Time Let the active client be x_i and all passive clients be x_{-i}. Let $P^{(i)}$ be the local processing of the data before sending the request (See Figure 2.8b). Let T_a be the network latency generated by requesting an inference from a server and T_d be the latency time of sending the prediction from the server to the active client (See Figure 2.8b). Let T_{-i} be the time until the last local dataset x_{-i} arrives. We assume that the request is done asynchronously. Thus, the waiting time is equal to the slowest response of all passive clients x_{-i} (See Equation 2.9). Furthermore, let P_s be the processing time on the server to calculate the prediction (See Figure 2.8b).

$$T_{-i} := \max(T_b^{(k)} + P^{(k)} + T_c^{(k)})_{k \in x_{-i}} \tag{2.9}$$

From Protocol 2.4, we derive that the Round Trip Time (RTT) from requesting a prediction from a server to receiving an answer is

$$RTT_{OnCloud} := T_a + T_{-i} + P_s + T_d \tag{2.10}$$

We call models used for On-Cloud inferences *remote-only* models.

On-Device Inference

Even though On-Device inference is not collaborative, we add this to this section for completeness. To completely eliminate communication latencies, On-Device inference is a practical alternative because the whole model is deployed on the client. Each client is self-sufficient and only relies on their own data (See Figure 2.9). Consequently, the input space for this model is limited to the view of the client. Thus, the client loses the option to add valuable information that would be available to a collaborative system. In this case, we trade accuracy for faster inference time and less computational cost. As seen in Protocol 2.4, the client does not request information from a server. Thus, the RTT is

$$RTT_{OnDevice} := 0 \tag{2.11}$$

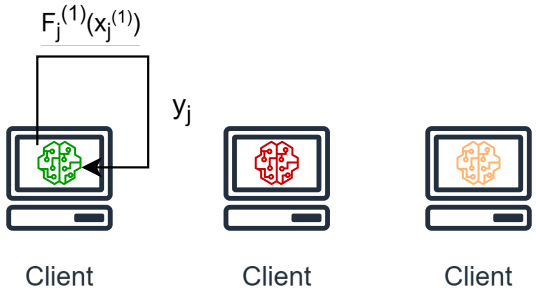


Figure 2.9: On Device Inference

Protocol 2.2. On-Device Inference

Inputs: Active client retrieves input $x_j^{(i)}$.

Protocol:

1. Active client computes $\hat{y}_j^{(i)} = F^{(i)}(x_j^{(i)})$.

Since $RTT_{OnDevice} := 0$, the inference time only depends on the local processing time. We call models used for On-Device inferences *local-only* models.

Federated Inference

As stated in Chapter 1, with FI, we share computational responsibilities between edge and cloud components [26]. The submodels are deployed on the edge and the model outputs are concatenated by the server in the cloud to infer the target [13, 30]. Figure 2.11 visualizes how FI can be realized using Split FNN. The key distinction compared to On-Cloud inference is that instead of sending the raw data, the clients send the output of their local model for the aggregation (See Protocol 2.4). Note that the sent activations can be arbitrarily large. Thus, the communication efficiency depends on the network design and the size of $F^{(i)}(x_j^{(i)})$. We call models used for FIs *federated* models.

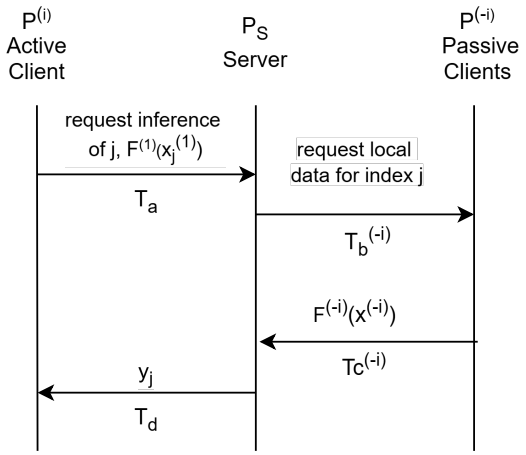


Figure 2.10: Communication Flow of Federated Inference

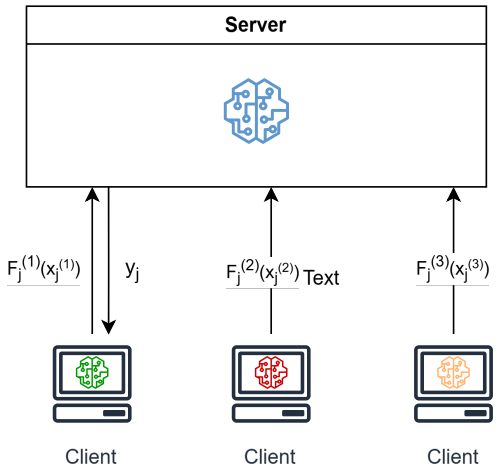


Figure 2.11: Split FedNN with hybrid deployment

Protocol 2.3. Split Federated Inference

Inputs: Active client sends index j and input $x_j^{(i)}$.

Protocol:

1. **Active client** $\rightarrow$ **Server**: Send $(j, F^{(i)}(x_j^{(i)}))$, the local activation.
2. **Server** $\rightarrow$ **Passive clients**: Broadcast request for activations corresponding to the index j.
3. **Passive clients** $\rightarrow$ **Server**: Send $F^{(-i)}(x_j^{(-i)})$, their local activations for index j.
4. **Server**: Aggregate all activations:

$$f_j = \bigoplus_i F^{(i)}(x_j^{(i)})$$

   Then compute the output:

$$\hat{y}_j = F(f_j)$$

5. **Server** $\rightarrow$ **Active client**: Send prediction $\hat{y}_j$.

Round Trip Time Similar to Equation 2.10 for On-Cloud inference, we have for FI the following RTT based on Protocol 2.4 and Figure 2.10:

$$T_{-i} := max(T_b^{(k)} + P_F^{(k)} + T_c^{(k)})_{k \in x_{-i}} \tag{2.12}$$

$$RTT_{Fed} := T_a + T_{-i} + P_{F,s} + T_d \tag{2.13}$$

where $P_F^{(k)}$ represents the processing time to infer a submodel $F^{(k)}$ and $P_{F,s}$ denotes the computation time for the remaining layers of model F.

Chapter 3

Related Work

Communication cost poses a challenge in production VFL systems, due to the high number of required message exchanges for a single inference (See Protocol 2.4). Additionally, Liu et al. argue that network variability, high volume of encrypted data, and long geographical distances between server and clients in production VFI make coordination a system bottleneck [18]. Current research focuses on reducing communication costs to mitigate these issues and enable production-grade VFI system that require high communication frequencies like autonomous driving [11].

Compression is the most commonly used approach in VFI and VFL [3, 4, 6, 12, 31]. Castiglia et al. introduced Compressed Vertical Federated Learning (C-VFL) which applies quantization and sparsification methods such as scalar quantization, vector quantization, and top-k sparsification to sent vectors to reduce communication costs [4, 6]. Other studies suggest using feature extraction methods such as Principal Component Analysis [3, 12] and Autoencoders [8, 12] to reduce the dimension of the input space to make training and inference more efficient. Valdeira et al. proposes Error Feedback Compressed Vertical Federated Learning (EF-VFL) a compression method that keeps track of the compression error and keeps it in a feedback loop while training the split model [31].The results indicate that Error Feedback Compressed Vertical Federated Learning (EF-VFL) improves convergence without increasing the communication cost, but this is not directly applicable to inferences of independent inputs [31]. Another common approach uses feature filtering to reduce the amount of sent data for instance Local Communication Efficient Group laSSo For Vertical Federated Learning (LESS-VFL) [5].

Other research focuses on knowledge distillation to completely eliminate inter - component communication [19, 27]. This method includes other clients in the training process to improve model performance, while the model structure enables each client to infer targets locally after model deployments [19, 27]. Ren et al. suggest, for example, a knowledge distillation method to create a model that operates fully locally [27]. The main idea includes a teacher model trained by all clients using their target and numerous student models trained on the soft targets predicted by the teacher model [27]. We see indication in our experiments that this approach may be limited by the discriminative capacity of local data and could struggle in scenarios involving ambiguous classifications due to incomplete feature information.

