TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Embedded System Architecture for the Acceleration of Collaborative Learning in Neural Networks

---

Author:
Emmanouil PETRAKOS

Thesis Committee:
Prof. Apostolos DOLLAS
Prof. Michail G. LAGOUDAKIS
Asst. Prof. Vassilios PAPAEFSTATHIOU (CS Dept., U. of Crete)

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

March 23, 2023

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Embedded System Architecture for the Acceleration of Collaborative Learning in Neural Networks

by Emmanouil PETRAKOS

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too…

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Embedded System Architecture for the Acceleration of Collaborative Learning in Neural Networks

by Emmanouil PETRAKOS

Η περίληψη της διπλωματικής γράφεται εδώ (και συνήθως αποτελεί αυτή την μία μόνο σελίδα). Η σελίδα αυτή κρατάται στοιχισμένη στην μέση οριζόντια και κάθετα, ώστε να μπορεί να επεκτείνεται στον κενό χώρο και πάνω από τον τίτλο...

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ANN	**A**rtificial **N**eural **N**etwork
API	**A**pplication **P**rogramming **I**nterface
CCPA	**C**alifornia **C**onsumer **P**rivacy **A**ct
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessor **U**nit
DNN	**D**eep **N**eural **N**etwork
DL	**D**istributed **L**earning
DP	**D**ifferential **P**rivacy
FedAvg	**Fed**erated**Av**erag**g**ing
FL	**F**ederated **L**earning
FLP	**F**lushable **P**ipeline
FRP	**F**ree **R**unning **P**ipeline
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GAN	**G**enerative **A**dversarial **N**etwork
GDPR	**G**eneral **D**ata **P**rotection **R**egulation
GPU	**G**raphics **P**rocessing **U**nit
GE	**G**lobal **E**poch
HLS	**H**igh **L**evel **S**ynthesis
II	**I**teration **I**nterval
IID	**I**ndependent and **I**dentically **D**istributed
LR	**L**earning **R**ate
MEC	**M**ulti-access **E**dge **C**omputing
ML	**M**achine **L**earning
MSE	**M**ean **S**quare **E**rror
MPC	**M**ulti-**P**arty **C**omputation
PL	**P**rogrammable **L**ogic
RNN	**R**ecurrent **N**eural **N**etwork
SoC	**S**ystem **O**n **C**hip
XRT	**X**ilinx **R**untime **L**ibrary

Dedicated to my family and friends…

Chapter 1

Introduction

In recent years, edge devices with advanced computing and data collection capabilities are becoming commonplace. As a result, massive volumes of new and useful data are generated, which can be exploited in Machine Learning (ML). When combined with recent advances and techniques in ML, new opportunities emerge in a variety of fields, including self-driving automobiles and medical applications.

Traditional ML approaches demand the data to be consolidated in a single entity where learning takes place. However, due to unacceptable latency and storage requirements of centralizing huge amounts of raw data, this may be undesirable. To address the inefficiency of data silos, cloud computing architectures such as Multi-access edge computing (MEC) [1] have been proposed in order to transfer the learning closer to where the data is produced. Unfortunately, these techniques still require raw data to be shared between the edge devices and intermediate servers.

Due to growing privacy concerns, recent legislation like General Data Protection Regulation (GDPR) [2] and California Consumer Privacy Act (CCPA) [3] have severely limited the usage of technologies that transfer private data. To continue leveraging the increasing real-world data while adhering to such regulations, the concept of Federated Learning (FL) [4] has been introduced. FL is a collaboratively decentralized privacy-preserving technology, in which learning takes place at the data collection point, i.e. the edge device. The edge devices train a ML model provided by the server and share model updates instead of raw data. As a result, collaborative and distributed ML is possible while maintaining the privacy of the participating devices.

1.1 Motivation

Most FL research, to our knowledge, focuses on simulations and treats edge devices as black boxes; generally ignoring their nature and constrains. Taking in consideration the complexities of implementing ML on hardware, recent advancements in FL might be diminished or invalidated. The main motivation of this thesis is to identify, explore and possibly overcome the intrinsic conflicts that exist between FL and Artificial Neural Network (ANN) training in Field Programmable Gate Arrays (FPGA)s.

Instead of being incompatible, these two technologies may complement each other, which is worth investigating. Frequently in FL, transformations are applied on the generated ANN variables to reduce network utilization and enhance privacy. These transformations, which include quantization [5], adding Gaussian noise [6] and others, tend to be spatially independent and could be implemented highly efficiently in hardware accelerators like FPGAs.

Finally, FL literature is almost devoid of wall-clock time examples. This thesis aims to provide a real world FL implementation that may be considered as a benchmark for future research. Furthermore, in order to be extendable and utilized in future works, the FL implementation is modular and platform independent.

1.2 Scientific Contributions

The main focus of this thesis is combining FL training with FPGA based ANN implementations, while exploring and overcoming their inherent conflicts. Furthermore, it focus on the mostly unexplored FL setting of small client pools and its implicit difficulties. Finally, it provides a real world implementation of FL that can be used as a benchmark for future works. This FL implementation is agnostic of the ANN training implementation and can be used as a starting point for future works.

1.3 Thesis Outline

- **Chapter 2 - Theoretical Background:** Description of the theoretical background of ML and FL.
- **Chapter 3 - Related Work:** Related works on FL, optimization techniques and hardware implementations of it.

- **Chapter 4 - FL architecture & design:** Description of the FL architecture, design and implementation developed.
- **Chapter 5 - Robustness Analysis:** Analysis of the quality and performance of the FL implementation.
- **Chapter 6 - FPGA Implementation:** Description of the ANN architecture, design and implementation on FPGA developed.
- **Chapter 7 - Results:** Analysis of the quality and performance of the complete system.
- **Chapter 8 - Conclusions and Related Work:** Chapter 8 description

Chapter 2

Theoretical Background

2.1 Artificial Intelligence & Machine Learning

Various researchers and textbooks may provide different definitions of Artificial Intelligence (AI). Depending the school of though, AI is an artificial actor that thinks or acts, rationally or human-like, depending on what it knows. Generally, AI can be described as the study of intelligence agents. It is a modern science that encompasses a large variety of sub-fields, ranging from general-purpose areas, such as learning, to specific tasks like playing chess and giving medical diagnoses. AI can be relevant to any intellectual field, as it systematizes and automates intellectual tasks. [7]

Machine learning (ML) is an AI field in which agents, in addition to the performance element, include a learning element that utilises their past experiences to enhance their behaviour. The core idea behind ML is that perception should be used to improve the ability to act in the future, not simply react in the present. Designing a learning element is a multi-facet problem that is affected by three major issues. [8]

2.1.1 Information management

The first issue is determining what information what information is useful and how it should be utilized. Different components of the input and output data should be learnt depending on the context in which the learning actor operates. One method is to directly link the current state of the actor or the world to their actions. Sometimes it can be more appropriate to infer relevant patterns from the data while ignoring unnecessary information. Another way is to collect action-value information indicating the desirability of actions based on their effect in the world state. These and other options

may need to be combined in order to extract the most meaningful knowledge from the available data.

A common example is feature extraction. In ML, a feature [9] is an individual measurable property or characteristic of a phenomenon being observed. They can be generic, such as edges in an image, or specialized, such as wheels and animal height. Feature extraction is the process of transforming such raw data into numerical features that can be processed.

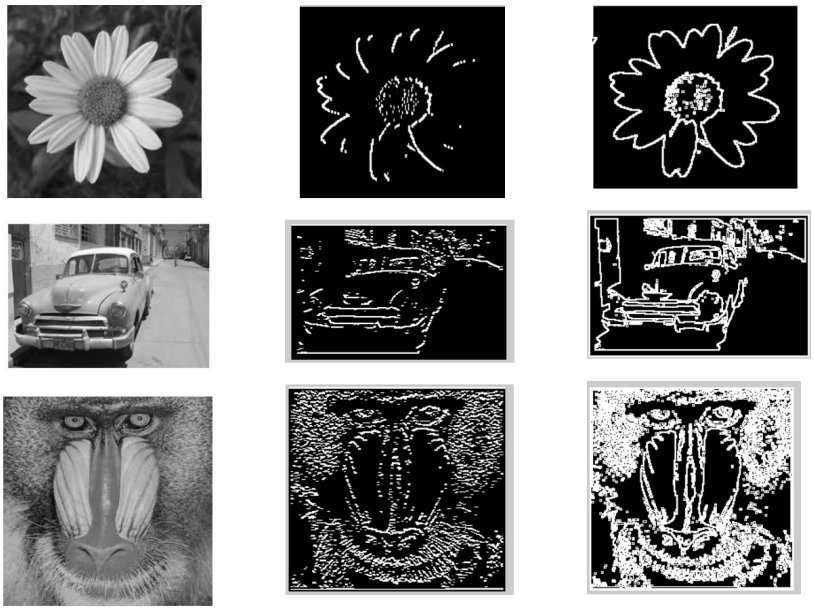

FIGURE 2.1: Edge detection in greyscale images: URL.

Another key factor when designing learning systems is the availability of prior knowledge. Researchers have extensively looked into the issue where the agent uses only information that they encounter, but ways for transferring prior knowledge have been devised to speed up learning and improve decision-making.[10]

2.1.2 Feedback mechanism

The type of feedback available has a significant impact on the design and is perhaps the most crucial aspect of the learning problem. Usually three major types are distinguished: supervised, unsupervised, and reinforcement learning.

Supervised learning problems involve learning functions between sets of inputs and outputs. This is the case of a fully observable environments where the effects of the actors actions are immediately visible or the existence of a third party providing the correct solutions.

Unsupervised learning problems, on the other hand, do not supply output values and learning patterns are solely based on the input. As it has no knowledge of what constitutes a correct action or a desired state, an unsupervised learning agent cannot learn what to do. The hope is that through mimicry, the algorithm will generate imaginative content from it. This is a common scenario for probabilistic reasoning systems or when generating output data is prohibitively expensive. For the last case, a semi-supervised learning setting, in which only a subset of the outputs is generated, might be useful.

In the reinforcement learning setting there is no correct output provided, instead a reward is given to actor appropriate to the desirability of their actions. This is common when the world which the actor take part in continuously change according to their actions, or a desirable or undesirable state may be reached after a series of actions.

2.1.3 Representation of the learned information

The representation of the learned information is another important factor in establishing how the learning algorithm should operate. Common schemes include linear weighted polynomials for utility functions, propositional or first order logic, probabilistic representations like Bayesian Networks[11] and ANNs[12], and other methods have all been created.

2.2 Deep learning

Deep learning is a sub-field of ML, partially overlapping with big data science. It consists of algorithms that use the perceptron as their basic building block, which is a mathematical function based on the McCulloch-Pitts model of biological neurons. They typically have hundreds of thousands to millions of perceptors with a variety of designs and topologies. Deep learning architectures include Deep Neural Networks (DNN)s, Convolutional Neural Networks (CNN)s, Recurrent Neural Networks (RNN)s and others, each one offering different capabilities and options.

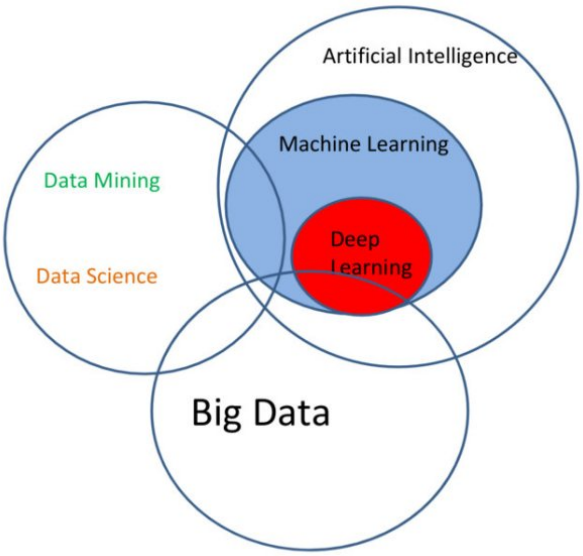

FIGURE 2.2: AI, ML, DL, Data Mining, Data Science, and Big Data: URL.

Deep learning applications have demonstrated human-like or superior capabilities in several scientific and commercial fields such as image[13] and speech[14] recognition, natural language processing[15], climatology[16] and biotechnology[17]. Due to these exceptional capabilities and wide range of applications, deep learning has attracted a large number of researchers from various scientific domains, resulting in its tremendous expansion. However, the science is still young and there are a number of challenges to be overcome. Expecting deep learning combined with improved data processing being a solution to computers gaining generic human-like intelligence (human equivalent AI) is still a distant dream.[18]

Historically, the field of deep learning emerged in 1943 with the inception of the aforementioned McCulloch-Pitts perceptron. In 1949, Donald Hebb noted out in his book "The Organization of Behavior" that neural pathways are strengthened each time they are utilized, a principle that is crucial to how humans learn. He claimed that when two nerves fire at the same moment, the link between them is strengthened. This progress resulted in the creation of the first real-world application of ANNs, "MADALINE" an adaptive filter that eliminates echoes on phone lines. In 1962, Widrow & Hoff developed a learning procedure that distributed the error across the ANN, resulting in its eventual elimination. Despite these advances, deep learning

research plummeted due to a variety of internal and external factor, including the widespread use of fundamentally faulty learning function and the adoption of von Neumann architecture across computer science.

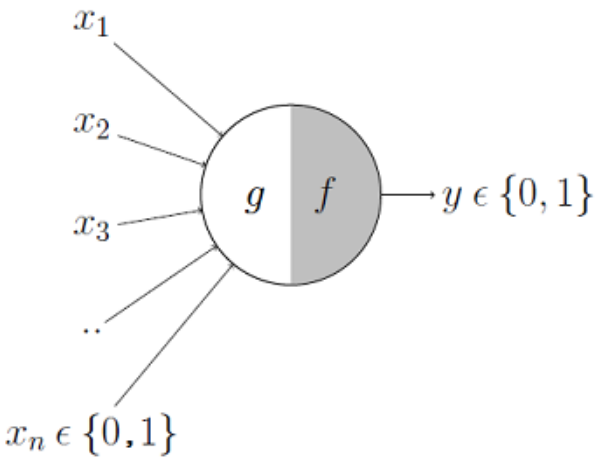

FIGURE 2.3: The McCulloch-Pitts Neuron: URL.

Deep learning research stagnated until 1975, when developments such as Werbos' backpropagation and the building of the first multilayered network reignited interest in the field. Since then, the field continues to expand with innovations like hybrid models and ANN pooling layers. The current focus is on developing deep learning-specific hardware, as fast and efficient ANNs rely on it being defined for their use. Generally, architectures based on accelerators such as GPUs and FPGAs, or VLSI hardware-based designs, outperform CPU-based architectures. [19]

2.2.1 Artificial Neuron

As previously stated, the perceptron, also known as the artificial neuron, is the fundamental building element of the deep learning algorithms. In its simplest form, the artificial neuron receives one a set of inputs and sums it to produce an output. In practice, each input is weighted, then summed with a bias variable that acts as a threshold value, and the output is produced using an activation function.

The mathematical formula of the artificial neuron is defined as:

$$y = \Phi(b + \sum_{i=1}^{I} x_i * w_i) \tag{2.1}$$

Where:

y = output
b = bias
I = number of inputs
Φ = activation function
w = weight

2.2.2 Activation Functions [1]

The activation function[20] of the artificial neuron is arguably its most important feature. It specifies how the weighted total of the inputs is transformed into an output (a target variable, class label, or score). Sometimes they limit their output range and are called squashing functions. There are various functions that are used as activation functions, with different properties and use cases each.

Most activation function are usually nonlinear so that the output varies nonlinearly with the inputs. With a linear activation function, regardless of how many layers a ANN has, it would behave just like a single-layer perceptron, as stacking linear functions creates just another linear function. Nonlinearity is, arguably, the most important aspect of the activation functions.

Activation functions are usually differentiable, which means that for a given input value, the first-order derivative can be determined. This is necessary because ANNs are mostly trained using the backpropagation of error algorithm, which requires the derivative of prediction error to update the model's parameters.

Binary Step

This is arguably the most basic activation function, as it was originally used in the McCulloch-Pitts Neuron and operates like a simple threshold. It activates the output of the perceptron when a certain value is exceeded, else the output is set as zero.

$$f(x) = \begin{cases} 0 & x \leq threshold \\ 1 & x > threshold \end{cases} \tag{2.2}$$

[1] Also called transfer functions.