Wang et al. propose Tunable Vertical Federated Learning (T-VFL) for a communication -efficient model serving [32]. Similarly, to our approach, they intend to find a way

to selectively switch between local and remote models [32]. In contrast, they propose to fixate the trained global model and train a local model and a discriminator model separately [32]. In contrast, we propose a method that trains all model heads simultaneously and enables shared model parameters, which can reduce the size of the mode. Furthermore, our results indicate that local models can benefit from learning using a shared model base.

Furthermore, their results indicate that there are more opportunities, in addition to compression, feature filtering, and knowledge distillation methods, to save communication costs [32]. These methods reduce the size of activations or representation vectors, but still require the participation of all clients for every inference. In this work, we introduce a new NN structure, namely HF -NN that differentiates easy from difficult samples to coordinate requests to reduce communication costs. This structure can be trained end-to-end and enables shared parameters that introduce beneficial properties for local models.

Chapter 4

Hybrid Federated Neural Network

In this chapter, we present our strategy for reducing communication costs and then detail how the hybrid architecture realizes them.

4.1 System Design

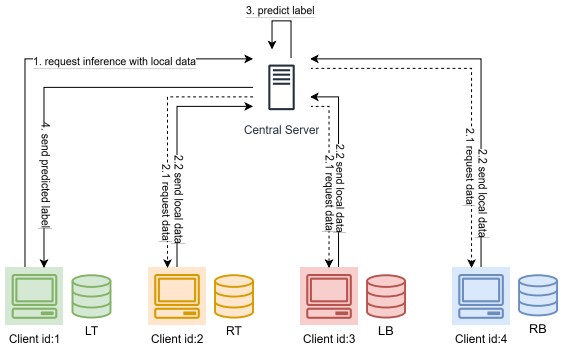


Figure 4.1: CI with four static clients. The arrows show the inference protocol in which Client LT request the prediction of a sample.

The system consists of four clients and an aggregation server. The model is trained using CL, i.e the model parameters are trained in the cloud and then distributed to the clients (See Figure 2.7b). The raw data for inference remain local and only the activations are shared at inference time.

4.1.1 System Assumptions

For simplicity, we assume homogeneous and honest edges that can deploy the same router and convolutional base. We assume that each client can be identified uniquely. This reflects the scenario in which there exist a fixed number of stationary sensors with permanent IP addresses. Moreover, the same requirement can be realized by providing the identifier with every response and request. This ensures that we can always assign the client to a fixed position in the concatenation process. In addition, we have predeter-

mined clients. Thus, the model does not have to scale or handle dynamic client changes. For completeness, we assume that the communication network is stable without network breakdowns.

4.1.2 Data Assumptions

Using the advantages of CL, we assume that the data can be correctly aligned and processed to achieve an independently and identically distributed (i.i.d) training set.

4.2 Solution Approach

As a reminder, we want to find a NN structure that supports communication cost improvements and enables a trade-off between accuracy and communication cost. The idea is to reduce the communication cost

- by decreasing the dimensions or size of $size(F^{(i)}(x_j^{(i)}))$ with $size(F^{(i)}(x_j^{(i)})) <= size(x_j^{(i)})$, which automatically decreases the data volume sent by each client. If the condition $size(F^{(i)}(x_j^{(i)})) <= size(x_j^{(i)})$ is not fulfilled, remote-only models are preferred over federated models from a communication cost perspective, provided that security requirements are not a concern. Furthermore, the size of $F^{(i)}$ has an impact the RTT since the upload and download time depends on it.(See Equation 2.13).
- by increasing the local coverage using an local-only model. Let $R\bar{T}T$ be the average RTT. The average hybrid $R\bar{T}T_{Hybrid}$ is then defined as

$$R\bar{T}T^{(i)}_{Hybrid} = \text{Remote Coverage}^{(i)} R\bar{T}T_{Fed} + (1 - \text{Remote Coverage}^{(i)}) * R\bar{T}T_{OnDevice} \tag{4.1}$$

where Remote Coverage$^{(i)}$ denotes the proportion of the sample sent to the cloud. Since $R\bar{T}T_{OnDevice}$ is always equal to zero, we have

$$R\bar{T}T^{(i)}_{Hybrid} = \text{Remote Coverage}^{(i)} R\bar{T}T_{Fed} \tag{4.2}$$

Thus, by controlling the remote coverage we can reduce the average latency induced by server communication and the average sent data volume.

The intuition behind our idea is similar to the one described by Kag et al. [2]. The idea is that with respect to a classification task, there might exist inputs that are generally easier or more difficult to infer. For easier inputs, it might be possible to use a local model to infer the correct label without requesting additional information; i.e., additional data would not change the answer. We therefore could use a hybrid model to reduce the number of remote inferences. In the next section, we explain how the HF -NN provides these features.

4.3 Model Architecture

The HF-NN consists of four core components, namely a convolutional base (cb), and a local prediction head (lh) and global prediction head (gh) and a head router (r).

The convolutional base consist of convolutional layers. Each prediction head consists of fully-connected layers that are trained on the same task of predicting the target. The router decides which head should be used for the prediction depending on the difficulty of the instance x_j. By training two prediction heads, we introduce the option to early exit the model to use On-Device inference. The local base is trained to solve a joint optimization problem based on the weighted sum of all loss functions similar to BNet which reduces the number of necessary model parameters [29]. To train the router, we use the proxy that estimates the likelihood that the local model misclassifies a training sample and the global model provides the label correctly. This routes difficult samples to the cloud and keeps easy samples on the edge. Thus, providing the option to learn a routing strategy to reduce the communication cost by controlling remote coverage. After model training, the convolutional base, the router head, and the local head are distributed to the respective clients.

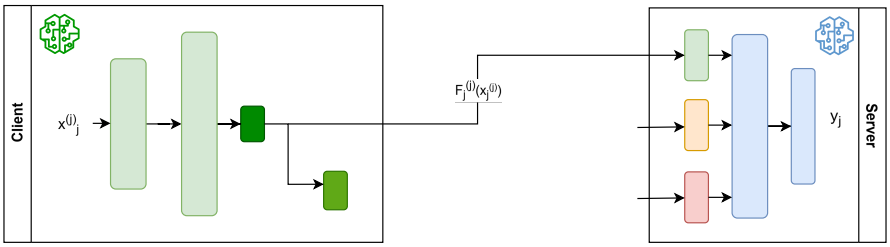


Figure 4.2: Cloud-Edge collaborative inference with shared convolutional base (light green), local prediction head (green), router (dark green) and global prediction head (blue)

Encoder-Only Compression Block As mentioned in Chapter 2.4, the communication efficiency depends on the network design and the size of $F^{(i)}(x_j^{(i)})$. Thus, we need a component which controls the submodel outputs and ensures that $F^{(i)}(x_j^{(i)})$ remains small, i.e $size(F^{(i)}(x_j^{(i)})) <= size(x_j^{(i)})$. In this work, we use a linear compression block which becomes part of the local base. The compression block consists of an average pooling layer and a local linear layer. We add an average pooling layer to reduce the feature maps to a single value and add a linear layer for compression into the desired output dimension. The idea is that by adding a local linear layer, we can optimize the representation of the sent data while reducing the local output size. Note that by adding a single linear layer after the convolutional block, we would increase the number of model parameters, significantly leading to a higher risk of overfitting. This block can be replaced with Autoencoders or similar representation-based compressions [3, 4, 6, 12, 31], but the comparison with other compression methods is out of scope for this project.

4.4 Model Training

We define the evaluation of the model to the last local convolutional layer with $z^{(i)} := F_{cb}^{(i)}(x^{(i)})$. Furthermore, we define the logits of the global model and the local model with $a_{gh}^{(i)} := F_{gh}(z^{(i)})$ and $a_{lh}^{(i)} := F_{lh}(z^{(i)})$, respectively. The logits of the local router

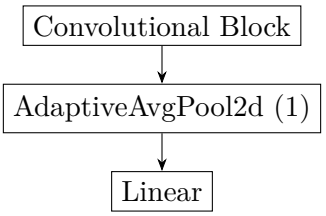


Figure 4.3: Illustration of the encoder-only compression Block

are defined as $a_r^{(i)} := F_r^{(i)}(z^{(i)})$. We use

$$\hat{y}_F := \arg\max_{k \in Y} a_{F,k}$$

to denote the hard decision based on the logits. $a_{F,k}$ is the logits of submodel F for class k.

4.4.1 Loss Functions

As seen in Figure 4.2, the model consists of two prediction heads and a router.

Prediction Heads

The prediction heads lh and gh have the goal as described in Section 2.1.2. Thus, their main goal is to optimize the Cross Entropy Loss to solve the multi-view classification task (See Equation 2.6). By adding the notation of Section 4.3, we derive that the loss functions can be defined as

$$\mathcal{L}_{\mathrm{H}} = \frac{1}{|B|} \sum_{i=1}^{|B|} \left[-a_{i,y_i} + \log \left(\sum_{j=1}^{|Y|} e^{a_{i,j}} \right) \right] \tag{4.3}$$

where $a_{i,j}$ is the logit for class j of sample i, y_i is the index of the correct class for sample i, $|Y|$ is the number of classes, $|B|$ is the batch size and it holds for the heads $H \in \{lh, gh\}$.

Router

Similarly to HNNs [2], we use a proxy to train the router. The idea is that we should only send the samples to the cloud, if the global model is likely correct and the local model is likely incorrect. Thus, the router of client i is trained to find a model $F_r^{(i)} : X^{(i)} \to \{0, 1\}$, which maps the inputs $x^{(i)}$ to 1 if the local model is incorrect and the global model is correct. Thus, we can treat the router objective as a binary classification task with 0 and 1 representing the routing decision, respectively. The labels for the training sets are calculated on batch levels by fixing the model heads $F_{gh}(x^{(i)})$, $F_{lh}(x^{(i)})$ and the convolutional base and mapping the input $x^{(i)}$ with

$$\mathcal{R}(x^{(i)}, y^{(i)}) = \mathbb{1}(\hat{y}_{F_{gh}}^{(i)} = y^{(i)} \cap \hat{y}_{F_{lh}}^{(i)} \neq y^{(i)}) \tag{4.4}$$

The resulting loss function using the Binary Cross Entropy similar to Equation 2.8 can be defined as

$$\mathcal{L}_r(\mathbf{r}, \hat{\mathbf{r}}) = -\frac{1}{|B|} \sum_{i=1}^{|B|} (r_i \log(\hat{r}_i) + (1 - r_i) \log(1 - \hat{r}_i))$$

with $r = \mathcal{R}(x^{(i)})$ being the correct mapping and $\hat{r} = F_r^{(i)}(x^{(i)})$ being the predicted mapping of the router model and $|B|$ being the batch size.

Convolutional Base

Similar to the loss function defined by Teerapittayanon et al. [29] for early exits, the loss function of the convolutional base is the weighted sum of all loss functions and defined as

$$\mathcal{L}_{ch} = \sum_F \alpha_F \mathcal{L}_F$$

where α_F with $F \in \{gh, lh, r\}$ controls the weight of each loss function. It is implemented by adding the gradients of the respective heads prior to the parameter updates with $\alpha_F = 1$.

4.4.2 Training Algorithm

Since the previously introduced model structure, needs multiple loss functions, and prediction heads, we need to adjust the Algorithm 4.1 to enable end-to-end training for HF -NNs. $F_r^{(i)}(x^{(i)})$ depending on $F_{gh}(x)$ and $F_{lh}(x^{(i)})$ introduces cyclic dependencies of these heads. We use a similar approach to alternating optimization introduced by Kag et al. to circumvent the problem of cyclic non-convexity [2]. The algorithm starts by iterating through multiple epochs, which defines how often we iterate through each batch from the training set (See Line 4-5). For each batch B_j, we update the parameters for the local head using backpropagation for each client i (See line 7-9). Thus, we only use from the batch B_j, the view $B_j^{(i)}$ that is relevant to the client i (See line 6). For the following steps, we save the activation of the convolutional base of each client because they are needed to update the other submodels. Since the outputs of the convolutional layers are matrices, we have to flatten the matrices to concatenate them into a single vector. For readability, this step was omitted in the pseudo code (See Algorithm 4.1). Note that we assume that each client can be uniquely identified. Thus, each client has a fixed position in the concatenation. In line 11-13, we concatenate the activations and apply feedforward and backpropagation for the global prediction head. We use the previously saved logits of the global and local model to find the hard prediction for each sample and apply the mapping function 4.4 to create the training set for the router. Afterwards, we apply backpropagation to the router head (See line 16, 17). In line 19, we sum up the gradients of each loss function with respect to $W_{(cb,e-1)}$ and update the base parameters.

4.5 Federated Inference

After model training and submodel distribution, we need a new protocol to enable hybrid inference (See Protocol 4.5). The protocol starts by evaluating the convolutional base (See line 1-2). The base forwards the activations to the router head to decide on the classification head. If the router logit is smaller than the fixed threshold ϵ_r, we use the local classification head for the inference (See line 3-5). Otherwise, we send a request to the server which evaluates the multi-view classification task (See line 6-12).

Algorithm 4.1: Mini-Batch Gradient Descent Algorithm for Hybrid Federated Neural Networks

Require: a training set $D = \{x_k, y_k\}_1^n$ divided into disjoint batches $B = \{B_j \mid B_j \subseteq D, \bigcup_j B_j = D\}$

function Train()
  Initialize a CNN model F with random parameters W_0,
  a loss function L, learning rate η and number of epochs E
  for epoch $e \in E$ **do**
    for batch $B_j \in B$ **do**
      for batch $B_j^{(i)} \in B_j$ **do**
        $z_{(cb,j)}^{(i)} \leftarrow F_{cb}^{(i)}(B^{(i)})$ ▷ Forward Pass of Convolutional Base
        $a_{(lh,j)} \leftarrow F_{lh}^{(i)}(z_{cb}^{(i)})$ ▷ Forward Pass of Local Head
        $W_{(lh,e)} \leftarrow W_{(lh,e-1)} - \eta \nabla_{W_{(lh,e-1)}} L_{lh}$ ▷ Local head parameter updates
      end for
      $z_{(cb,j)} \leftarrow \bigoplus_i z_{(cb,j)}^{(i)}$ ▷ Concatenation of activations
      $a_{(gh,j)} \leftarrow F_{gh}(z_{(cb,j)})$ ▷ Forward pass of global head
      $W_{(gh,e)} \leftarrow W_{(gh,e-1)} - \eta \nabla_{W_{(gh,e-1)}} L_{gh}$ ▷ Global head parameter updates
      for batch $B_j^{(i)} \in B_j$ **do**
        $r_j \leftarrow \mathcal{R}(B_j^{(i)})$ ▷ router proxy
        $a_{(r,j)} \leftarrow F_r^{(i)}(z_{(cb,j)})$ ▷ Forward pass of router
        $W_{(r,e)} \leftarrow W_{(r,e-1)} - \eta \nabla_{W_{(r,e-1)}} L_r$ ▷ router parameter updates
      end for
      $\nabla_{W_{(cb,e-1)}} L_{cb} = \sum_F \nabla_{W_{(cb,e-1)}} \alpha_F L_F$ ▷ combined gradients
      $W_{(cb,e)} \leftarrow W_{(cb,e-1)} - \eta \nabla_{W_{(cb,e-1)}} L_{cb}$ ▷ base parameter updates
    end for
  end for
  return W_E
end function

Protocol 4.1. Hybrid Federated Inference

Inputs: Index j, input $x_j^{(i)}$, router threshold ϵ_r
Protocol:

1. **Active client** evaluates $z_{j,cb}^{(i)} \leftarrow F_{cb}^{(i)}(x_j^{(i)})$
2. **Active client** evaluates $a_{j,r}^{(i)} \leftarrow F_r^{(i)}(z_{j,cb}^{(i)})$
3. **if** $a_{j,r}^{(i)} < \epsilon$
4. **Active client** evaluates $\hat{y}_j \leftarrow argmax(F_{lh}^{(i)}(z_{j,cb}^{(i)}))$
5. **end if**
6. **else**
7. **Active client** $\rightarrow$ **Server**: Send $(j, z_{j,cb}^{(i)})$
8. **Server** $\rightarrow$ **Passive clients**: Broadcast request for activations of index j.
9. **Passive clients** $\rightarrow$ **Server**: Send $z_{j,cb}^{(-i)}$, their local activations for index j.
10. **Server**: Aggregates all activations:

$$z_{(cb,j)} \leftarrow \bigoplus_k z_{r,cb}^{(k)} \quad k \in C$$

Then compute the output:

$$\hat{y}_j \leftarrow argmax(F_{gh}(z_{j,cb}))$$

11. **Server** $\rightarrow$ **Active client**: Send prediction $\hat{y}_j$.
12. **end else**
13. **return** $\hat{y}_j$

Chapter 5

Experimental Design

5.1 Experimental Setup

Datasets

We include two public datasets namely MNIST [7] and Fashion MNIST [36]. They consist of 70000 28×28-pixel images from 10 different classes. From those 70000 images, we use 60000 for training and 10000 for testing. After each epoch, 12000 images are randomly selected from the training set to use them as validation set. To create a vertically separated data environment, we divide those images into four quadrants as seen in Figure 2.1 resulting in 14×14-pixel images.

Models

Hybrid V1 The convolutional base consists of two convolutional layers, a single batch normalization layer, a ReLU layer, and a max-pooling layer (See Figure 5.1). We use this base, because it has shown high accuracy in another paper for On-Cloud and VFL models [33]. Each head additionally includes two linear layers with ReLU activation and a dropout layer. We have included a compression block after the convolutional block to reduce the output dimension to 49. This ensures that the number of bytes transmitted for a single inference by a client is equal to the size of the raw image pixels. This has the advantage that we do not send the raw data to the server and do not have to recompute the activation to maintain the initial communication cost.

Hybrid V2 The second model has a compression block, which has the output dimension of 32. Since this means less bytes per inference sent by a client, this reduces the average communication costs, automatically.

Remote-Only CNN For comparison, we train CNN model with a convolutional block as seen in Figure 5.1, where the raw data are concatenated column-wise. For this architecture, we share model parameters between the raw inputs of different clients. This setting would correspond to a centralized inference setting in which each client sends their raw images for inference.

Local-Only CNN To evaluate the local models, we train a CNN model for each client separately and use the local view as input without data aggregation. This corresponds to the isolated setting in which each client is self-sufficient.

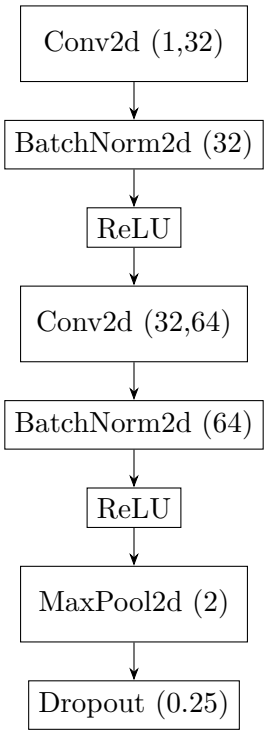


Figure 5.1: Illustration of the convolutional neural network block with two convolutional layers, batch normalization, ReLU activation, and a max pooling layer of the convolutional base.

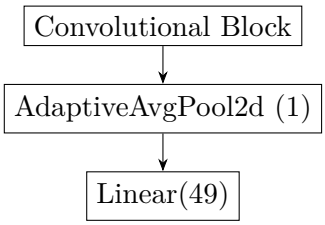


Figure 5.2: Compression Block V1

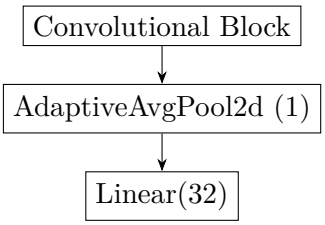


Figure 5.3: Compression Block V2

Figure 5.4: Illustration of the compression blocks to reduce the number of output dimensions at the client-side.

Hyperparameters of the Training Algorithm

We use a learning rate $\eta = 0.001$ and training, validation, and test batch size of 64 as set by default by the model framework PyTorch[21]. Similarly, the meta parameters for the Adam Optimizer are set to the default values $\beta_1 = 0.9$ and $\beta_2 = 0.999$ with $\epsilon = 1e - 8$ [21]. In addition, we use early stopping as a regularization method to stop the training process as losses increase due to overfitting [24]. The early stopper is initialized with patience 10 and $\delta = 1e - 5$ and an epoch upper limit of 50, which has shown for both

datasets a high level of convergence.

Hardware and Software

We run our experiments on a Windows 11 machine with an Intel(R) Core(TM) i9-14900HX processor having 24 cores with a clock speed of 2.20 GHz and 32GB RAM. Additionally, we are using the CUDA platform for parallel processing with a NVIDIA GeForceRTX4080 Laptop GPU which has a memory data rate of 18Gbps and 7424 cores. We use PyTorch [21] as model framework, Numpy for vector and matrix calculations [10] and scikit-learn[22] for the evalutation.

5.2 Evaluation Structure

As a reminder, our initial research question is: *Is VFI always necessary, or can communication cost be reduced by selectively requesting additional support from a server only when necessary?*

Our evaluation therefore needs to show that we train a router with reduced input space to quantify hard-to-classify instances. This will answer the question if VFIs are always necessary. If we can find a routing strategy that does not reduce the accuracy significantly, we can say that for a set of samples the inferences were unnecessary. Furthermore, we need to show that by using Protocol 4.5, we increase the communication efficiency of a vertical federated system.

5.2.1 Evaluation Metrics

We use hybrid accuracy, local coverage and sent data volume to measure the model performance and communication efficiency.

5.2.2 Hybrid Accuracy

To measure the HF -NN model performance, we use the hybrid accuracy for the prediction

$$\text{Hybrid Accuracy}^{(i)} = \mathbb{P}(F_r(x) = 0, F_{lh} = y) + \mathbb{P}(F_r(x) = 1, F_{gh} = y)$$

for each $i \in C$

5.2.3 Remote and Local Coverage

Similar to Kag et al. [2] and shown in Equation 4.2, we can use remote coverage as a measure for communication savings. This approach assumes a stable communication network, and we can interpret local coverage as the average communication cost (data volume, latency) reduction achieved by using the local model instead of the remote model. However, this method does not account for latency spikes, disconnections, or bandwidth fluctuations. As these network anomalies can significantly impact actual latency, using remote coverage as a measure can overestimate savings in real-world settings where communication conditions are dynamically changing and more challenging.

$$\text{Remote Coverage}^{(i)} = \mathbb{P}(F_r(x) = 1)$$

$$\text{Local Coverage}^{(i)} = \mathbb{P}(F_r(x) = 0) = 1 - \text{Remote Coverage}$$

5.2.4 Data Volume

For the data volume, we only consider the data the client has to send for the inference, i.e. without the data needed for the inference request and server responses. Depending on the type of data sent, one of the two following equations is used to calculate the data volume.

For raw images, we use

$$\text{Data Volume}^{(i)} = \text{number of sent pixels} * \text{bytes per pixel}$$

with bytes per pixel being one for the data type $int8$, which is the usual representation of a grayscaled image.

For model activations, we use

$$\text{Data Volume}^{(i)} = \text{number of sent activations} * \text{bytes per activation}$$

with bytes per pixel being four for the data type $float32$, which is the default data type in PyTorch [21]. Note that some image formats use compression methods that do not result in a loss of quality. A comprehensive comparison would include those methods, but that is beyond the scope of this work.

5.2.5 Evaluation Experiments

At first, we compare the data volume of each model to evaluate the cost savings of different compression blocks. These are the communication cost savings that we can achieve by using a federated model without a router for a single inference. Afterwards, we compare the accuracies of the global and local models to answer the question how local (global) models of HF -NNs perform compared to the traditionally trained local (global) models. Furthermore, we compare the histograms of the router outputs of different classes to gain intuitions of how the router quantifies hard-to-classify instances. Last but not least, we evaluate the router by comparing the hybrid accuracies and remote coverage to show communication cost savings enabled by the router.