Sigmoid

Also known as the logistic function, it normalizes and squashes the output of the neuron between 0 and 1. Its most important properties are that the output is barely affected by extreme values and the derivative is easily calculated.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

ReLU

Because of its simple implementation, non-linearity, and high performance, the Recti-Linear Unit or ReLU function is arguably the most commonly utilized function in ANNs. It combines the binary step function for negative values and the identity function for positive values.

$$f(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{2.4}$$

Softmax

Softmax ensures that all the outputs sums to 1 by normalizing them to a probability distribution. As such, it is mostly used as the final activation function in multi-decision ANN models.

$$f(x)_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_j}} \tag{2.5}$$

2.2.3 Artificial Neural Network architectures

ANNs are collections of artificial neurons, typically organized in layers. Different layers may utilize different activation functions and/or apply different transformations to their inputs. Generally, the outputs of one layer's neurons are connected with the inputs of the following layer's neurons. If this holds true for all neurons in the ANN, the ANN is "fully connected". Alternatively, connections can be sparser, or loops between one or more layers can be created, giving the ANN different traits and capabilities.

When designing a layer, its position in the ANN is probably one of the most important variables. The input layer is the layer that accepts external data and is significantly dependent on the structure of the input; text input requires quite different management than visual input. The output layer is the

layer that generates the final result, and its primary design factor is the nature of the output, which can be a yes or no answer, a classification or a set of probabilities. Usually, in order to have a human-readable output, specialized activation functions like softmax are used.

Deep Neural Network (DNN)

Between the input and output layers, there can be zero or more "hidden" layers. Typically, the majority of the network's computation takes place in these layers, and their design is influenced by a variety of criteria such as the nature of the problem and the input, available processing resources, and the required minimum capabilities. A DNN is defined as a ANN that has multiple hidden layers.[21]

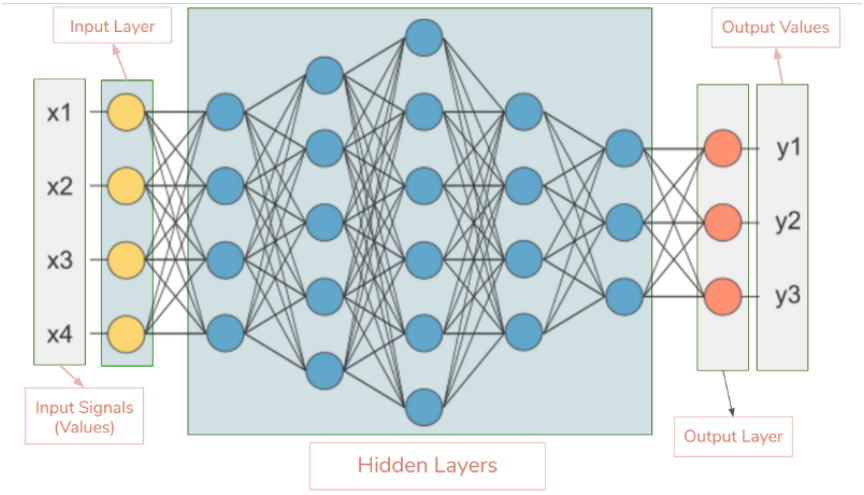

FIGURE 2.4: DNN with 5 hidden layers: URL.

Convolutional Neural Network (CNN)

The introduction of CNNs[22] is arguably one of the most significant achievements in the field of Deep Learning. They excibit great performance in image and video recognition, recommender systems, image classification, image segmentation, medical image analysis, natural language processing, brain-computer interfaces, and computer vision, among other applications. They perform best when the input is an image or a succession of images, but they are also effective in other scenarios.

CNNs are distinguished by their use of convolutional and subsampling layers, which enable the creation of multiple filters that can be trained in parallel. These filters are utilized to isolate and extract features from input data

that would be undetectable by simpler DNNs. Subsequently, in order to get a result, the output of these filters is fed to fully connected layers. The design and depth of these filters are directly responsible for the network's feature extraction capabilities.

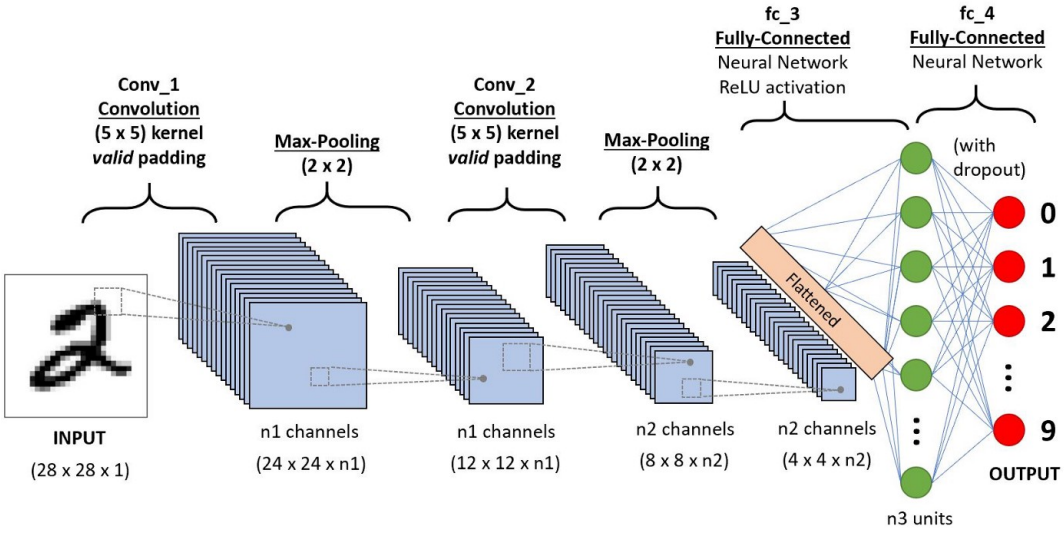

FIGURE 2.5: A CNN sequence that classifies handwritten digits using 2 convolutional layers: URL.

Convolutional layers carry out the convolution process with the help of small matrices known as kernels. The kernel is the beating heart of a layer, and its type and dimensionality determine how the layer functions. Typically, two-dimensional kernels are used, while their size is mainly depended on the size of the input and their position on the network. A single convolutional layer can usually only produce filters that detect generic low-level features, such as edges and color. In order to create more specialized filters that can detect high-level features, multiple layers are used.

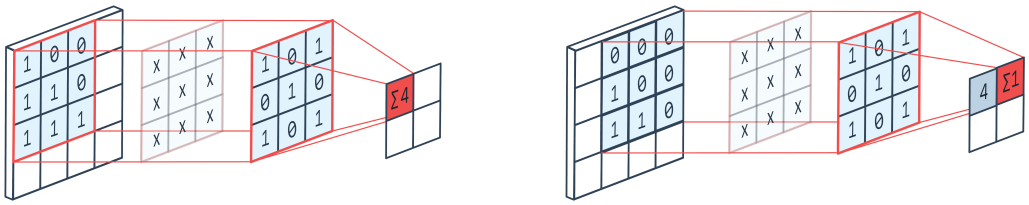

FIGURE 2.6: 2D convolution: URL.

The subsampling layers is the second distinguishing innovation of CNNs. Their primary task is to enable the network to recognize features without relying on their exact location. Furthermore, they simplify the network by reducing its number of parameters. Typically, they immediately follow convolutional layers in order to decrease the size of the features. Common subsambling layers include max pooling, mean pooling and others.

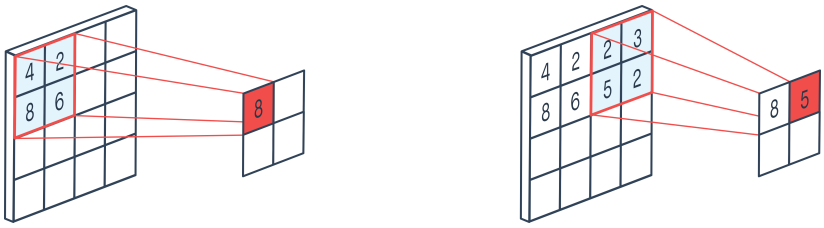

FIGURE 2.7: 2D max pooling: URL.

2.3 Training Artificial Neural Networks

As previously stated, ANNs are made up of neurons, which contain multiple parameters known as weights and biases, which are generally referred simply as weights. Training[2] is an iterative process that aims to improve the ANN's performance by optimizing these parameters. To accomplish this, three key elements are required: a loss function, an optimization algorithm such as gradient descent and a training algorithm like backpropagation.

In supervised learning, input-output examples are fed to the ANN. It produces predictions based on the inputs and then uses the loss function to compare these predictions to the intended outputs. The loss function calculates the ANN's error, which is a quantifiable difference between the expected and actual output. The error gradients of the ANN's weights are then determined, commonly using the backpropagation process. Finally, the optimization algorithm uses the error gradients to generate new values for the weights that should perform better.

In unsupervised learning, only input examples are given. The ANN attempts to mimic the data it is given and optimizes itself using the mistake in its output. Instead of a loss function, the error is represented as the likelihood of an incorrect output. The error gradients can be computed using a variety of learning algorithms, such as Maximum Likelihood, Maximum A Posteriori,

[2] Also called fitting.

and others, rather than the predominantly used backpropagation in supervised learning. Finally, to generate new values for the ANN's weights, any optimization algorithm may be employed.

In reinforcement learning, the ANN produces a prediction and subsequently receives a feedback[3], usually numerical, regarding its performance. The loss function uses this feedback and prediction, and like in the supervised learning, the error gradients are calculated through backpropagation. Finally, the ANN's weights are updated using a optimization algorithm.

The training technique varies greatly depending on the problem, the ANN architecture, and numerous other factors, but it is always iterative. An epoch is typically defined as using all of the data points in the training set once.

2.3.1 Initialization

The initialization of the ANN's weights has a significant impact on the ANN's final performance and training time. Naive methods, such as zeroing all weights or assigning them fully random values, might produce detrimental effects. If the weights in a ANN start out too small, the signal will shrink as it passes through each layer, eventually becoming too small to be useful. Likewise, if the weights in an ANN start out too large, the signal grows too huge as it goes through the layers, eventually overwhelming all other signals. As a result, the ANN may require a significant amount of training time or possibly become stuck in its initial state and not converge to a solution.

One common ANN initialization scheme used to solve this problem is called Glorot[4] Initialization[24, 25]. The idea is to initialize each variable with a small Gaussian value with mean of 0 and variance based on its fan-in and fan-out[5]. The Glorot Initialization not only outperforms uniform random initialization (in most circumstances), but it also eliminates the need to determine appropriate fixed limit values. There are actually two versions of Glorot initialization, Glorot uniform and Glorot normal, with different distribution and variance.

The variance of the Glorot Initialization is defined as:

[3] Feedback is frequently given after a series of predictions.
[4] also known as Xavier. [23]
[5] In a fully connected ANN, the fan-out of a layer equals the fan-in of the next layer.

$$V[W_i] = \frac{2}{n_i + n_{i+1}} \quad (2.6) \qquad V[W_i] = \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}} \quad (2.7)$$

Uniform distribution — Normal distribution

Where:

V = variance
i = layer
W = weights
n = fan-in of a layer

The Glorot initialization makes the assumption that the activations immediately after initialization are linear, as the initialized values are close to zero and their gradients close to 1. While this holds true for the traditional activation function its development was based on[6], it is invalid for the more modern rectifying nonlinearities[7] in which the non-linearity is at zero. As such, the He Initialization [26, 27] was proposed, with Gaussian distribution and the following variance:

$$V[W_i] = \frac{2}{n_i} \quad (2.8)$$

2.3.2 Loss Functions [8]

A loss function[28] provides a real number that represents the error a function associated with an event. In Deep Learning, it quantifies the inaccuracy of a ANN. The training algorithm tries to minimize this number by altering the ANN's weights, in hopes that it improves the network's accuracy. The choice of loss function is influenced by the nature of the input, as well as by the nature of the output. Some of the most common loss functions are listed below.

Regression Loss Functions

Regression problems involve predicting numerical values, a number or set of numbers. This is a usual problem in supervised learning. The appropriate loss functions measure the distance between the prediction and the ideal values.

[6] Sigmoid, tanh and softsign.
[7] ReLU and PReLU.
[8] Also called cost or error functions.

The most frequent regression loss function is Mean Squared Error (MSE). This method is utilized when the prediction belong to a continuous plane. The MSE is the mean of the squared distances between the predicted values and the target variables.

$$Loss = \frac{\sum_{i=1}^{n} \left(y_i^{target} - y_i^{pred.} \right)^2}{n} \tag{2.9}$$

When the data are discrete values, the Poisson loss function is more appropriate. Under the assumption that the target comes from a Poisson distribution, minimizing the Poisson loss is equivalent of maximizing the likelihood of the data.

$$Loss = \frac{1}{N} \sum_{i=0}^{N} \left(y_i^{pred.} - y_i^{target} \log y_i^{pred.} \right) \tag{2.10}$$

Classification Loss Functions

In classification problems, the examples must be classified into one or more classes, which may or may not be preset. The ANN generates a probability distribution that represents its confidence in the example's classification.

Binary cross-entropy is a loss function that is used in binary classification tasks with predefined classes. These are tasks that answer a question with only two choices.

$$Loss = -\frac{1}{\substack{output \\ size}} \sum_{i=1}^{\substack{output \\ size}} y_i^{target} \cdot \log y_i^{pred.} + \left(1 - y_i^{target}\right) \cdot \log \left(1 - y_i^{pred.}\right) \tag{2.11}$$

In problems with more than one classes, the categorical cross-entropy loss function, a generalization of binary cross-entropy loss function, is most commonly used. The y_i^{target} is the probability that event i occurs and the sum of all these probabilities is 1, meaning that exactly one event may occur.

$$Loss = -\sum_{i=1}^{\substack{output \\ size}} y_i^{target} \cdot \log y_i^{pred.} \tag{2.12}$$

Classification problems in unsupervised learning is quite different, as the desired output is not provided to the ANN. The most commonly used training

algorithm is k-means clustering. It aims to partition the examples into a predefined number of clusters. To achieve this it tries to minimize the pairwise squared deviations of points in the same cluster. The equivalent to a loss function is defined as:

$$\arg\min_{S} \sum_{i=1} k \frac{1}{|S_i|} \sum x, y \in S_i \, \|x - y\|^2 \tag{2.13}$$

Where:

S = clusters
x, y = points in cluster

Reward Functions

Reward functions serve the same goal as loss functions in that they quantify the accuracy of an ANN in order to optimize it, but in the opposite direction. Rather than penalizing the ANN, it rewards it based on its performance, with the learning algorithm aiming to maximize this reward. This is most frequently seen in Reinforcement Learning. Reward functions specify how the agent should behave, hence their structure is heavily influenced by the problem and the laws of the universe in which the agent lives.

2.3.3 Backpropagation

Backpropagation[29, 30] is a training algorithm for feedforward ANNs under supervised learning[9]. Feedforward ANNs refers to fully connected networks with no cyclical connections, most DNNs and CNNs adhere this standard. For a single example, backpropagation computes the gradient of the loss function with respect to the network weights. The gradient[31] represent the direction and rate of fastest rise. If a function's gradient is non-zero at a point, the gradient's direction is the direction in which the function increases the fastest, and the magnitude of the gradient is the rate of growth in that direction, in respect of that point.

Backpropagation is sometimes misconstrued to mean the entire learning algorithm for ANNs. Backpropagation is merely the method for computing the gradient; another algorithm, such as stochastic gradient descent, is needed to accomplish learning using this gradient. Furthermore, backpropagation

[9] Generalizations of the algorithm can be used for other network architectures and different training schemes.

is frequently misinterpreted as being limited to ANNs while, in fact, it may compute derivatives of any function. Its use to ANNs is critical because it enables efficient training, especially when using hardware accelerators.

The ANN can be mathematically expressed as:

$$g(x) \coloneqq f^L\left(W^L f^{L-1}\left(W^{L-1}\cdots f^1\left(W^1 x\right)\cdots\right)\right) \tag{2.14}$$

Where:

x = input
$g(x)$ = prediction
f^l = activation functions at layer l
W^l = weights at layer l
L = number of layers

Then the error function C with desired output y is:

$$C\left(y, f^L\left(W^L f^{L-1}\left(W^{L-1}\cdots f^1\left(W^1 x\right)\cdots\right)\right)\right) \tag{2.15}$$

By using the chain rule the total derivative of the loss function is:

$$\frac{\mathrm{d}C}{\mathrm{d}y} \circ \left(f^L\right)' \cdot W^L \circ \left(f^{L-1}\right)' \cdot W^{L-1} \cdots \left(f^1\right)' \cdot W^1 \tag{2.16}$$

Given that the gradient ∇ in respect to the input is the transpose of the derivative in respect to the output, the total gradient can be determined as:

$$\nabla_x C = \left(W^1\right)^T \cdot \left(f^1\right)' \cdots \circ \left(W^{L-1}\right)^T \cdot \left(f^{L-1}\right)' \circ \left(W^L\right)^T \cdot \left(f^L\right)' \circ \nabla_y C \tag{2.17}$$

The partial gradients at each layer δ^l, which represent the effect of the weights in the corresponding layers on the error function, may be easily determined by eliminating the effect of the previous ones:

$$\begin{aligned}
\delta^1 &= \left(f^1\right)' \circ \left(W^2\right)^T \cdot \left(f^2\right)' \cdots \circ \left(W^{L-1}\right)^T \cdot \left(f^{L-1}\right)' \circ \left(W^L\right)^T \cdot \left(f^L\right)' \circ \nabla_y C \\
\delta^2 &= \left(f^2\right)' \cdots \circ \left(W^{L-1}\right)^T \cdot \left(f^{L-1}\right)' \circ \left(W^L\right)^T \cdot \left(f^L\right)' \circ \nabla_y C \\
\delta^{L-1} &= \left(f^{L-1}\right)' \circ \left(W^L\right)^T \cdot \left(f^L\right)' \circ \nabla_y C \\
\delta^L &= \left(f^L\right)' \circ \nabla_y C
\end{aligned} \tag{2.18}$$

A naive approach would be to compute these derivatives forward. Backpropagation, on the other hand, eliminates duplicate multiplications by employing dynamic programming, as the derivative of one layer can be used to calculate the derivative of the previous one. Furthermore, by going backwards, a vector δ^l is multiplied by exactly one matrix $\left(W^l\right)^T \circ \left(f^{L-1}\right)'$ at each step. When calculating forwards, however, each multiplication multiplies a matrix with $L - l$ matrices, which is a far more expensive operation.