Chapter 6

Evaluation

Implementation For this experiment, we run the training process as described in Algorithm 4.1 with hybrid models and Algorithm 4.1 for the local-only and remote-only CNN models with three randomly chosen *seeds* $\in \{4, 13, 27\}$ for reproducibility. The training processes are executed three times so that we can estimate the variance of the models given different initial parameters and data since they can significantly influence the model performances. We calculate the accuracies of the remote-only and local-only CNN models, the local and remote models of the Hybrid V1 and V2 for comparison. For the hybrid models, this represents deployments in which we use either only the remote or only the local model to infer the targets.

6.1 Data Volume Comparison

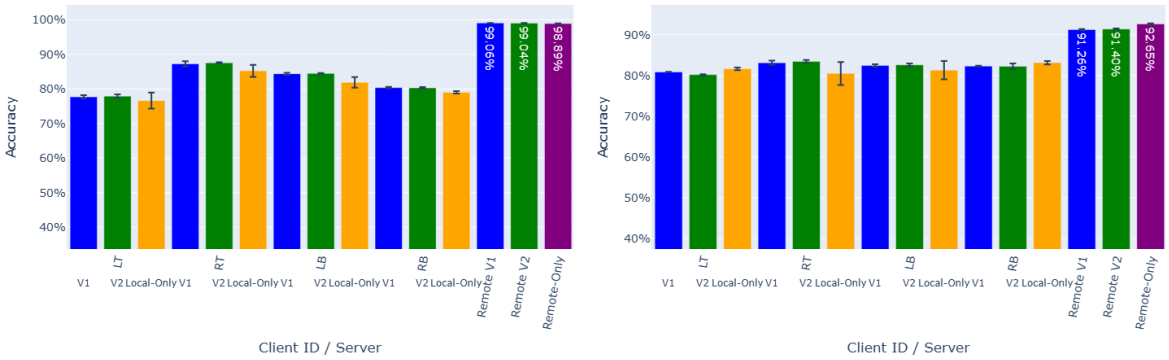
Table 6.1: Comparison of model variants across output size and data transfer.

	Local-Only CNN	**Remote-Only CNN**	**Hybrid V1**	**Hybrid V2**
Local Output Dimension (size of $F^{(i)}(x_j^{(i)})$)	-	-	49	32
Data Volume (Bytes per client)	0	196	196	128

In Table 6.1, we see a comparison of the different deployment models and the corresponding data volume transmitted per client, required for a single inference of one sample. The Hybrid V1 model with 49 activations transmitted and the remote-only CNN model with 14×14 pixels sent both result in an identical data volume of 196 bytes per client. For the Hybrid V2 model, we register a reduction of 34.7% to 128 bytes. This is the reduction that we achieve while only using the remote model of the Hybrid V2 model.

6.2 Performance of Hybrid Federated Neural Networks

The composite loss function of the shared base and the simultaneous training of three different heads with cyclic dependencies add complexity to the training process compared to traditional CNN models. With the following experiment, we want to compare the local and remote performances of those models.



(a) MNIST **(b)** Fashion MNIST

Figure 6.1: Comparison of average accuracy between the local model and global model by different clients (LT, RT, LB, RB) and different models (Hybrid V1, Hybrid V2, Local-Only CNN and Remote-Only CNN) over three different seeds. The bar plots illustrate the mean and the error bars visualize the standard deviations of the results.

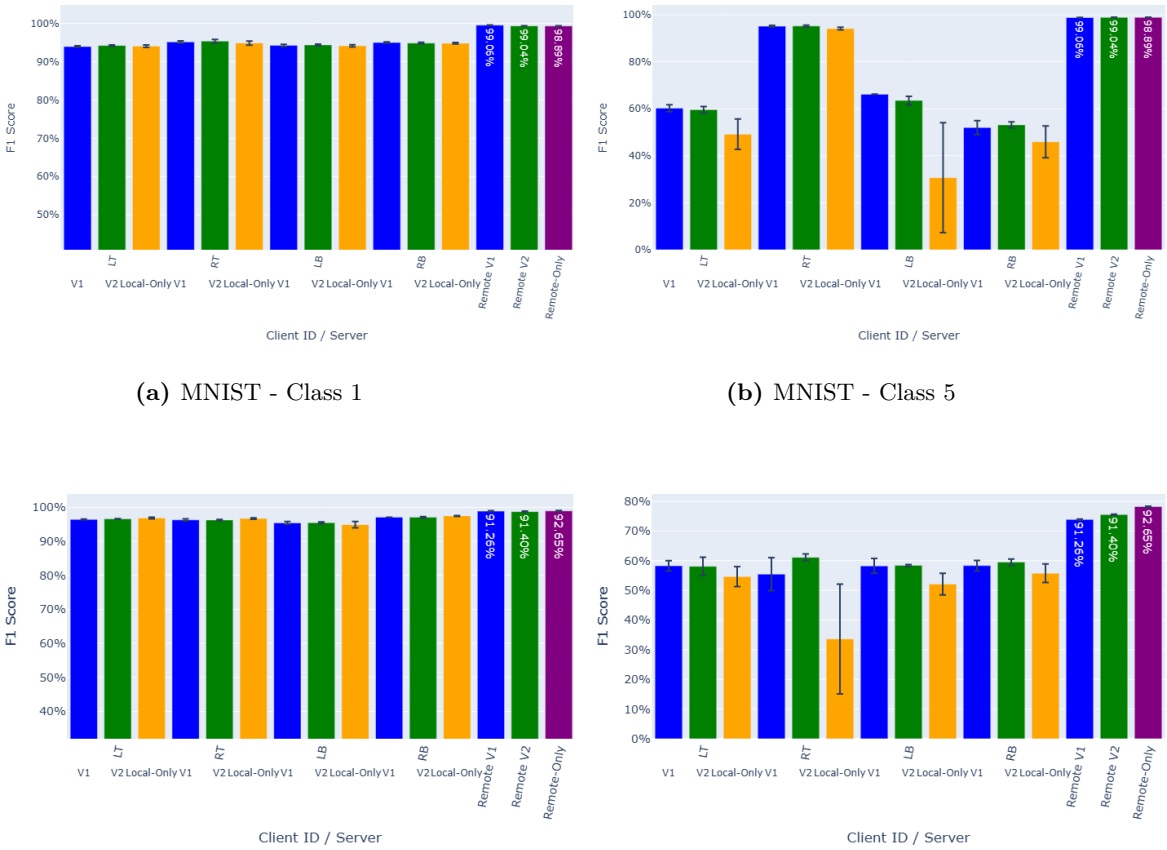
In Figure 6.1a, we see the accuracy of the four different models divided into local or remote inferences. The blue and green bars visualize the performances from HF -NN models, the yellow bars represent local-only models and the purple bars depict remote-only models.

We achieve the highest average accuracy with the Hybrid V1 model with an average of 99.0% for MNIST and 92.6% using the remote-only CNN model on Fashion MNIST. In MNIST, all remote models have similar average performances with the maximum difference being $\Delta acc = 0.002$ between Hybrid V1 and the remote-only CNN. In comparison, we see a drop in performance from 92.7% using the remote-only model to 91.2% using the Hybrid V1 model for the Fashion MNIST dataset. Unexpectedly, even Hybrid V2 with its lower data volume has a higher accuracy than the Hybrid V1.

The results indicate that we can trade local computation for better bandwidth usage for the MNIST dataset if we deploy the remote model of the Hybrid V2 model instead of the remote-only CNN. Because both hybrid models in the Fashion MNIST setting achieve accuracies over 91%, we can say that we can reduce communication cost with a minimal loss of performance.

Analyzing the local models, for the MNIST dataset, we find the lowest average accuracy of 76.6% with the local-only CNN model in Client LT. For the Fashion MNIST dataset, the lowest average of 80.2% was achieved by Hybrid V2 in Client LT. As seen in Figure 6.1a, local-only CNN models have higher standard deviations with an average of ± 0.015 and a maximum deviation of ± 0.023 for the MNIST dataset. Fashion MNIST-based local-only CNN models have an average standard deviation of ± 0.014 and a maximum of ± 0.028. This indicates higher performance fluctuations and weaker convergence in local-only models relative to their hybrid local counterparts.

Looking at the F1-Scores of these models, we find that not every class is equally affected by the splitting of the images (See Figure 6.2). While the F1-Scores of the local



(a) MNIST - Class 1

(b) MNIST - Class 5

(c) Fashion MNIST - Class Trouser

(d) Fashion MNIST - Class Shirt

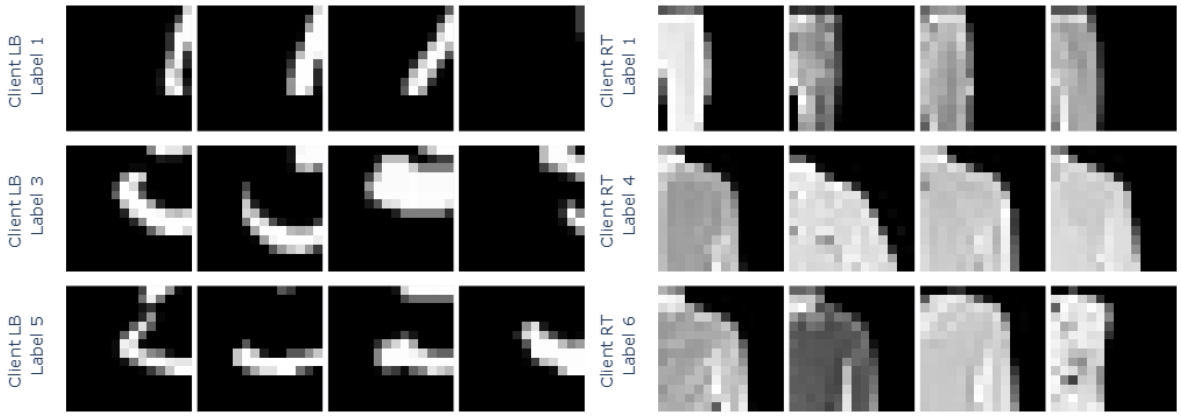
Figure 6.2: Comparison of average F1-Score between local and global models across different clients (LT, RT, LB, RB) and models (Hybrid V1, Hybrid V2, Local-Only CNN, and Remote-Only CNN), evaluated over three seeds. Each bar plot shows the mean F1-score, with error bars representing standard deviations.

models for class 1 and class 'trouser' remain high throughout all clients after image splitting, we see for class 5 in MNIST and for class 'shirt' in Fashion MNIST high variances in performance. The lowest scores are found for the MNIST dataset in the LB client with an average score of 30.6% and for the Fashion MNIST in the RT client with 33.6%. The reason for this mean reduction and the high variance in performance is that the division of the image into quadrants transforms the local image classification task into ambiguous single-label classification tasks. If we look at the local view of Client LB (RT) for the MNIST (Fashion MNIST) dataset, we see that instead of a hard label of '5' ('shirt'), a soft label of [0.5, 0.5] for the classes [3, 5] (['shirt', 'coat']) would be more appropriate for the local classification task (See Figure 6.3). Without the information of the other clients, the local task is therefore ambiguous, because both answers can be right depending on the information that we find in the views of other clients. Because both answers could be correct without additional knowledge, we therefore see higher or

lower performances depending on the local class bias. This explains the high variance in those ambiguous classes and there we also find the biggest improvements.

In contrast, for already stable classes such as class 1, the performance is not always higher using the hybrid model as seen in Figure 6.2a and Figure 6.2c.

Thus, while this method improves the average performance and convergence over all clients, this still can lead to reduced accuracy or increased standard deviations in single clients or single classes. *The results show that vertical separations of data can introduce ambiguity between classes for local clients, i.e., an isolated client cannot distinguish between certain classes and needs access to additional data to make class distinctions. This can limit the performance of isolated local models. The new task is therefore a combined task of classification with an information gathering decision. We observe that mostly clients with high performance fluctuations have an greater incentive to train hybrid models, if all clients remain self-sufficient after training. The remote model improves the performance of all clients, but the resulting remote model does not necessarily perform better than a model with the same convolutional block with shared convolutional layers between clients.*



(a) MNIST - Class 1, 3, 5

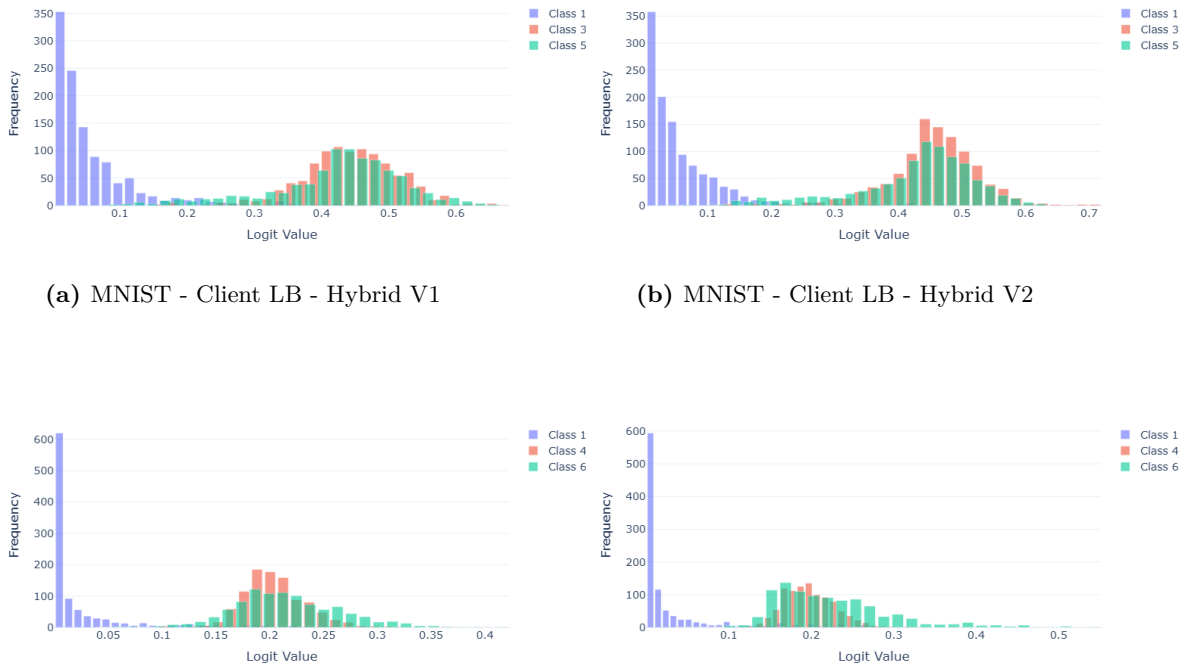
(b) Fashion MNIST - Class Trouser, Coat and Shirt (top to bottom)

Figure 6.3: Visualization of Ambiguous Single-Label Classification Due To Vertically Separated Data

6.3 Router Evaluation

Implementation For this experiment, we choose the models with the highest remote performance to analyze the router, which is $seed = 4$ for both hybrid models for Fashion MNIST and for the MNIST dataset, it is $seed = 13$ for V1 and $seed = 27$ for V2.