2.3.4 Gradient Descent

Gradient descent[32, 33] is an optimization algorithm which is commonly used to train ANNs. Gradients generated by training algorithms such as backpropagation are used to alter the network's weights, in order to produce the minimal possible error. Its basis is that a differentiable function F decreases fastest at a point a_n, in the direction of the negative gradient of that point $-\nabla F\left(a_n\right)$. Mathematically it is defined as:

$$a_{n+1} = a_n - \gamma \nabla F\left(a_n\right) \tag{2.19}$$

The learning rate parameter γ is the size of the step taken each time the algorithm is executed. It has a significant impact on the overall performance of the training procedure and should be fine-tuned. If it is too large, there is a high risk of overshooting the minimum of the function. If it is too small, more iterations are needed, and there is a risk too end up in a suboptimal local minimum.

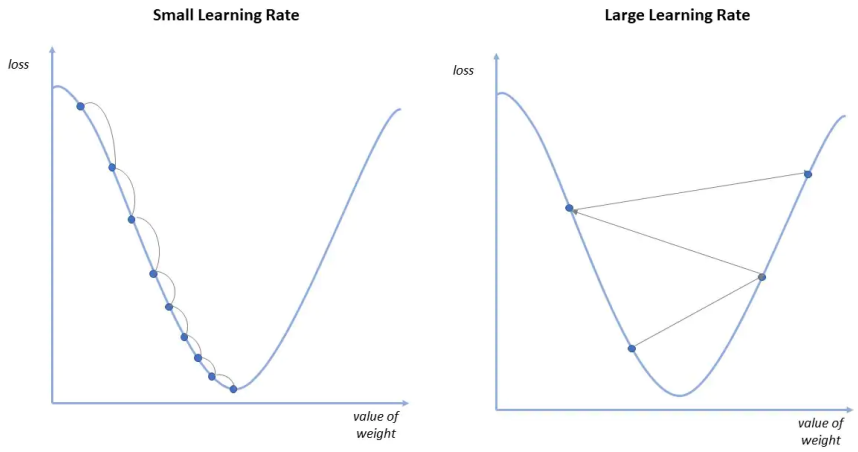

FIGURE 2.8: Effect of learning rate in Gradient Descent: URL.

Challenges

Gradient descent faces challenges with local minima and saddle points, when the gradient gets close to zero the algorithm is unable to re-adjust the network's weights. Local minima resemble the global minimum in shape, trapping the algorithm. Saddle points are stable positions with no relative maximum or minimum, making it difficult for the algorithm to decide what to do.

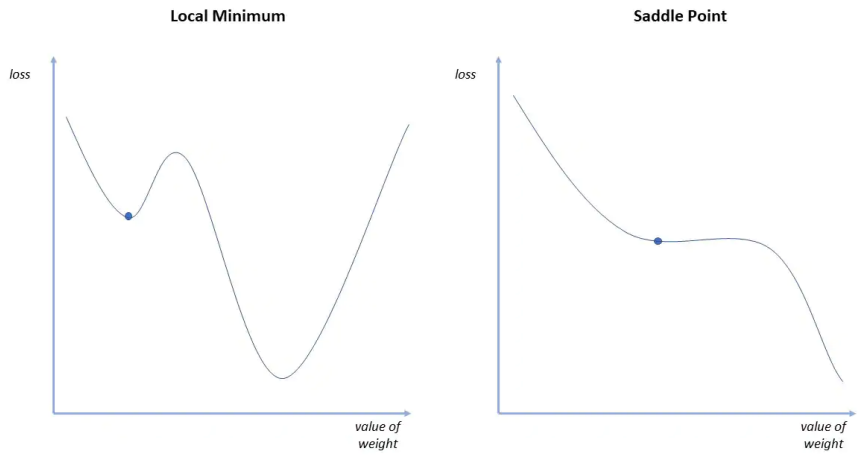

FIGURE 2.9: Local minimum and saddle point: URL.

To address this issue, a number of enhancements have been developed culminating in the Nesterov Momentum[34] extension. To accelerate the process, the first adjustment is to add a momentum variable, a percentage β of the previous iterations' change m. Simply adding that tends to result in overshooting. To mitigate this, the calculation of the gradient takes the momentum of the previous steps into account. With Nesterov momentum, the gradient descent is defined as:

$$\begin{aligned} m_{n+1} &= \beta m_n - \gamma \nabla F\left(a_n + \beta m_n\right) \\ a_{n+1} &= a_n + m_{n+1} \end{aligned} \tag{2.20}$$

In deep ANNs with numerous or repeating hidden layers, training with backpropagation and gradient descend introduce the phenomenon of vanishing gradients. As the algorithm travels backwards through the layers, the gradients get smaller and smaller, eventually becoming insignificant and unable to alter the weights of the network. Non monotonic activation functions, such

as ReLU, and more complex topologies, such as residual ANNs, are prevalent but not exclusive solution.

Another problem, especially frequent in RNNs, is exploding gradients. This occurs when a gradient gets too large, turning the model unstable. To address this, techniques such as dimensionality reduction have been developed, with the goal of reducing the model's overall complexity.

Variations

In vanilla Gradient Descent, each example's error is assessed, the gradients are produced and then the weights of the ANN are updated. In order to calculate the error of an example, the update of the prior one must be applied first. Since this process cannot be parallelized and must be repeated for each example, it is computationally inefficient.

Furthermore, the dataset is often used multiple times during the training of a model. In vanilla Gradient Descent, the examples are used in order. This pattern is often recognized by the models, which then introduce biases that lead to less-than-ideal solutions.

To address these inefficiencies, three key variations have been developed:

- Batch Gradient Descent

Batch gradient descent performs backpropagation and updates the network only after calculating the loss function for *all* the examples in the training dataset. The expensive operations of calculating the gradients and new weights occur just once per epoch, resulting in a computationally more efficient algorithm. Furthermore, the loss function can be parallelized indefinitely.

This method yields a stable error gradient and convergence, but it frequently leads to local minima. Furthermore, in order to calculate the loss, all of the data must be in memory, making the approach unsuitable for huge datasets. Finally, more passes through the dataset are needed, as updates are infrequent.

- Stochastic Gradient Descent (SGD)

SGD works similarly to vanilla Gradient Descent, with the exception that the training examples are chosen at random. This eliminates the bias produced by consuming the examples in a particular order. Furthermore, its frequent updates produce noisy gradients, which aid in avoiding local minima.

- Mini-batch Stochastic Gradient Descent

Mini-batch SGD builds on the ideas of the previous variations by splitting the training dataset randomly into small batches and performing updates on each one of them. This method achieves a balance between the computational efficiency of batch gradient descent and the randomness of SGD. This is by far the most popular variation, and it is commonly abbreviated just as SGD.

2.3.5 Model Overfitting

The goal of training ANNs is to improve their performance on real-world data, i.e. to generalize its knowledge. When training, the model[10] may sometimes fit exactly against training data, severely limiting its effectiveness with previously unseen data and negating its objective.

Training is typically conducted with a sample dataset. If the model trains on this for too long, or if the model is overly sophisticated, it may memorize irrelevant information, the "noise" within the training dataset. This is known as overfitting[35], and the most common signs are unusually high accuracy on the training dataset and high variance within the predictions of the network.

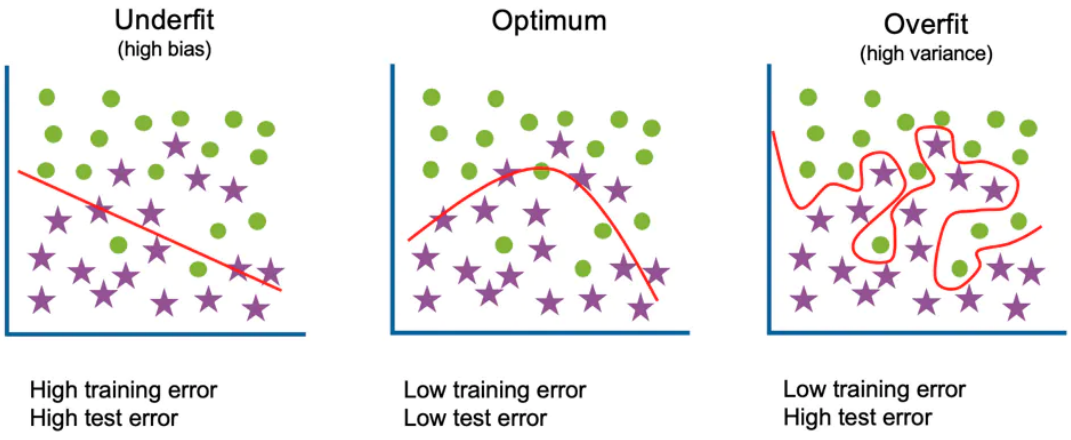

FIGURE 2.10: Model overfitting and its opposite, model underfitting: URL.

Multiple methods to avoid or suppress overfitting have been developed, some common ideas are listed below:

[10] Most statistical models, not just ANNs, exhibit this phenomenon.

- Early stopping

This method aims to stop the training before the model starts learning the noise within the model. To achieve this, a portion of the training dataset is held aside for testing rather than being used during training. This dataset is used to evaluate the ANN after each epoch, and if the accuracy is lower than before, training is terminated. There is a risk of stopping too soon and underfitting the model; a middle ground should be sought.

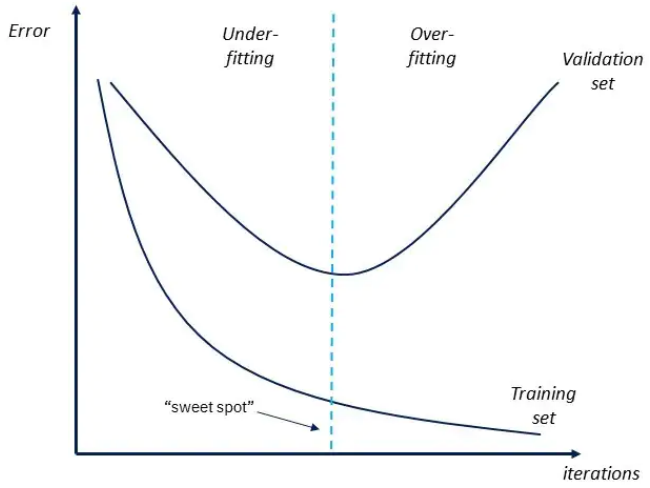

FIGURE 2.11: Border between overfitting and underfitting: URL.

- Data manipulation

A common way to reduce overfitting is through manipulating the input data. Expanding the training dataset with real-world or machine-generated data can assist the model in identifying patterns between the input and output variables. When using clean and relevant data, this strategy is effective; otherwise, the model may grow too complex and overfit even more. Another technique is augmenting already existing data by adding noise to them. The goal is to help the model discern between useful and irrelevant patterns.

- Model simplification

Multiple methods attempt to enhance the model's performance by simplifying it and the problem that is called to solve. Feature selection refers to a class of methods that enhance the training dataset by removing examples. Such methods include removing highly correlated features and incomplete examples, selecting the best features through statistical methods and others.

Another family of methods, such as the Principle Component Analysis, seek to transform the features by reducing their dimension.

The preceding methods necessitate some level of domain knowledge, which is not always available. In this scenario, regularization methods are particularly helpful, as they aim to reduce complexity by altering the model. In general they try to penalize input parameters with large coefficients, typical in examples with significant noise, in order to minimize the variance in the model. Such methods include L1 regularization, dropout and others.

2.4 Federated Learning

Federated Learning [4, 36] (FL) is a ML setting in which multiple clients, ranging from big enterprises to personal mobile devices, collaborate to train a model under the supervision of a central server. The goal of this is to alleviate many of the systemic privacy problems associated with centralization by decentralizing the training data. Under FL, any model that employs SGD-like approaches can be trained. ANNs, linear regression, Support Vector Machines, and other models fall into this category. FL acts as a wrapper for ML; what is true for a model when trained locally tends to hold true when trained in a FL context.

In general, the FL setting has two basic entities: data owners (participating clients) and model owners (orchestrating server). Participants never share their datasets, instead use them to locally train a model sent by the orchestrating server. The generated weights are shared, which the server aggregates them in order to construct a global model. The models trained by the clients are referred as local models whereas the aggregated model is referred as global model.

The entities are typically configured in a hub-and-spoke topology, as shown in figure 2.12, with the hub representing the coordinating server and the spokes connecting to the clients. The server organizes the training but never access the training data.

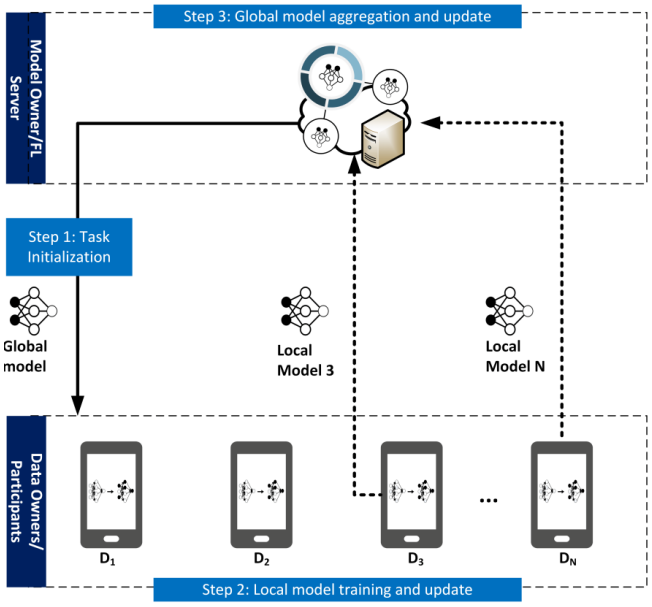

FIGURE 2.12: Typical FL topology: URL.

2.4.1 Typical Federated Training Process

FL training is a continuous process. Each iteration is referred as a global epoch (GE) and can be broken down into three main steps:

Task Initialization

Before any training can begin, the server has to complete a series of necessary tasks. It must first determine whether training should continue; if the target accuracy has been met or there are no available clients, there is no point to do. Furthermore, the server must specify any parameters or hyperparameters that are under its responsibility. FL design is flexible; factors such as learning rate may be controlled centrally by the server or by the clients.

After deciding how the training will proceed, the server must select N clients to participate. Clients may be chosen at random, based on eligibility requirements, etc. Finally, the server broadcasts the weights of the global model, together with any relevant metadata such as training parameters or a training program.

Local Training

Upon receiving the global model, each selected client locally computes an update to it using their private data. This update is referred to as a local

model. Training is carried out in accordance with any training parameters or programs that are provided. The objective f of a selected client n is to minimize its loss function L depending on the weights of the global model w_g and the local data d_i:

$$f_n\left(w_g\right) = \arg\min_{n \in N} L\left(w_g; d_n\right) \tag{2.21}$$

Subsequently, any required transformation may be applied to the local model. Such transformations include quantization and compression to reduce communication time, adding differential noise to increase privacy, and others. The finalized local model weights are sent to the server, together with any relevant statistics, and the client waits till it is selected once more.

Model Aggregation

The server collects and aggregates the local models to generate a new global model. The aggregation is implementation dependent; it might simply be averaging the models, or it could be biased toward some based on their statistics, how many times they have participated, and so on. The global model can be evaluated using server-accessible public data. The objective of the server is to minimize the global loss function:

$$F\left(w_g\right) = \frac{1}{N} \sum_{n=1}^{N} f_n\left(w_g\right) \tag{2.22}$$

This process is repeated until the global loss function converges, target accuracy has been achieved etc.

2.4.2 Federated Learning Settings

FL can be used in a broad array of applications with significantly diverse contexts and constraints [37, 38]. An example of FL across data centers could be hospitals that cooperatively train a cancer recognition model utilizing data from their patient diagnoses. Moreover, a real-world application of IoT FL is the training of a next-word prediction model for Google's Gboard [39] utilizing users' personal text messages. Table 2.1 seeks to describe two generalized FL scenarios and compare them with data center Distributed Learning (DL).

	Data center DL	Data center FL	IoT FL
Setting	Training is distributed among nodes in a data center. A centralized dataset is used.	Organizations collaborate to train a model utilizing data in their data centers.	A large number of IoT devices are utilized to train model with their private data.
Data Distribution	Data is balanced across nodes. Clients can access the whole dataset.	Data is created locally and is kept decentralized. A client cannot access other clients' data. Generally, data is not independently or identically distributed.	
Data Partition	Flexible, data can be repartitioned arbitrarily during training.	Fixed, partition axis can be by example or by feature.	Fixed partitioning by example.
Orchestration	Centrally orchestrated.	The training is organized by a central server, which has no access to the training data.	
Topology	Fully connected nodes in a cluster.	Typically hub-and-spoke.	
Scale	Typically 1 to 1000 nodes.	From a couple to a few hundred data centers.	Massively parallel, up to millions of clients.
Availability	Almost always available.		Only a fraction of the IoT devices is available at any single time.
Client Reliability	Few to no failures.		Unreliable, a part of the participating clients is expected to disconnect due to power or network issues.
Addressability	Clients are identifiable and can be addressed explicitly.		Generally unaddressable to enhance privacy.
Client Statefulness	Statefull, nodes can participate in every epoch, carrying state from one to the next.	Any, design depended.	Mostly stateless, clients will most likely participate in only one epoch before being replaced.
Primary Bottleneck	Computation. In a data center, a very fast network between nodes can be assumed.	Can be either computation or communication, problem depended.	Both, IoT tend to have low processing power and operate on slow connections (e.g. wifi).

TABLE 2.1: FL scenarios in comparison with data center distributed learning.

2.4.3 Unique characteristics & challenges of FL

Aside from the standard challenges associated with ML development, there are a number of obstacles specific to FL. These issues distinguish the federated setting from more traditional problems such as private data analysis and data center DL. [36, 37, 40, 38, 41]

Expensive communication

Communication is a critical bottleneck in many FL applications. In traditional data center DL, the communication environment is assumed to be perfect, with low latency, high bandwidth and negligible to no packet loss. This assumption is not appropriate to FL training, as clients are expected to be in different locations and with varying amounts of resources. This is especially true in edge FL, where the on-device datasets are small and connections are slow and unreliable, resulting to a high communication to computation ratio.

System heterogeneity

Client computational and communication capabilities can vary greatly in FL. They may differ in architecture (CPU, GPU, FPGA) and resources such as clock speed and RAM availability. Furthermore, they may be networked using different technologies (e.g., 4G, 5G, wifi) with varying reliability and bandwidth. Finally, some of them may be less eager to participate. All this leads to random and unpredictable client disconnections, as well as the appearance of "stragglers" [42], clients who take substantially longer to provide their updates than the rest and impede the entire process.

Statistical heterogeneity

It is frequently assumed in distribution optimization problems that the data are independent and identically distributed (IID). This is commonly violated in the federated setting, adding complexity to problem modeling, analysis, and solution evaluation. Data generation and collection are frequently uneven between clients, and the server is unable to collect or check the quality and distribution of their data due to privacy issues.

Privacy and Security concerns

The primary concern of FL apps is to protect the privacy of participating customers. However, malicious participants or third parties may be able to

infer sensitive information from shared parameters, defeating one of FL's key goals. Furthermore, its mostly assumed that all participants are well-intentioned, yet this is not always the case. Malicious clients may try to undermine the model's accuracy or install backdoors into it by utilizing poisonous datasets.

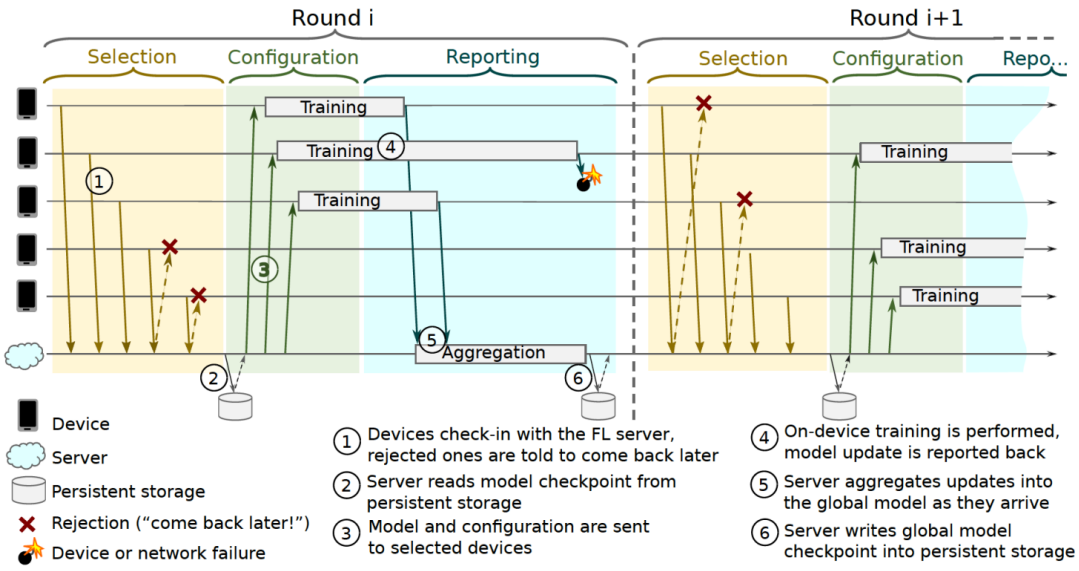

FIGURE 2.13: FL protocol with stragglers, client disconnections and client selection: URL.

2.4.4 Communication cost reduction

To achieve a satisfactory model in FL, multiple rounds of training and communication between the server and the clients are required. Communication can be a big bottleneck if the ANN being trained is massive and has millions of parameters, or if the clients are under slow connections. As a result, a series of techniques for improving communication efficiency have been developed, which can be classified into three groups: increasing computation, decreasing communication size, and decreasing communication frequency.

Edge and End Computation

Increasing parallelism by selecting more clients each GE is an obvious technique to increase computation in edge devices. In general, client-wise parallelism is desirable, but it provides diminishing returns as the number of participating clients increases. Furthermore, if all of the connected clients are participating, this strategy is no longer applicable. Finally, there is the risk of rapidly exhausting the available datasets.

The most typical technique to increase computation is to have clients perform more local model updates per GE. This can be achieved in two ways, more local epochs, i.e. more passes through the local dataset, or smaller batch size, i.e. more updates per pass through the local dataset. In traditional DL, such techniques tends to produce negative effects like overfitting. On the other hand, FL algorithms, due to their model averaging, produce regularization effects equivalent to dropout, ultimately increasing the accuracy of the model under training.

While these techniques are effective, having too many local updates between communication rounds can have a negative impact. To begin with, local models may diverge too much from each other, especially when under non-IID data distribution, significantly decreasing the convergent rate of the global model. As a result, additional training is required, defeating the aim of these techniques.

Another concern is that the likelihood of stragglers occurring is considerably increased due to client heterogeneity. Clients with fewer resources will take disproportionately longer to complete the additional computations, widening the gap between them and the faster ones. Since simulations are most often used when developing FL algorithms, researchers frequently overlook this issue.

Model Compression

These techniques, which are extensively employed in DL, attempt to decrease the amount of the communicated updates. The weights of the model under training make up the majority of the updates; applying transformations like sparification, quantization, and subsambling to them can reduce the size of the updates. In general, they are classified into two types: structured updates, which attempt to select what information is sent, and sketched updates, which attempt to compress the communicated information.

Structured updates require that updates adhere to a set, reduced format. This is possible in multiple ways, like putting a predermined per-client mask on the model after training to sparsify it. Another method is to instruct a client to train and communicate only specific layers or pieces of them. A more complex alternative is for the server to apply dropout to the global model in order to create a submodel, which the clients train, and the server maps back to the global model during aggregation. In general, these methods try to shift

the responsibility of compression to the server, with the aim to make it more predictable and correct its error.

Sketched updates refer to techniques that encode the update in one side and decode in the other. One such method is probabilistic quantization [43], in which the update matrices are vectorized and quantized for each scalar. Another option is to use a random mask like in a structure update, with the difference that it is randomly generated by the client and communicated to the server together with the encoded local model.

All these methods can be lossy and introduce error as a result of information loss. This error can be characterized as noise, which in most cases has a mean value of zero due to the nature of the compression algorithms commonly used. As such, the averaging of the FL algorithms can reduce it or even eliminate it from the accumulation of the local updates. For this to be true in practice, a large number of clients must be participating, which that is not always achievable. Moreover, as there is only one global model at any given time, any error in server-to-client communication cannot be reduced.

Importance-based Updating

By sending only what seems important, importance-based updating aims to reduce the volume of communication. Based on the observation that most parameters in an ANN are close to zero or hardly vary [44], this can be done in a fine-grained manner by sending to the server only a small percentage of the model parameters. It can also be applied in a coarse-grained way by asking clients to review their updates and send them only if they think they would help the overall model. These techniques have demonstrated that, when applied properly, they can occasionally reduce communication while also increasing accuracy and convergence rate. In contrast, they may produce the opposite effects if used excessively.

2.4.5 Systems heterogeneity

The previously discussed systemic characteristics of FL aggravate issues like straggler mitigation and fault tolerance. The slowest participating client has a significant impact on the duration of a GE. They can substantially impede training speed, thus removing them from the process is generally considered. Furthermore, if the server waits indefinitely for the client updates and a client disconnects without notifying, the entire system would hang. These issues

are especially prevalent in on-edge setting where there is little information or control over the clients' resources.

FL algorithms must be able to handle heterogeneous hardware and resist against random and unpredictable client drops. A frequent option is to disconnect clients who have not responded within a predetermined amount of time. Additionally, using more clients per epoch than necessary and accepting updates from those who respond first can help to eliminate stragglers. While effective, such strategies can induce biases in favour of the datasets of the fastest clients, reducing the overall accuracy of the model.

More sophisticated proposed solutions include intelligent client selection by only accepting clients that report their resources prior participating or tracking statistics on their overall performance. Such methods are not always feasible since they may jeopardize the privacy of the clients. A major portion of FL research employs simulations and avoid these problems, letting these challenges the least explored.

2.4.6 Data distribution

A dataset is independent and identically distributed (iid) if each example in it has the same probability distribution as the others and all are mutually independent. Models' accuracy, convergence rate, and fairness can all be degraded by training on non-iid datasets. Traditional ML avoids these issues by using a single, massive dataset that the designer is allowed to manipulate. On the other hand, FL encompasses a set of smaller datasets that could differ statistically from one another, and the designer may not be in control of or even aware of these differences. Ideally, a global dataset could be established by aggregating all of these small local datasets, however in reality this is impossible because the data cannot be centralized.

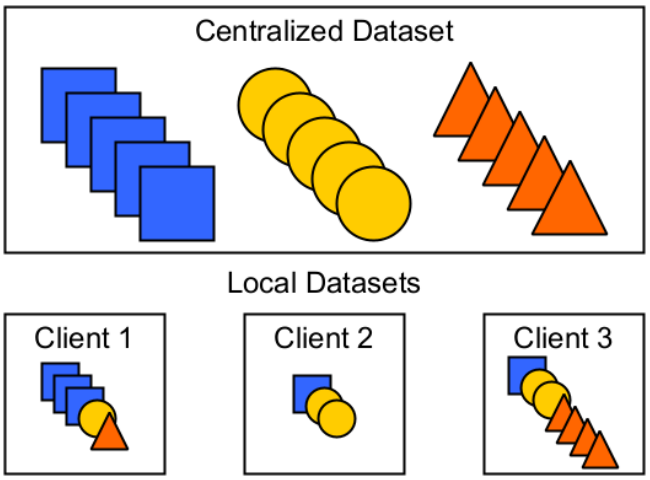

FIGURE 2.14: Centralized Dataset vs. Local Datasets. When distributed unevenly among the clients, the same data can behave as non-iid.

Non-identical client distributions

A global dataset used during FL may not be iid for a variety of reasons. First of all, a client's private dataset may locally violate independence, but this can usually be fixed via local shuffling. The statistical variations between the local datasets are more significant.

The distribution of a local dataset $P_i(x, y)$ can be rewritten as $P_i(y|x)P(x)$ and $P_i(x|y)P(y)$, where x and y are the input-output pairs. The distributions of at least two clients, I and j, must differ for the data to be non-iid, that is $P_i \neq P_j$. There are several causes for this, including:

- *Feature distribution skew* (covariate shift), $P_i(x) \neq P_j(x)$:
  Even if $P_i(y|x) = P_j(y|x)$, the marginal distributions $P(x)$ may differ. This is frequent in the domain of handwriting recognition, where two clients may write the same text in a different writing style.
- *Label distribution skew* (prior probability shift), $P_i(y) \neq P_j(y)$:
  Even if $P_i(x|y) = P_j(x|y)$, the marginal distributions $P(y)$ may differ. This is common when clients are bound to specific locations. Clients from different areas may use different terms and phrases depending on their local accent and lingo.
- *Same label, different features (concept drift)*, $P_i(x|y) \neq P_j(x|y)$:
  Even if $P_i(y) = P_j(y)$, the marginal distributions $P(x|y)$ may differ. The same label y can have different meaning for different users based on

their culture and standard of leaving. Images of clothes, for example, can vary greatly according on location.

- *Same features, different label (concept shift):*, $P_i(y|x) \neq P_j(y|x)$:
  Even if $P_i(x) = P_j(x)$, the marginal distributions $P(y|x)$ may differ. This is very common with labels that reflect sentiment. As an example, clients living in cold climates may describe the same weather phenomena differently than clients lining in warm or temperate climates.

- *Quantity skew*:
  Clients can generate and save different amounts of data. This is dependent on a variety of factors, including available data storage, local data retention laws, and the habits of their users.

- *Violations of independence*:
  Throughout training, the distribution may change at any time. Clients may connect or disconnect, or their local datasets may be exhausted. Furthermore, if clients are available at specific times of day due to solstice, a strong regional bias is imposed. Finally, because the clients own their own datasets, they can modify them at any time during training.

- *Dataset shift*:
  The FL participants might not be indicative of the end users. For instance, clients with inferior devices may be underrepresented as a result of any eligibility criteria used during client selection.

A real-world FL dataset may have any combination of these effects. Due to the difficulties of generating such datasets and examining algorithms built on them, most research tends to concentrate on one or two of them. Depending on the Fl scenario under training, different distribution skew effects may aplly and different mitigation strategies may be required.

Dealing with non-iid distributions

Existing algorithms can be modified, either by altering their parameters and hyperparameters or by sophisticating features like client selection. While adjusting other parameters, reducing batch size and increasing local epochs can be increase model accuracy, but excessive use may hurt convergence rate and lengthen training time. Metalearning approaches could be used to discover an ideal equilibrium.

It has been demonstrated that system heterogeneity and data heterogeneity interplay. By discarding straggles, unique and useful data could be wasted, degrading the model's fairness and accuracy. Stragglers can partially work, by personalizing parameters or reparameterizing on the fly, according to their resources. In this way, their local datasets can be exploited without slowing down the overall process.

Another approach is for the server to request data distribution statistics from the clients. With this information, the server can select those that will result in a balanced distribution. In addition, the server may be able to share some relevant publicly available data with the clients in order to rebalance their datasets. If no such data are available, willing clients may, if practical, provide their datasets to aid in the overall training process. These techniques can alleviate non-iid distribution problems, but they require additional communication and bandwidth. Additionally, they have the potential to compromise privacy, which would undermine one of FL's main goals.

Similarly, frameworks for multitask learning may be employed. Clients can train their local personalized models concurrently with the global model and share knowledge between them. Such techniques may not always be available as they demand greater processing and memory resources from the clients.

2.4.7 Privacy and Security

One of FL's key goals is to protect participants' privacy by simply requesting them to share model parameters and not any of their personal information. Additionally, FL wants to improve the model's fairness and accuracy by incorporating personal data. However, if any FL participant is malicious, these goals could be defeated. Model updates obtained from them can be used to reconstruct data, and poisoning attacks can corrupt the model.

Types of Attacks

- Data poisoning attacks
  In order to lower the accuracy of the model and add biases, these attacks introduce tainted data that violate the dataset's integrity. Model skew attacks [45] and feedback weaponization[46] fall within this group. The goal of model skew attacks is to decrease the model's accuracy by

obfuscating or distorting the boundaries between the classifiers. Feedback weaponization, on the other hand, tricks the model into misclassifying certain labels to introduce biases against them.

- Adversarial attacks [11]
  Adversarial attacks [47] involve specially crafted data that have designed to be consistently misclassified by the model. They can be subdivided to non-targeted and targeted. Non-targeted attacks try to evade being correctly classification during inference, any incorrect result is acceptable. Targeted attacks aim to incorporate backdoors into the model, that means manipulate the network to give specific erroneous inference for certain inputs.