The idea of the hybrid model is that we want to reduce communication efficiency by using the router to decide which classifier head to call for inference. As seen in Figure 6.4, the router shows different router logit distributions for different classes. For the LB client using the MNIST dataset, we observe that classes with higher local class performance are screwed to the left and have a majority of their logit values between 0 to 0.2, while difficult classes are located closer to the center between 0.3 to 0.6. Similarly for the Fashion MNIST dataset, we find the logit values of class 1 between 0 to 0.1 and for the classes 4 and 6 values between 0.15 to 0.3. This indicates that the router does quantify the difficulty of samples considering that it can place classes with high F1-Scores closer to zero than classes that are ambiguous, as previously discussed. *Thus, if we define images that are more ambiguous due to high feature similarities between classes as locally hard-to-infer instances, then the router logit indicates the difficulty of a sample.*



(a) MNIST - Client LB - Hybrid V1

(b) MNIST - Client LB - Hybrid V2

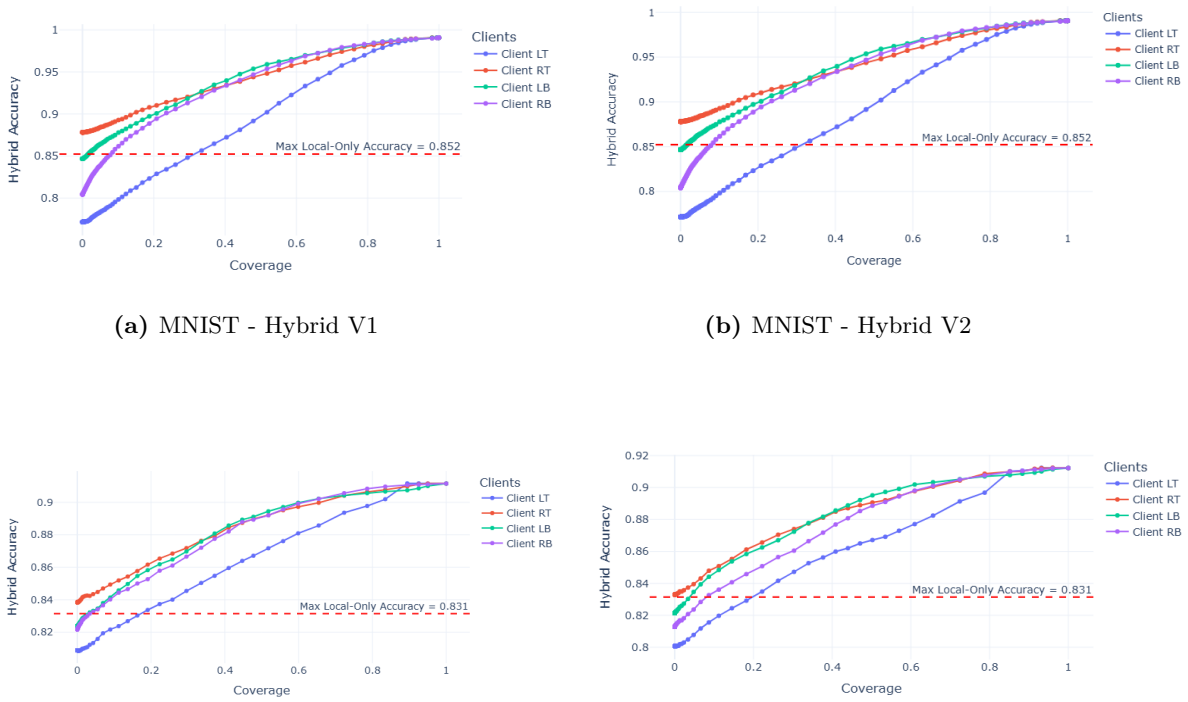
(c) Fashion MNIST - Client RT - Hybrid V1

(d) Fashion MNIST - Client RT - Hybrid V2

Figure 6.4: Histograms of the router logits for Hybrid V1 and Hybrid V2 grouped by classes

6.4 Communication-Efficient Federated Inference

Last but not least, we discuss the results regarding the communication efficiency of the hybrid models using Protocol 4.5. Similarly to the previous experiment, we have chosen the models with the highest remote performance for evaluation. In Figure 6.5, we see the hybrid accuracy and remote coverage curves (HARC curve) of trained HF -NN models with decreasing thresholds. In addition, we have added the red dotted lines to visualize the maximum value of the local-only models from Figure 6.2.



(a) MNIST - Hybrid V1

(b) MNIST - Hybrid V2

(c) Fashion MNIST - Hybrid V1

(d) Fashion MNIST - Hybrid V2

Figure 6.5: Hybrid Accuracy - Remote Coverage Curve of Hybrid V1 and V2 with decreasing threshold $\epsilon \in [0, 1]$. A threshold of $\epsilon_r = 0$ means that all instances are sent to the cloud, while $\epsilon_r = 1$ denotes that the model only uses local inferences.

The curves display the trade off relations between coverage and hybrid accuracy. It shows for the MNIST dataset that we have a steep increase in accuracy in the beginning which fades out near $\sim 80\%$ remote coverage for the clients RT, LB and RB. For Client LT, which also has the lowest average local accuracy, the accuracy improves almost linearly as remote coverage increases, continuing up to around 85% coverage. This indicates that the mislabels are uniformly distributed over the router spectrum, suggesting that there are many ambiguous samples. This highlights our concerns about knowledge distillation-based learning methods and emphasizes the importance of involving other

clients during inference. For the Fashion MNIST dataset, we observe similar patterns in the trade-off curves. In contrast, we see more jumps in hybrid accuracies in the curves of the Fashion MNIST dataset for instance by client LT at $\sim 80\%$ remote coverage, which shows that the incorrectly labeled instances are more clustered around certain thresholds. The reasons for these jumps are currently unknown and additional research is needed to explain those findings. We discover that with only $\sim 30\%$ remote coverage for MNIST and only $\sim 20\%$ for the Fashion MNIST dataset, we reach for all clients the best average local-only accuracy. We can reduce remote coverage by $\sim 15\%$ for Fashion MNIST and $\sim 10\%$ while only decreasing the model performance by $\Delta acc = -1\%$ point. *We see that we can control the amount of communication by adjusting the router threshold. The amount that can be saved for a fixed hybrid accuracy depends on the information value of the clients.*

Chapter 7

Discussion

Before going into the limitations of the solution, we will summarize the findings. Our evaluation shows that the router can estimate the complexity of different samples and can be used to balance performance and bandwidth usage. The amount that can be saved for a fixed hybrid accuracy depends on the information value of the clients. Thus, VFIs are not necessary for all samples to maintain a high hybrid accuracy. Furthermore, the router evaluation shows that by using Protocol 4.5, we can increase the communication efficiency of a vertical federated system. In addition, the results indicate that vertical separations of data can introduce ambiguity between classes for local clients. This can be a limiting factor for isolated local models. We observe that mostly clients with high performance fluctuations gain higher convergence properties via hybrid training and profit the most from the federated system. Although the remote HF -NN model outperforms all local models individually, this advantage does not necessarily lead to better performance compared to a model that shares the same convolutional block across clients. For example, as seen for Client LT using the MNIST dataset, the accuracy improves linearly as the remote coverage increases which indicates that most samples are similarly difficult to infer. For those clients, it might be better to skip the router and directly send the activation to reduce the computational overhead of computing the router probability.

One limitation is derived from the network assumptions that we use. We use remote coverage to measure possible communication cost savings. Real-world settings include network fluctuations that can significantly impact communication costs. More sophisticated network models can improve our understanding of the applicability of this model for VFI to reduce communication latency.

Furthermore, we compare the data volume of the activation with the data volume of uncompressed images. Since int8 images have compression methods that do not reduce image quality, this overestimates the possible fixed savings of the data volume.

Since the HF -NN model simultaneously trains five classification heads and four different routers, the model-complexity is significantly higher compared to typical CNN models. This produces a longer training time and higher initial energy costs.

The evaluation of the router is limited to the comparison of logit distributions of a set of classes known to differ in classification difficulty. Comparing it to other routing strategies might give us a better understanding of the effectiveness of the router. It would be a great idea to compare it with confidence thresholding to assess if the additional

computation cost for a learned router is necessary.

Chapter 8

Conclusion and Future Work

To answer the research questions, we have shown that we can train a router using HF -NN to quantify the difficulty of samples based on the proxy of how likely the local model mislabels a similar sample and can receive the correct label from the global model. Thus, we can find a set of samples for which VFI is unnecessary to maintain a high hybrid accuracy. We show that a router can be used to trade accuracy for better bandwidth usage and that it can be controlled by setting the router threshold accordingly. This can improve communication efficiency with respect to data volume and latency.