- Inferring Attacks
  Attacks of this kind aim to gain information about the participants and their data, and can be classified into two categories. The first one is tracing attacks, which aim to detect whether a client is actively participating into training. The second one reconstruction attacks, which aim to recreate examples used in training, or features of them.

  Such attack can perpetrated using model extraction algortihms and Generative Adversarial Networks [48] (GANs). GANs is a ML technique where two competing ANNs are trained, a generator network and a discriminator network. The generator tries to create fake data while the discriminator tries to discern real data from the fake by analyzing the output of the discriminator. After some training, the generator can create data that closely resembles the real ones using the statistics learned by the classifier.

  A GAN attack seeks to infer as much useful information as possible about elements in a target class that are not under its possession. The GAN tries to mimic samples of that class, mislabels them and feeds them to the ANN under federated training. The rest of the participants must then work harder to discriminate between the target class and the mislabeled class, resulting in additional knowledge about the target class in the global model. This process is repeated until convergence, and the GAN has enough information to reliably reconstruct samples from the target class that approximate the original examples.

[11] Also called model poisoning.

This attack can be generalized to any number of classes and users, as well as any type of collaborative learning. If the server or another entity with access to the victim's communications is the malicious actor,it can be made more efficient by using the victim's local updates rather than the global model.

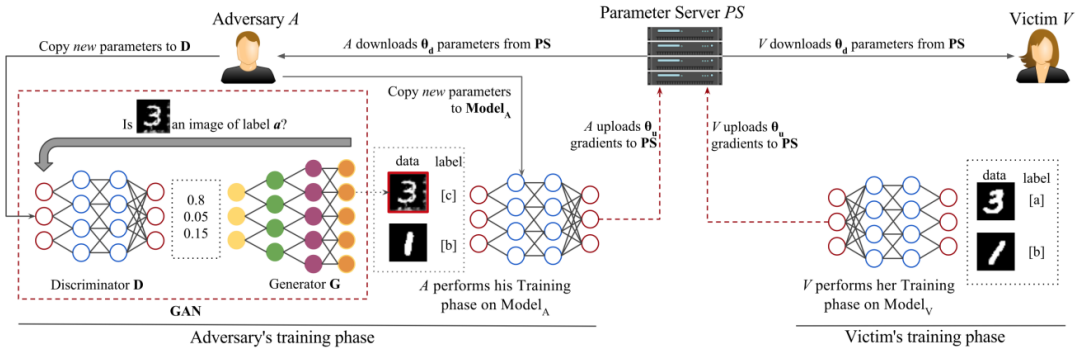

FIGURE 2.15: GAN attack. The adversary mimics images of class *a*, labels them as class *c* and uses them to train the collaborative model. To distinguish between these two classes, the model will require more information from the victim. The adversary does not need to have any true samples.: URL.

Countermeasures

FL algorithms are somewhat resistant to the aforementioned attacks due to the regularization effect of averaging local models. Data poisoning and adversarial attacks require a sizable portion of the dataset to be tainted in order to succeed. Additionally, inferring attacks demand that the malicious actor go through numerous training epochs. In an IoT environment since most clients, if they are even chosen, will only participate once. Even when several malicious clients collude, the is a very small probability of achieving their goals.

Such scale is quite challenging to achieve in an environment closer to the datacenter, making it much more vulnerable. Furthermore, clients might seek stronger privacy guarantees as the server might not always trustworthy. For these reasons, further measures for protecting privacy and security are necessary.

An additional level of security can be provided with minimal adjustments to the FL protocol, by scanning the clients' updates for unusual patterns. Repeated updates with outlandish values could be a sign that a client is attempting to corrupt the model. Furthermore, the updates of clients that try to

inject backdoors frequently resemble one another, which is rare, especially in a non-iid dataset. Such methods can improve the security of the model, but require plain-text access to the local updates, which is not always available due to privacy enhancements based on cryptography.

Clients might request further privacy protections, particularly if they don't trust the server or the connection between them. Secure Multi-Party Computation (MPC) [49], a subfield of cryptography, can be utilized to accomplish that. MPC simulate a trustworthy third party between two or more collaborating parties. Its homomorphic encryption techniques, which allow mathematical operations to be performed directly on cyphertexts and enable the server to aggregate the local models without having access to their plain-text contents, are particularly helpful. As a result, privacy can be ensured, albeit at a high computational cost that comes with cryptographic operations.

The state of the art method to enhance model security and restrict information exposure is differential privacy (DP) [48]. The fundamental tenet of DP is that by blurring a model's weights, they can not be associated with the data they were produced with. In FL, clients add random noise[12],with a mean value zero, to their local updates prior sharing them with the server. In addition of concealing the clients, this technique hinders dackdoor injections to the model, as the malicious clients need to send a precise set of parameters to achieve their goals.

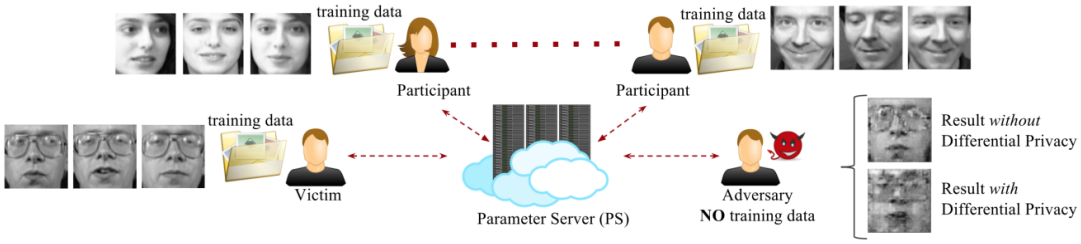

FIGURE 2.16: Effect of Differential Privacy on a GAN attack.: URL.

Theoretically, under DP transformation, the accuracy of the model and its convergence rate won't be impacted as the noise should vanish during aggregation. In practice, several samples are necessary to generate a distribution with a mean value close to zero, thus FL must be scaled appropriately.

[12]Usually Gaussian.

Finally, as attacks get more sophisticated, these countermeasures might not be effective and combinations of them or new ones are required.

Chapter 3

Related Work

3.1 Training Dataset

The training dataset is the most important element of the training process, as ML models directly extract knowledge from it. Regardless of the training algorithm, using inadequate data can only result in underperforming models. A well-known adage still holds brutally true when it comes to training data for ML: garbage in, garbage out.

In the context of this work, the Fashion-MNIST [50] dataset is utilized. It consists of 28 $\times$ 28 grayscale images of 70,000 fashion items from 10 equally sized categories, split into a training set of 60,000 images and a testing set of 10,000 images. It is designed as a direct drop-in replacement of the original MNIST dataset [51] that provides a more challenging classification problem.

The major factor of its popularity is its small size which enables DL researchers to swiftly prototype and test their algorithms. Furthermore, it is highly accessible due to its strait-forward encoding and its permissive license. Finally, DL frameworks (e.g TensorFlow) provide auxiliary functions and convenient examples that use it right out of the box, makes it highly compelling.

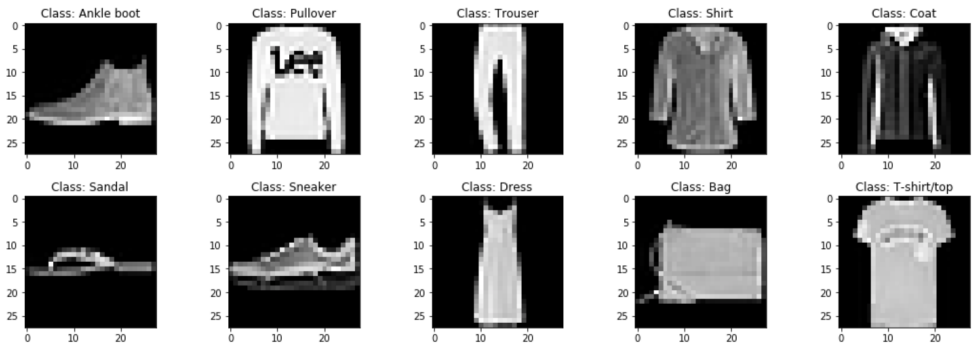

FIGURE 3.1: Examples of Fashion MNIST Dataset: URL.

3.2 ANN architectures

Any DL model that can be trained locally should be possible to be trained in FL, since FL is essentially another DL training method. To demonstate that the FL impementation in this work is accurate, multiple models of different architectures have been implemented and incorporated into the FL training loop.

3.2.1 LeNet-5

LeNet-5 [52], one of the first CNNs to describe its fundamental form, was proposed in 1989. Its original use was for handwritten digit recognition, a task in which it performed greatly and piqued academics' interest in the development and use of ANNs. It possesses the fundamental building blocks of CNNs, interconnected convolutional and pooling layers, followed by fully connected layers. Across all these layers it uses the tanh activation function, and to make the computation less difficult maintains sparse connections between them.

3.2.2 AlexNet

AlexNet [13] is a CNN architecture designed to compete in the ImageNet Large Scale Visual Recognition Challenge of 2012. The depth of the network's model, five convolutional layers, some of which were followed by max-pooling layers, and then three fully connected layers, allowed it to outperform its competition in terms of accuracy. Furthermore, it used the non-saturating ReLU activation function, which shows better performance than prior activation functions like tanh.

Training such a large network on a CPU, which was the standard at that time, is computationally prohibitive, but was made possible by training it on graphics processing units (GPUs). That novelty spurred huge interest in CNNs and training them with accelerators, making it one of the most influential ANN architectures.

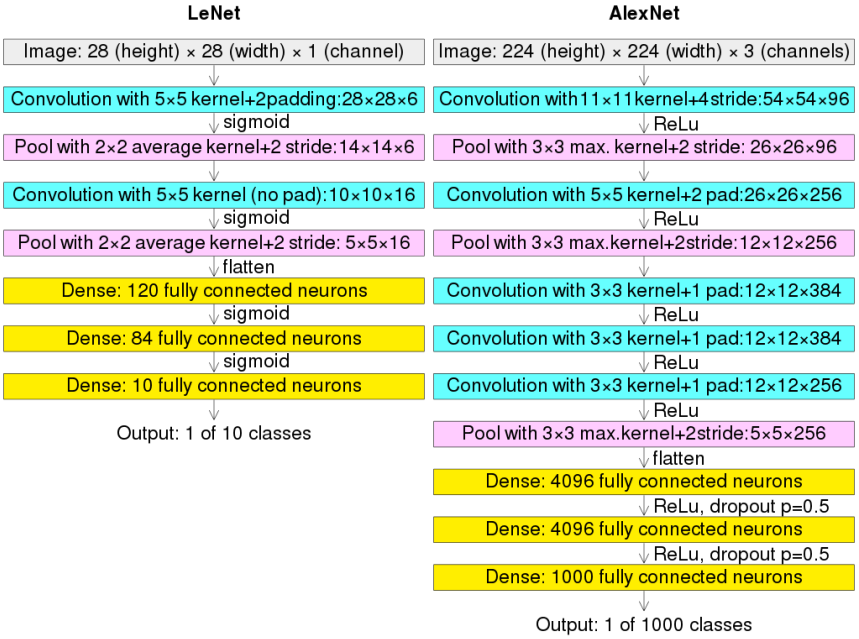

FIGURE 3.2: Comparing the LeNet and AlexNet architectures: URL.

3.2.3 ResNet

An ANN known as a residual neural network is distinguished by its shortcut connections that skip several layers. In this manner, it is possible to build ANNs that have hundreds of layers and are very deep without experiencing the vanishing gradients phenomenon. Additionally, it reduces the accuracy saturation issue, in which adding additional layers to a complex model causes the training error to increase.

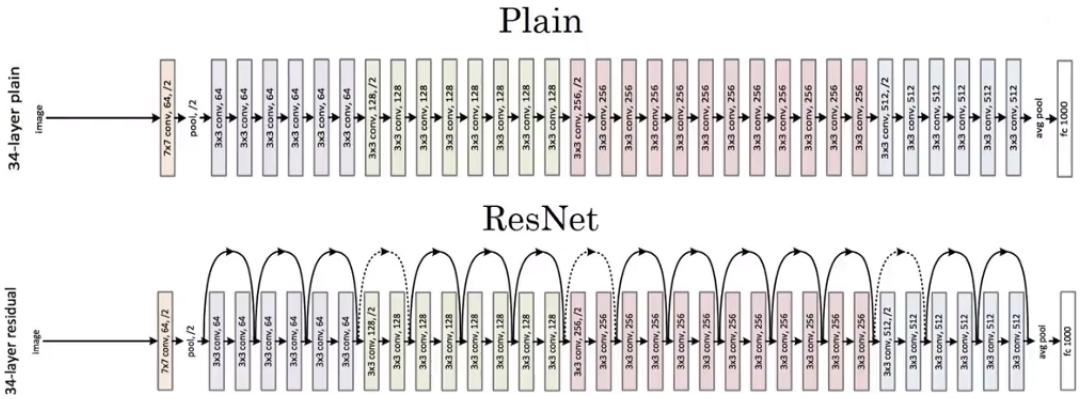

FIGURE 3.3: Comparing a ResNet and a plain model: URL.

3.2.4 Inception Module

In CNNs, the appropriate size of the filters depends on how the important information is located in the training data. Filters with small kernels are better in detecting information with a local distribution, while large kernels are preferred when dealing with a global distribution. Each sample may have a different distribution, making it challenging to select an ideal filter.

FIGURE 3.4: The dog, which is the important information, can occupy differently sized portions of the picture: URL.

The inception module [53] addresses this issue by providing multiple filters of various sizes that operate in parallel. These filters seek after the same information but in differently sized parts of the input. Consolidating their outputs is sufficient to determine whether one of them found the sought-after information because the one who identified it will overshadow the rest.

In its original form, it consists of three convolutional layers, with sizes of 1×1, 3×3, 5×5, as well as one 1×1 max pooling layer. To reduce computation, prior to the 3×3, 5×5 convolutions, and after the max pooling, additional 1×1 convolutionals are added. These layers are dual-purposed, as they apply dimension reduction and an extra layer of ReLU activations.

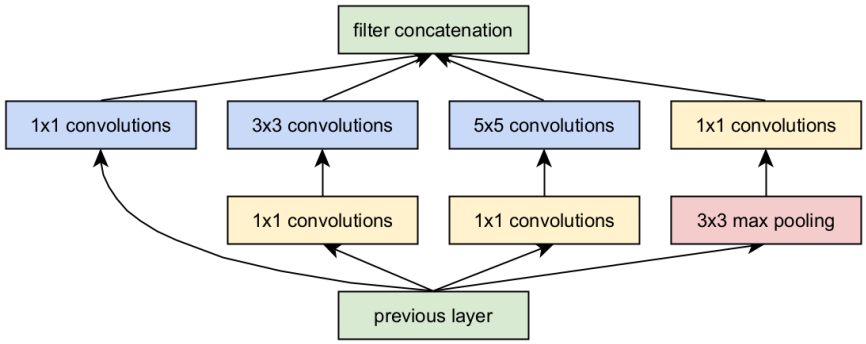

FIGURE 3.5: Architecture of the Inception module with dimension reductions.: URL.

3.3 Federated Learning Algorithms

3.3.1 Distributed SGD

Due to their iterative structure, conventional learning algorithms like SGD are inherently serial. Parallelism can only be applied over an example, e.g. pixels in a CNN, or over a mini-batch, where each example in it can be used in parallel. Synchronous SGD [54], a variation of mini-batch SGD, aims to enable cross-batch parallelism to reduce training time.

Before training starts, the dataset is distributed between N workers. In every iteration, each worker processes a mini-batch independently of the others, as follows:

- it fetches the up-to-date model parameters;
- it then computes new parameters[1] using a local mini-batch;
- finally, these parameters are sent to a synchronization service, generally a chief thread[2], that computes the new model parameters. When this is done, a new iteration begins.

Algorithm 1 `Distributed SGD`. The N workers are indexed by n; B is the local mini-batch size, w are the model weights, and η is the learning rate.

Synchronization service executes:
initialize w_0
$D_{1\ldots N} \leftarrow$ (distribute data to N workers)
for each round $t = 1, 2, \ldots$ **do**
  for each worker $n \in N$ **in parallel do**
    $w_{t+1}^n \leftarrow \text{WorkerUpdate}(n, w_t)$
  $w_{t+1} \leftarrow \frac{1}{N} \sum_{n=1}^{N} w_{t+1}^n$

WorkerUpdate(n, w) //Run on worker n
$b \leftarrow$ (1 batch of size B from dataset D_n)
$w \leftarrow w - \eta \nabla l(w; b)$
return w to synchronization service

[1] Instead of parameters, it can calculate and send gradients, letting the synchronization service do the backpropagation. This is true for most distributed or federated algorithms.

[2] As the workers are usually in a shared memory environment, this can also be done by one of them.

While initially intended for training in datacenters or localized clusters, it can easily be generalized for the federated setting by swapping out the workers for clients, the synchronization service with a server, and the distributed datasets with client generated ones. As a result, it became a major source of inspiration for FL.

3.3.2 FederatedAveraging

Many successful implementation of DL have relied on variations of SGD for optimization. As a matter of fact, they can be regarded as adaptations of the models and their loss functions to be more susceptible to optimization through simple gradient-based methods. Thus, it is natural that building FL algorithms begins with SGD.

Distributed SGD could naively be used in the federated setting, with each client computing the gradients of a single batch, a few samples, for each communication round. Although this method may be computationally efficient, it takes tens of thousands of training rounds to converge in a solution. This is prohibitive in a federated setting as communication costs are much higher than in a datacenter.

A straightforward method to reduce communication is expanding the batch until it includes the client's whole dataset. Thus, every client performs one full-batch (non-stochastic) gradient descent calculation per round. Furthermore, each client may have a different amount of examples, since in the federated setting, they are independently generated by the clients instead of being distributed by the server. As such, aggregation is weighted by the number of examples in each client. This approach is typically called FederatedSGD.

While FederatedSGD improves the computation-to-communication ratio, a number of other problems emerge. First of all, clients might not be able to meet the very high memory requirements of full-batch gradient descent. Furthermore, when under non-iid data distribution, convergence of the global model is not guaranteed due to high divergence between the local models. A more sophisticated approach, that overcomes these issues, is maintaining a more balanced batch size while performing several updates to the local models before sharing them with the server. This method is referred as FederatedAveraging (FedAvg) [4] and is the cornerstone of FL, as most FL algorithms are its derivatives.

Algorithm 2 `FederatedAveraging`. The N client are indexed by n; k is the size of the local datasets, while K is their total size; E is the number of local epochs, B is the local mini-batch size, w are the model weights, and η is the learning rate.

Server executes:
initialize w_0
for each round $t = 1, 2, \dots$ **do**
$S_t \leftarrow$ (set of selected clients)
for each client $n \in S_t$ **in parallel do**
$w_{t+1}^n \leftarrow \text{ClientUpdate}(n, w_t)$
$w_{t+1} \leftarrow \sum_{n=1}^{N} \frac{k_n}{K} w_{t+1}^n$

ClientUpdate(n, w)**:** //Run on client n
$S_b \leftarrow$ (split local dataset into batches of size B)
for each local epoch i from i to E **do**
for each $b \in S_b$ **do**
$w \leftarrow w - \eta \nabla l(w; b)$
return w to server

3.4 The FPGA Perspective

In our knowledge, no prior works have attempted to merge FL and FPGAs. Modern system-on-chip (SoC) FPGAs contain all the tools required to implement FL, including running on an operating system and having an Ethernet adaptor. As such, connecting the two technologies should be feasible.

FL on the FPGA shouldn't be substantially different from centralized training in terms of design or implementation. In contrast, the driver of the reprogrammable hardware will differ, as it must facilitate the FL procedure. Furthermore, The FL algorithm itself is not computationally demanding and may be conveniently offloaded to the SoC's CPU.

3.5 Thesis Approach

As the thesis moves forward, conflicts in terms of design and implementation are anticipated to arise between the two technologies. Furthermore, this is an mostly unexplored field. As such, a conservative and steady approach is expected to work best.

Initially, a FL implementation that is agnostic and independent of the underlying training implementation, will be developed. Its robustness will be thoroughly validated, using TensorFlow to facilitate the local training. Subsequently, an FPGA-based CNN training implementation will be created. Finally, by combining the networking code of the FL clients with the FPGA driver, an intermediate layer will connect the previous two implementations.

Chapter 4

FL architecture & design

4.1 Software

4.1.1 Tensorflow & Keras

TensorFlow [55] is an interface for expressing ML algorithms and an implementation for executing such algorithms. It offers a complete, flexible ecosystem of tools, libraries and community resources that that facilitates the development and deployment of ML powered applications. Its main advantage is the ability to use high-level APIs like Keras with eager execution, enabling immediate model iteration and easy debugging.

Tesnorflow & Keras were used in all experiments during modelization. It is chosen due to its simple, flexible architecture, which turns new ideas into code quickly. In addition, due to the existence of TensorFlow Federated (TFF) [56] framework, there are many compatible theoretical resources and tutorials. TFF is simulating FL to facilitate research and experimentation with FL algorithms, thus it is incompatible with this work which aims to implement real-world FL with hardware accelerators.

4.1.2 Python/C API

As the goal is to integrate FL with FPGA accelerators, the majority of the codebase is developed in C++. This include all the networking, communication, model aggregation and any required model transformations. TensorFlow on Python is utilized for model evaluation and, throughout the modelization phase, for training. To connect these two components, the Python interpreter is embedded to the core program using the Python/C API [57, 58].

With the TensorFlow C API [59], TensorFlow could be used directly in C++, however several capabilities, like the Neural Network library, are not supported. Furthermore, quickly rotating among ANN architectures, training techniques, etc. is quite usual in FL development. With the C API that becomes tedious and slow, since it is geared more toward uniformity and simplicity than convenience, and C++ needs to be recompiled after every change. Due to these factors, integrating the Python interpreter and using TensorFlow in Python is considered as a more appropriate solution.

4.1.3 POSIX sockets

POSIX sockets [60] is an application programming interface (API) for Internet and Unix domain sockets, used for inter-process communication (IPC). A socket is an abstract representation for the local endpoint of a network communication path. According to the Unix philosophy, the POSIX sockets API defines it as a file descriptor that offers a standard interface for input and output to data streams.

The 4.2 Berkeley Software Distribution [61] Unix operating system, which was introduced in 1983, is where the API originates from. POSIX sockets transitioned mostly unchanged from a de facto standard to a POSIX specification component. They are commonly referred to as "Berkeley sockets" or "BSD sockets" to acknowledge the Berkeley Software Distribution, where they were first implemented.

In FL, entities possess their own private data. This is best implemented through processes with private data space that communicate using sockets. Therefore, the POSIX socket API implementation provided by the LINUX operating system is used for all inter-entity communication.

POSIX sockets can be configured for blocking or non-blocking operation. In blocking operation, the program halts until the entire message is sent or received. In contrast, during non-blocking operation they only retrieve or send data that is immediately available. Thus, the program does not stall on straggler connections and many deadlock situations are avoided, but there is no guarantees that the messages will be send or received in one piece, especially when said messages are large [1].

[1] This is due to limited sized socket buffers set up by the operating systems.

4.2 Data Preparation

4.2.1 Normalization

Dataset normalization [62], as part of data preparation, is a standard practice in ML. Normalization transforms the features of a dataset to a common scale, without distorting discrepancies in the ranges of values or losing information. This technique prevents large scaled characteristics to dominate during training. Furthermore, many algorithms, such as ReLU non-linearities, exhibit better performance when fed with data of floating-point format.

In this work, the Fashion-MNIST dataset provided by TensorFlow Datasets [63] collection is utilized. It is consisted of gray-scale images, where each pixel is represented by an integer in the range $[0, 255]$. They are normalized to floating-point format in the range $[0, 1]$ with the script `prepare_dataset.py`. Furthermore, to avoid repeating this procedure for every experiment, the processed dataset is saved on disk.

4.2.2 Distribution

In FL, each client is meant to have their own unique, individualized dataset. Given that the provided Fashion-MNIST dataset is a single, concentrated collection, it must be distributed among the clients in order for federated training to be possible. Two approaches of partitioning the data among the clients are explored:

IID

The data are randomly partitioned in equally sized shards, one for every client. For example, if there are 10 clients, each will receive a shard containing 6000 examples. Although this distribution is not IID in the strictest sense[2], it is closer to a real-world scenario and many issues, such as class underrepresentation, can be easily avoided.

non-IID

Although statistical challenges are not the focus of this study, some testing with non-IID data has been done for sake of completeness. The dataset is

[2]Due the shards being mutually exclusive, knowing that an example belongs to one of them indicates that it does not exist in others shards. Thus, knowledge about the other local datasets can be inferred and independence is violated.

broken up into shards, each of which includes examples from only one label. Each client receives two shards of different labels. If there are 10 clients, for instance, twenty shards will be produced, and each client will receive 3000 examples from two labels for a total of 6000 examples. Despite such a pathological non-IID distribution being atypical of a real-world scenario, it will assist investigate how severely the algorithms fail on extremely non-IID data.

4.2.3 Pipeline

The input pipeline that feeds the training data to the models is constructed using the `tf.data` API provided by TensorFlow. More specifically, before training begins, each client optimizes the use of its dataset by transforming it through caching, shuffling, batching, prefetching, and repeating. Additionally, this process is parameterized for flexibility and enable experimentation with different local dataset and batch sizes.

4.3 Embedding the Python Interpreter

As mentioned in section 4.1.2, the Python Interpreter is embedded on top of the C++ codebase. To make this integration as seamless as possible from both sides, an integration layer that operates as a wrapper for the C/Python API, has been developed. The C++ codebase can call Python code with simple function calls, while the Python code can access data from the C++ space like it would access data from its own space.

To achieve this, A number of steps need to be completed. First of all, a straitforward abstract class is defined, which specifies a train and an evaluate function, as well as an input and an output model. The C++ codebase is interfacing with an implementation of this class. Its tasks include initializing the Python interpreter, loading the appropriate Python module, passing the necessary data and creating C++ function wrappers for the Python function.

Moving data from one side to the other can be trickier than it first appears. Using the appropriate API calls, such as `PyModule_AddIntConstant`, simple constants and macros can be passed by copy to the Python module in a straitforward manner. This approach fails when dealing with large amounts of data, such as the model parameters. Instead, by constructing NumPy array metadata over them and copying them, they can be passed by reference. In

this manner, both ends observe the same memory space and there is no significant data copy.

After exposing the parameters to the Python code, one more step is necessary to enable the TesnorFlow library to be able to use them. In order to assign the received parameters to the model under training, they must be first transformed into TesnorFlow tensors with dimensions and shapes that match its layers. Likewise, to extract parameters from a model and expose them to the C++ codebase, its layers must be concated in a NumPy array.

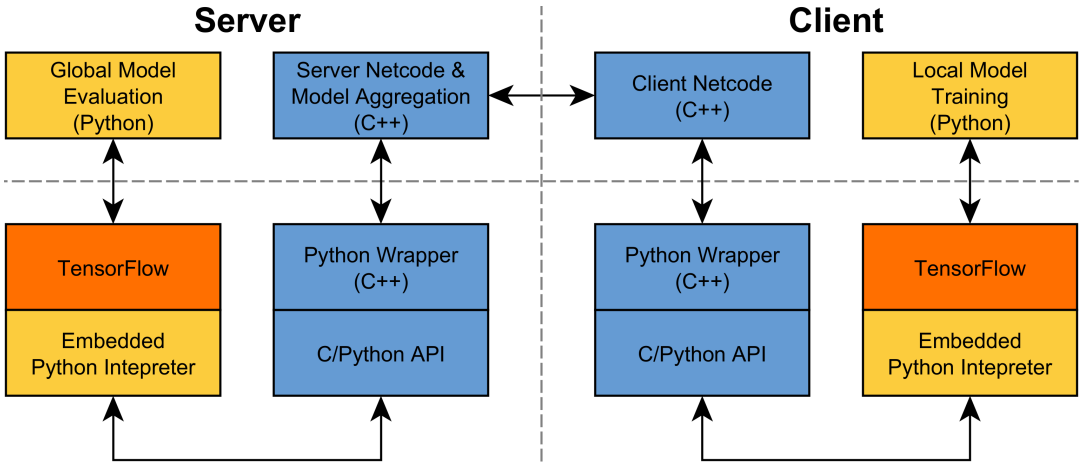

FIGURE 4.1: Overview of the C++/Python Integration. The FL protocol implementation components are represented in the top half, while the required libraries, APIs, and wrappers are shown in the bottom half.

4.4 FL Architecture

The architecture aims to offer a generalized FL loop that enables the implementation of various FL algorithms. To achieve this, it is designed to be flexible and modular, with each FL operation, such as client selection and aggregation, having its own specialized function. Additionally, all relevant training parameters and hyper-parameters, such as local epochs or participating clients per epoch, are compiled in the `definitions.hpp` file. Since the entire codebase accesses them from there, testing and experimentation are streamlined and less prone to mistakes.

In FL, multiple entities are present, the orchestrating server and the clients training the global model. As the aim of this work is to implement FL with

clients operating on separate devices, it is essential that each entity is a distinct process with its own private data-space. That data-space contains its private training or testing dataset, as well as its local or global models. All required communication is facilitated through POSIX sockets.

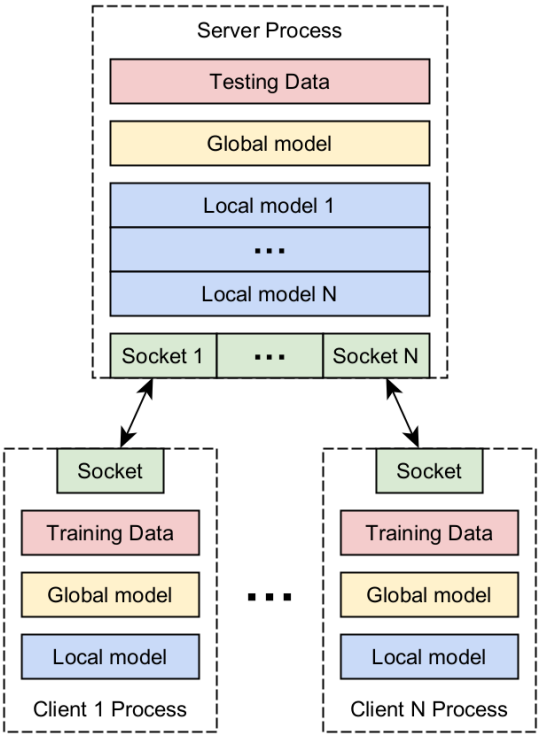

FIGURE 4.2: Process and memory layout of the FL architecture. Each client holds their private data, the global model and the models they produced. Meanwhile, the server holds the testing dataset, the global model and the most recent local model it received. All communication goes through dedicated sockets.

4.4.1 Server

Overview

The server generally adheres to the event-driven server paradigm. The process, while sleeping, listens for events such as new connections or messages from the clients, and reacts according to their context. This is very similar to FL in that the server receives local models, aggregates them, and then, after accumulating a sufficient number of them, creates a new global model and announces it to the clients. All these actions are triggered by client updates.

The server process is the focal point of FL and is responsible for several tasks, which can be distinguished between algorithmic, systemic and auxiliary. Algorithmic tasks are the components of the FL algorithm, such as model aggregation. Systemic tasks are necessary operations to implement the FL algorithm, such as connecting sockets. In addition, some tasks that are not required to implement the algorithm are included in order to enhance its utility and ease development.

Operation

The server's first action is to load a pre-trained model, if one exists. While not a prerequisite to facilitate FL, this is done to enable transfer learning and experimenting with retraining a model under different settings.

After that, the server completes a series of initializations. First of all, a listening socket is set-up in non-blocking operation, and the event-driven structure is established. Furthermore, the Python environment, where the global models are evaluated, is embedded and initialized. Finally, any structures or variables required by the FL algorithm are initialized.

After the initializations, the server enters a waiting state. To achieve this the `poll(2)` system call, which puts the process to sleep until an event occurs, is used. Four types of events may happen:

- The listening socket encounters a new connection, meaning a new client requests to join in the federated training. The socket is cloned, the clone establish the connection with the client, and any necessary data structures are created.
- A connected socket encounters an error, such as an sudden disconnection. Unreliable clients are expected to continue being unreliable, thus the most prudent course of action is discarding them.
- A connected socket receives new data. As a message can consist of millions of weights, it may be received across multiple events and a collection mechanism is needed to fully retrieve it. To achieve this, it is necessary to track the size of the received data per client and ensure that there is always adequate memory available to store a message from each connected client. If the message is complete and valid, its local model is aggregated to next global model, and the related client is considered as non-working.

- A connected socket can send new data. This indicates that a socket designated to send the global model to its connected client, is available to do so. As mentioned before, the messages can be quite large, thus multiple events may be required to fully send them. To achieve this, tracking of the amount of transmitted data per connection is necessary. When a message is fully send, the related client is considered as working. Furthermore, as there no more data to send, the `POLLOUT` flag of the socket is disabled.

Following any event, the server determines whether a new epoch should begin. It takes in consideration how many local models were successfully received this epoch, how many clients are connected, and how many clients are still working. If the current epoch requires further work, the process returns to the waiting state and sleeps until a new event occurs.

If the contrary is true, the new global model is created by dividing the aggregated local models by the number of received local models during the current epoch. This new model is evaluated, and then shared with clients that where randomly selected using the Durstenfeld-Fisher-Yates shuffle algorithm [64, 65]. The only action needed to share the model with a client is enabling the `POLLOUT` flag of the corresponding socket; the event loop will handle sending the message.