8.1 Future Work

To conclude this work, we summarize the future directions of our research. As mentioned in Chapter 7, there are questions that were beyond the scope of this research project. Due to time constraints, it was not possible to compare different compression algorithms in this work. It would be interesting to see how the proposed encoder-only compression layer performs compared to other compression methods or if there are better combinations to optimize the data volume. Another section that is not extensively explored yet, is the performance of the router. We want to compare the router with confidence thresholding in the future to better assess its effectiveness. Moreover, adding a mechanism that dynamically changes the router threshold based on current network conditions could potentially optimize network traffic while keeping the system performance as high as possible.

In addition, we can derive open research questions from the system assumptions to further validate its usage for different scenarios. As mentioned, we see opportunities in its usage for networks with high network fluctuations. Because it enables the option of inferring a target solely based on the local view, we believe that this can be used for fallback mechanisms in time-constrained systems. Furthermore, VFI usually requires the participation of all clients. If a client is missing, malicious, or for some reason cannot send the required data, we often find a significant drop in performance [30]. We hypothesize that comparing local and global inferences can mitigate these effects. Last but not least, the router indicates that there are different levels of gains that each client can achieve using HF -NNs. The local models with the highest performance fluctuations gain the most in accuracy and convergence. Furthermore, the way the router works suggests that

the better the local model performs, the less the global system is of use to that client. Thus, incentive mechanisms might be necessary to motivate clients to participate and share their data.

References

Literature

[1] Sawsan AbdulRahman et al. "A survey on federated learning: The journey from centralized to distributed on-site learning and beyond". *IEEE Internet of Things Journal* 8.7 (2020), pp. 5476–5497 (cit. on pp. 1, 10, 11).

[2] Igor Fedorov Anil Kag. "Efficient Edge Inference by Selective Query". *International Conference on Learning Representations* (2023). URL: https://par.nsf.gov/biblio/10447699 (cit. on pp. 8, 10, 12, 20, 22, 23, 29).

[3] Dongqi Cai et al. "Accelerating vertical federated learning". *IEEE Transactions on Big Data* 10.6 (2022), pp. 752–760 (cit. on pp. 17, 21).

[4] Timothy Castiglia et al. "Compressed-VFL: Communication-efficient learning with vertically partitioned data". In: *International Conference On Machine Learning*. PMLR. 2022, pp. 2738–2766 (cit. on pp. 17, 21).

[5] Timothy Castiglia et al. "LESS-VFL: Communication-efficient feature selection for vertical federated learning". In: *International Conference on Machine Learning*. PMLR. 2023, pp. 3757–3781 (cit. on p. 17).

[6] Anirban Das et al. "Vertical federated learning with compressed embeddings". US 12,033,074 B2. July 2024 (cit. on pp. 17, 21).

[7] Li Deng. "The mnist database of handwritten digit images for machine learning research". *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142 (cit. on pp. 4, 27).

[8] Siwei Feng. "Vertical federated learning-based feature selection with non-overlapping sample utilization". *Expert Systems with Applications* 208 (2022), p. 118097 (cit. on p. 17).

[9] Martin T. Hagan et al. *Neural Network Design (2nd Edition)*. 2014 (cit. on pp. 4, 5, 7, 8).

[10] Charles R. Harris et al. "Array programming with NumPy". *Nature* 585.7825 (Sept. 2020), pp. 357–362 (cit. on p. 29).

[11] Mert Kayaalp et al. "Causal Influence in Federated Edge Inference". *IEEE Transactions on Signal Processing* 72 (2024), pp. 5604–5615 (cit. on pp. 1, 17).

[12] Afsana Khan, Marijn ten Thij, and Anna Wilbik. "Communication-efficient vertical federated learning". *Algorithms* 15.8 (2022), p. 273 (cit. on pp. 17, 21).

[13] Afsana Khan, Marijn ten Thij, and Anna Wilbik. “Vertical federated learning: A structured literature review”. *Knowledge and Information Systems* (2025), pp. 1–39 (cit. on pp. 1, 14).

[14] Denise-Phi Khuu. *Data Poisoning Detection in Federated Larning*. Bachelor’s thesis. 2023 (cit. on pp. 5, 6).

[15] Jakub Konecný et al. “Federated Optimization: Distributed Machine Learning for On-Device Intelligence”. *ArXiv* abs/1610.02527 (2016) (cit. on p. 10).

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet classification with deep convolutional neural networks”. *Commun. ACM* 60.6 (May 2017), pp. 84–90 (cit. on pp. 4, 8).

[17] En Li et al. “Edge AI: On-demand accelerating deep neural network inference via edge computing”. *IEEE Transactions on Wireless Communications* 19.1 (2019), pp. 447–457 (cit. on p. 12).

[18] Yang Liu et al. “Vertical federated learning: Concepts, advances, and challenges”. *IEEE Transactions on Knowledge and Data Engineering* 36.7 (2024), pp. 3615–3634 (cit. on pp. 11, 12, 17).

[19] Linjun Luo and Xinglin Zhang. “Federated split learning via mutual knowledge distillation”. *IEEE Transactions on Network Science and Engineering* 11.3 (2024), pp. 2729–2741 (cit. on p. 17).

[20] Nathan Ng et al. “Collaborative Inference in Resource-Constrained Edge Networks: Challenges and Opportunities”. In: *IEEE Military Communications Conference*. IEEE. 2024, pp. 1–6 (cit. on pp. 1, 12).

[21] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035 (cit. on pp. 28–30).

[22] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 29).

[23] Maarten G. Poirot et al. “Split Learning for collaborative deep learning in healthcare”. *CoRR* (2019) (cit. on pp. 1, 8).

[24] Lutz Prechelt. “Automatic early stopping using cross validation: quantifying the criteria”. *Neural Networks* 11.4 (1998), pp. 761–767 (cit. on p. 28).

[25] Alec Radford et al. “Language models are unsupervised multitask learners”. *OpenAI blog* 1.8 (2019), p. 9 (cit. on pp. 4, 8).

[26] Wei-Qing Ren et al. “A survey on collaborative DNN inference for edge intelligence”. *Machine Intelligence Research* 20.3 (2023), pp. 370–395 (cit. on pp. 1, 14).

[27] Zhenghang Ren, Liu Yang, and Kai Chen. “Improving availability of vertical federated learning: Relaxing inference on non-overlapping data”. *ACM Transactions on Intelligent Systems and Technology* 13.4 (2022), pp. 1–20 (cit. on p. 17).

[28] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. *CoRR* abs/1609.04747 (2016) (cit. on pp. 6–8).

[29] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. "Branchynet: Fast inference via early exiting from deep neural networks". In: *International Conference on Pattern Recognition*. IEEE. 2016, pp. 2464–2469 (cit. on pp. 8, 10, 21, 23).

[30] Pedro Valdeira, Shiqiang Wang, and Yuejie Chi. "Vertical Federated Learning with Missing Features During Training and Inference". In: *The Thirteenth International Conference on Learning Representations*. 2025 (cit. on pp. 14, 41).

[31] Pedro Valdeira et al. "Communication-efficient vertical federated learning via compressed error feedback". *IEEE Transactions on Signal Processing* (2025) (cit. on pp. 17, 21).

[32] Junhao Wang et al. "Tvfl: Tunable vertical federated learning towards communication-efficient model serving". In: *IEEE Conference on Computer Communications*. IEEE. 2023, pp. 1–10 (cit. on pp. 1, 17, 18).

[33] Zhaomin Wu, Qinbin Li, and Bingsheng He. "Practical vertical federated learning with unsupervised representation learning". *IEEE Transactions on Big Data* 10.6 (2022), pp. 864–878 (cit. on p. 27).

[34] Jens-Peter M Zemke. *Lecture Notes Advance Machine Learning*. Sept. 2022 (cit. on pp. 5, 7, 8).

Online sources

[35] Fei-Fei Li, Jiajun Wu, and Ruohan Gao. *Lecture Notes on CS231n: Convolutional Neural Networks (CNNs / ConvNets)*. URL: https://cs231n.github.io/convolutional-networks/ (visited on 09/23/2025) (cit. on pp. 5–7).

[36] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG] (cit. on pp. 4, 27).