The event loop's final step is to determine whether any further training is required. If the target accuracy is achieved or a predetermined number of GEs have been completed, the server shows any relevant statistics, stores the final global model to disk, and shuts down.

4.4.2 Client

Overview

During an epoch, a participating client receives a global model, goes through a few local training rounds, and then sends the updated local model back to the server. Training cannot begin until the global model is fully received. Furthermore, after sending the local model, nothing further needs to be done until a new global model is received.

As a result, the client is controlled by its communication with the server, and a master-slave relationship is formed between them. To effectively implement this, client-side communication is blocking, meaning a client can not take any action until it has fully received or send its messages.

Operation

At startup, the client creates a socket and connects to the server. Additionally, it embeds and initializes the Python environment that is used for training. Following these initializations, the process moves into its main loop.

The main loop contains three major operation. First, it receives the global model shared by the server. Then, it is trained with the local private data, creating a new local model. Finally, any required transformations, such as quantization and compression, are applied to the new model, which is then send to the server. This process is repeated until the server informs that there will be no more training with that client.

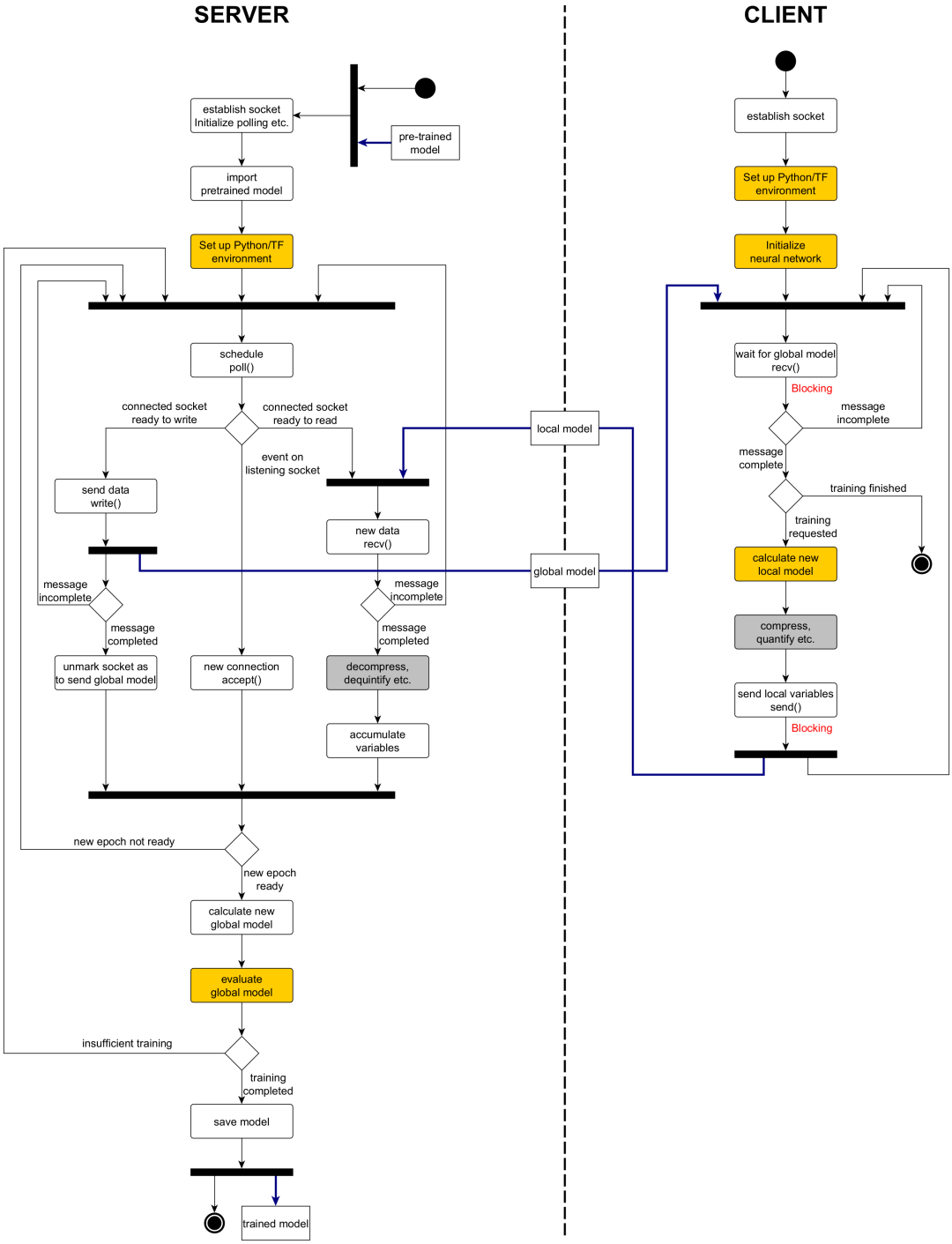

FIGURE 4.3: Server - Client Activity Diagram: Yellow states are modelled with TensorFlow, while grey states are not essential for FL. Blue arrows represent data movements. Error conditions and states are not displayed.

4.4.3 Communication Scheme

As stated in section 4.4, the server and the clients are separate processes that communicate over sockets. A concrete, predefined communication system is needed to accomplish this in a reliable manner. The server holds the barest amount of information on the clients, just what is necessary to stay in connection and communicate, to adhere to the cross-device FL setting. As such, the server can not address clients directly and messages must be generic. Furthermore, each message must be independent from the rest and self sufficient. As a result, messages sent by the server to clients must be general and self-sufficient.

Algorithmic solutions can reduce communication, but communication must be kept to a minimum in systemic level too. The messages, to be as compact as possible, only include their model and a few bytes of metadata required by the FL algorithm. Furthermore, they are C-aligned arrays, which means there are no delimiters between their values, or hidden metadata from predefined protocols of higher abstraction, such as Protobuf.

Minimizing communication frequency is another strategy used for cutting down on communication time. Any message send by the server that contains the global model, can be interpreted as an request to train it. Furthermore, if a client sends its local model, it can be presumed that it completed its task. As a result, each epoch only these two messages are required, and any synchronization or confirmation messages are unnecessary. With this approach, it is necessary for every party to interpret the messages in a same predefined way.

Server to client message
flags
GE
global model variables
...

Client to server message
GE
local loss
local accuracy
model variables / deltas
...

TABLE 4.1: The format of the communication between the server and clients.

The format of the messages is shown in Table 4.1. The flags field is intended to communicate particular instructions to clients, such as the message is the final one and no more communication will be accepted or that the client should initialize the model. The GE field is used to discard stragglers, as

the server can quickly reject any messages from an earlier GE. The local loss and accuracy fields are used to facilitate complex algorithms, such as ignoring local models with poor accuracy or higher loss than the prior GE. The global and local model parameters are the last part of the format and make up the bulk of the messages.

4.4.4 Model Library

Most ML models, if not all of them, are meant to be trainable in a federated environment. To demonstrate the accuracy of the developed FL environment, a library of ten typical models has been created. As the training problem is image recognition, the majority of the models are CNNs. However, models of different architectures, such as deep and residual ANNs, are also included.

1. The simplest model in the library is a DNN architecture. It consists of three fully connected ReLU activated layers with 128, 1024 and 128 neurons respectively, followed by a Softmax layer. In total, it contains 365,066 weights for an approximate size of 1.46 MBytes.

2. The first CNN model follows the original LeNet-5 architecture. It has two convolutional layers of 6 and 16 5×5 kernels, each one accompanied by an average pooling layer with 2×2 pool size. They are followed by two fully connected layers of 120 and 84 neurons, and a Softmax layer. All layers, except the final one, are activated with the hyperbolic tangent function. In total, it contains 61,706 weights for an approximate size of 0.25 MBytes.

3. For the following experiments, the model most used is a CNN architecture consisting of two convolutional ReLU activated layers of 32 and 64 3×3 kernels, each accompanied by a max pooling layer with 2×2 pool size. They are followed a 128-neuron fully connected ReLU activated layer, and a Softmax layer. It contains 421,642 weights for an approximate size of 1.69 MBytes. This architecture is compact enough to enable rapid experimentation and testing while being sufficiently sophisticated to provide an acceptable level of accuracy and necessitate several training epochs.

4. The CNN used in the original FL work [4] is also included in the model library. Its architecture is fairly similar with the previous one, but with

larger 5×5 kernels, and a fully connected layer of 512 neurons. In total, it contains 1,663,370 weights for an approximate size of 6.65 MBytes.

5. The next model included in the library aims to evaluate the FL environment with more sophisticated layers and combinations between them. It employs six convolutional layers, applies batch normalization on their outputs, and uses max pooling every two convolutions. There are 803,240 weights in it, giving it an approximate size of 3.2 MBytes.

6. To test the FL environment with extremely large models, the AlexNet architecture have been implemented. The model consists of 46,764,746 weights for a message size of 187 MBytes. As a result, it is unfeasible to train it repeatedly, as the FL operation needs, with the current available resources. Instead, it was trained for a single epoch with a few training data and conservative hyperparameters, just to demonstrate that the FL environment has no model size constraints.

7. For similar reasons the OverFeat-AlexNet architecture is included. This model is the largest one in the library, with 56,906,954 weights and a total size of 227 MBytes. The same constrains apllies.

8. The inception module detailed in section 3.2.4 is the foundation for two of the included models. The first one comprises of two such modules of different sizes and a Softmax layer. It has a total of 4,275,914 parameters and is about 17.1 MBytes in size.

9. The second inception architecture includes a module sandwiched between two convolutional layers, and max pools the outputs of all three. The output of the module is also subjected to the dropout transformation. Furthermore, they are followed by two fully connected layers and then a Softmax layer. In total, there are 277,082 weights for a size of 1.1 MBytes.

10. The final model is based on the residual architecture. It is consisted of two convolutional layers and a Softmax layers. The input of the network feeds the convolutional layers, but it also skips them and is directly connected to the Softmax layer. Furthermore, the dropout transformation is applied to the input of the Softmax layer. It has 539,466 parameters and is about 2.16 MBytes in size.

Chapter 5

Robustness Analysis

A number of experiments, each concentrating on a different component of the FL algorithm, have been conducted to demonstrate the robustness of the developed FL environment. Additionally, the tests begin with straightforward cases and progress to more complicated ones by building on their findings. The training problem is image recognition on the Fashion-MNIST dataset.

As the main goal of these experiments is to prove the algorithmic soundness of the FL environment, they were carried out on a single machine. As a result, the participating processes are competing for computing resources, and communication takes place on the operating system's loopback. Thus, it is impossible to draw any meaningful real-time inferences from these experiments; instead the communication frequency is used as a benchmark value.

5.1 Distributed SGD with IID data

The first experiment focuses on the most straitforward case, distributed SGD with IID data. The training dataset is split equally between the participating clients. The third model in the collection is used, and for simplicity's sake, the Adam optimizer with default parameters is employed.

parameters	
participating clients	4
local epochs	1
steps per epoch	3
batch size	10

TABLE 5.1: Parameters of the first experiment.

Using the aforementioned parameters, each client consumes 30 examples per GE. Considering that all four clients participate in each GE, 500 GEs are necessary to exhaust all training data.

The FL trained model is compared to a centrally trained one with same parameters. To do this properly, a common scale is required. As such, the number of times the training dataset is repeated is used.

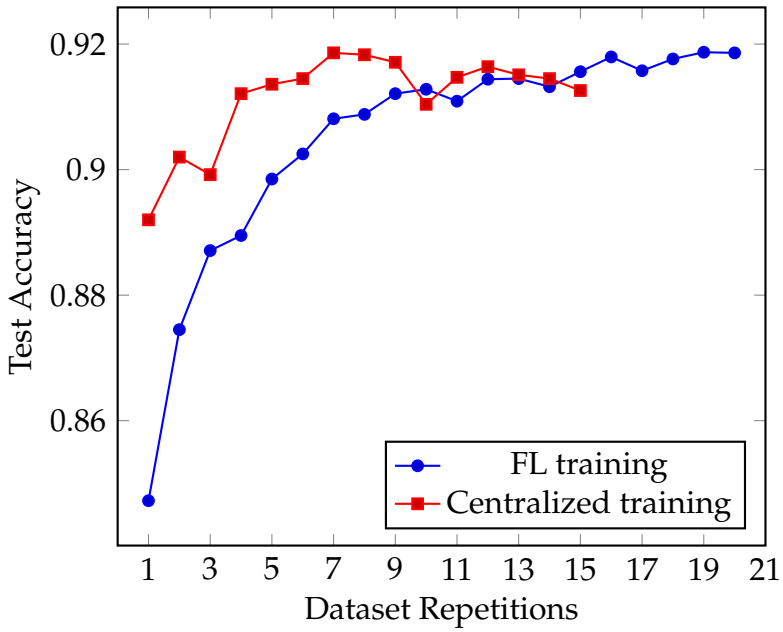

FIGURE 5.1: Experiment 1 results

The aforementioned findings demonstrate that training the model via FL yields the same accuracy as training it centrally, albeit at a slower rate. This is understandable given that the model in the first case is updated every 150 examples, whereas in the second case it is updated every 10 examples.

Another observation is the re-balancing effect of the FL algorithm. In centralized training, due to overfitting, the accuracy of the model degrades after peaking. This is not true when trained under the federated setting, as overfitted parameters are regularized when averaging multiple local models.

5.2 Distributed SGD with non-IID data

In this experiment, the third model is trained with distributed SGD and a pathological non-IID dataset. It is interesting to see how the batch size affects the performance of Distributed SGD, given that it is notorious for being

unable to handle non-IID datasets. Thus, the test was repeated with three distinct combinations of parameters.

case	1	2	3
participating clients	5	5	5
local epochs	1	1	1
steps per epoch	1	1	1
batch size	1	2	4

TABLE 5.2: Parameters of the second experiment.

The dataset is split between 5 clients, with each one getting all the examples of two labels. The first client holds all the examples with labels 0 or 1, the second client holds all the examples with labels 2 or 3 etc. As clients holds no knowledge on the other classes, self-training the model can only achieve a maximum accuracy of 20%. Therefore, it is required to either centralize the dataset or use a decentralized training method.

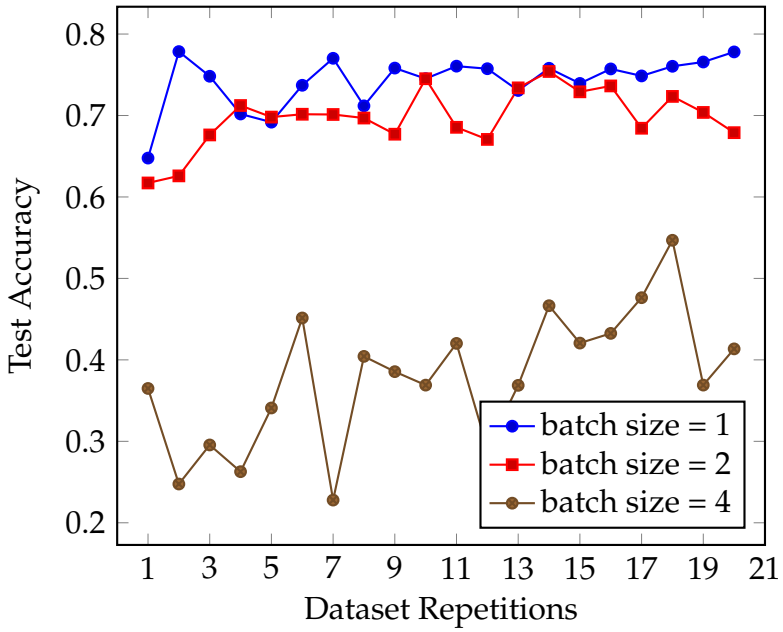

FIGURE 5.2: Experiment 2 results

Distributed SGD appears to struggle with non-IID data. With a batch size of just one example, it achieves accuracy consistent with prior works[4], but it is unable to converge with bigger batch sizes. This observation is consistent with FL theory, and in order to improve outcomes, additional techniques such as data rebalancing or expanding the client pool are needed.

5.3 Client Selection

In FL, it is frequently preferable to use a portion of the clients in each GE when there are several of them. In this method, data efficiency and model performance are improved since the global model can be updated more times before the training data run out. This experiment aims to test this functionality.

Eight clients are participating in training the Lenet-5 model. Every GE, only three clients are used. The dataset is split into 8 identically sized, mutually exclusive random shards, each of which is given to a client.

parameters	
total clients	8
clients per GE	3
local epochs	1
steps per epoch	2
batch size	20

TABLE 5.3: The parameters of the third experiment.

Data reshuffling is also incorporated in FL and centralized training. When all of the examples of a dataset have been used, it is resuffled and rebatched. Overfitting is thereby expected to diminish in both scenarios.

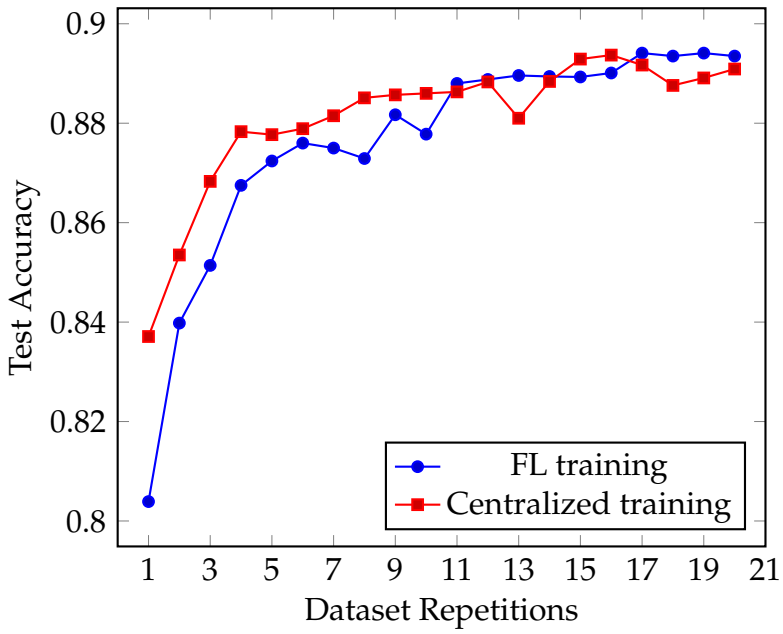

FIGURE 5.3: Experiment 3 results

In comparison to the first trial, where there was no client selection, FL training produces results that are comparable to those of centralized training more quickly. Furthermore, overfitting is decreased in both scenarios.

5.4 Greater data per GE consumption

The primary objective of this experiment is to assess the impact of increasing the consumption of local data per GE, prior migrating to the Federated Averaging algorithm. Furthermore, the FL environment is tested with a more complex architecture by using the ninth model that contains an inception module and a dropout layer.

The data is distributed randomly to 5 clients, but only 3 of them are used each GE. Two sets of parameters are used, with different number of local updates per GE.

parameters	FL set 1	FL set 2	Centralized training
total clients	5	5	1
clients per GE	3	3	1
steps per GE	1	2	examples/batch size
batch size	20	20	20
examples per GE	60	120	all
GEs to use all examples	1000	500	-

TABLE 5.4: Experiment 4 parameters

It is important to note that, compared to the first set of parameters, the second one needs only half as many communication rounds to exhaust the dataset.

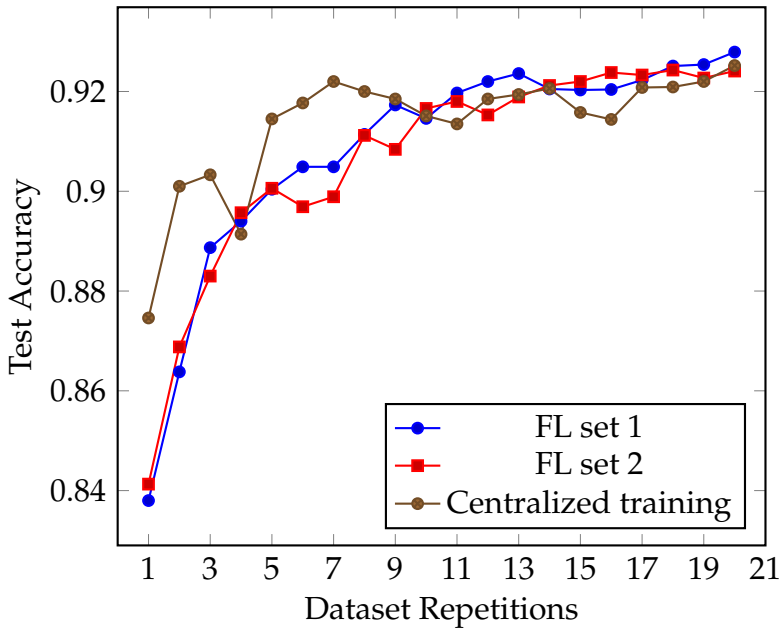

FIGURE 5.4: Experiment 4 results

Both FL scenarios reach comparable accuracy with centralized training. Although the second one appears to progress at a slower pace than the first, it only updates the global model half as often and needs half as much communication. This results in double the computation to communication ratio and being a more viable target for parallelization.

5.5 Client Fault Tolerance

In an edge environment, the clients may be unreliable and any algorithm must be resilient to random faults. This experiment aims to simulate such a case. To achieve this, 6 clients are initially participating in training the third model, but around 1/10 into training one of them abruptly disconnects. That means for 90% of the training, 1/6 of the data are inaccessible.

parameters	normal op	faulty op
total clients	5	6
clients per GE	3	3
steps per GE	1	1
batch size	20	20

TABLE 5.5: Experiment 5 parameters

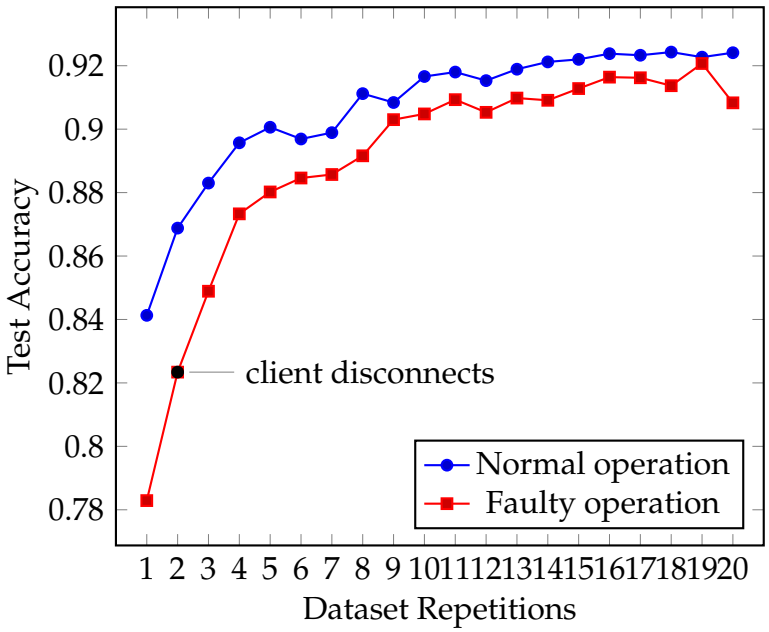

FIGURE 5.5: Experiment 5 results

Although the model's final accuracy drops, the effect is manageable as training continues and accuracy is still within acceptable bounds. In a real-world scenario, this issue can be resolved by postponing a portion of the training until after lost data resurfaces or new data becomes available.

5.6 Neural Network initialization

The initialization of an ANN can have a significant impact on the final accuracy, convergence rate, and training time, according to FL theory. It is generally accepted that the best course of action is to use the same initialization for all clients [4]. This major objective of this experiment is to assess this convention. To further emphasize the consequences of the initialization, the SGD optimizer with a low learning rate is employed.

parameters	FL seeded init	FL random init	centralized training
total clients	5	5	1
clients per GE	3	3	1
steps per GE	5	5	examples/batch
batch size	20	20	20
examples per GE	300	300	all
GEs to use all examples	200	200	-

TABLE 5.6: Experiment 6 parameters

The third model is used and initialized with the Glorot initializer. The model is trained twice, once initialized with the same seed across all clients, the other using random different seeds.

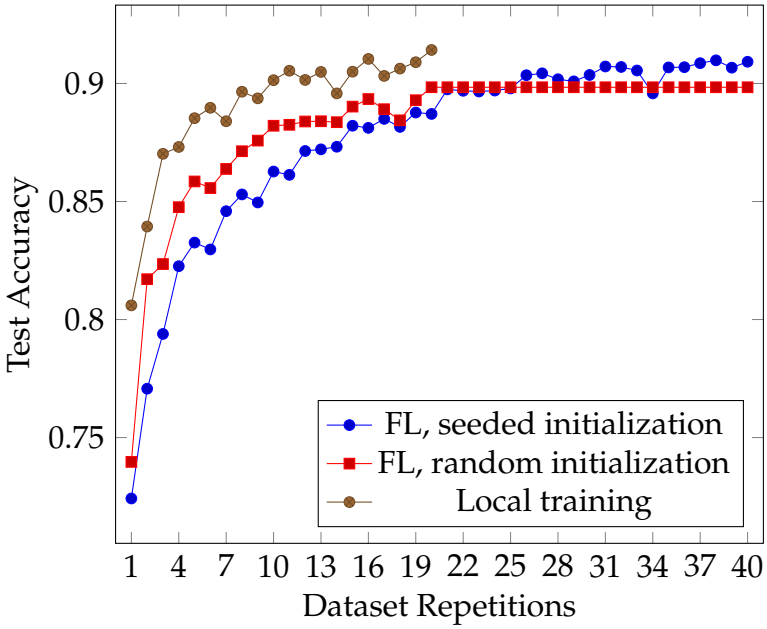

FIGURE 5.6: Experiment 6 results

The model with a random initialization quickly approaches and settles in a suboptimal local minimum. Both centralized training and FL with seeded initialization surpass its accuracy. This behaviour is consistent with FL theory.

5.7 Learning Rate (LR) decay strategies

Another aspect of FL worth investigating is learning rate (LR) decay strategies. The following three of them are implemented:

- Decay the LR every set number of GEs. All clients have the same LR at every moment.
- Decay the LR of a client based on the number of participated GEs. If a subset of the clients is used every GE, some clients may have been selected more times than others and as a result they will have a lower LR.
- The final strategy is to reduce a client's LR each time its dataset is repeated. This is an extension of the second strategy, where instead of

decaying slowly the LR every few rounds, there is a big drop every $\frac{\sum clients\ data}{\sum clients\ data\ used\ per\ GE}$ rounds of training.

parameters	FL, no decay	FL strategy 1	FL strategy 2	FL strategy 3
total clients	5	5	5	5
clients per GE	3	3	3	3
steps per GE	5	5	5	5
batch size	20	20	20	20
initial LR	1e-2	1e-2	1e-2	1e-2
LR decay	-	0.999	0.999	$\frac{0.999 * \sum clients\ data}{\sum clients\ data\ used\ per\ GE}$
decay interval (x = decay period)	-	x GEs	x participated GEs	$\frac{x\ participated\ GEs * \sum clients\ data}{\sum clients\ data\ used\ per\ GE}$

TABLE 5.7: Experiment 7 parameters

Each strategy is tested three times with different decay periods. The decay period dictates how often the decay applies. E.g. the second strategy with the a decay period of three means that LR decays every three participated rounds. A FL trained model without LR decay is used as a baseline.

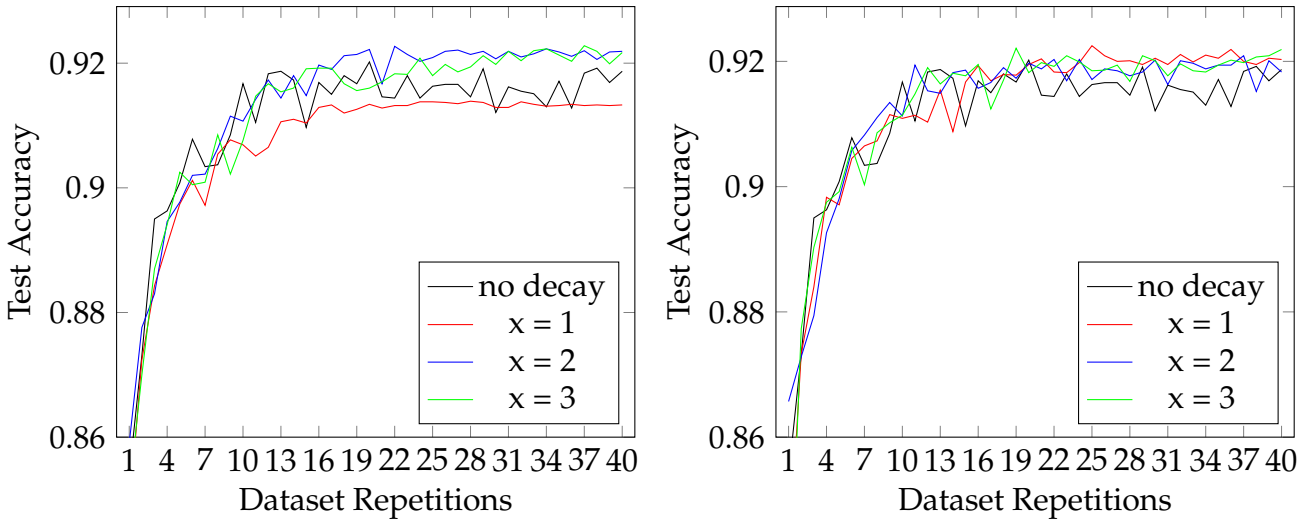

FIGURE 5.7: Experiment 7 results, strategies 1 and 2.

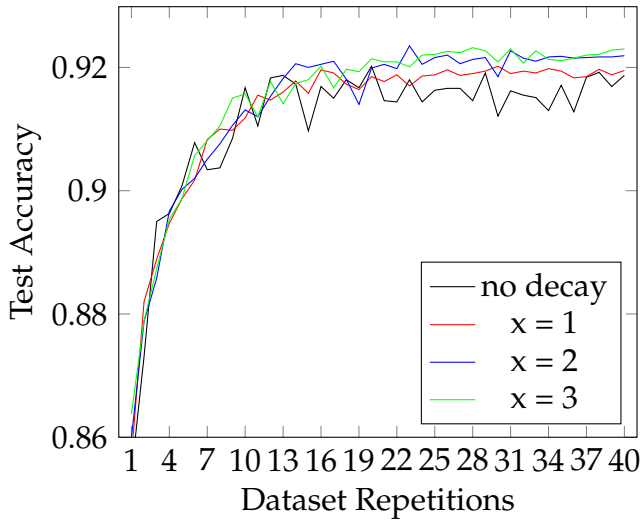

FIGURE 5.7: Experiment 7 results, strategy 3.

All strategies seems to to perform slightly better than the baseline, except the first one with decay period = 1. In that case, the decay is too fast and the LR degenerates in a state that cannot substantially alter the weights of the NN. The last strategy appears to be the most promising, which while outperforming the others is the most straightforward.

5.8 Federated Averaging (FedAvg)

In FedAvg, a client, when participating in a training round, uses all of its data and executes multiple SGD iterations. In the prior experiment, for a client to consume all of its data 200 GEs were necessary; whereas with FedAvg, only one GE (at most) is needed. The main objective of this experiment is to demonstrate the algorithm's compatibility with the developed FL environment.

parameters	FedAvg
total clients	5
clients per GE	3
local epochs	1
steps per epoch	600
batch size	20
initial LR	1e-2

TABLE 5.8: Experiment 8 parameters

The LR decay needs to be corrected to account for the reduced number of decay events, thus the model is trained multiple times to identify its ideal values. The prior experiment's LR decay is utilized for the first run of training, and each additional training reduces the descent slope by half. The model is also trained without LR decay.

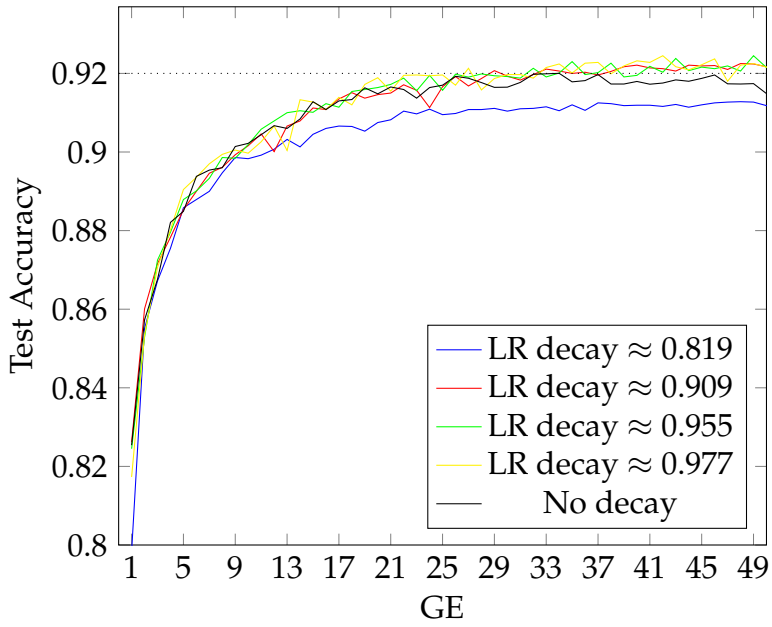

FIGURE 5.8: Experiment 8 results

The maximum accuracy of this model when trained locally is 92%. This is now regarded as the minimum baseline. In addition of showing maximum accuracy, the GE where that baseline was reached is also presented.

LR decay	Max accuracy	0.92 @GE
0.819	0.913	-
0.909	0.9232	30
0.955	0.9245	32
0.977	0.9245	27
No decay	0.9224	34

TABLE 5.9: Experiment 8 results

In comparison to the previous experiment, FedAvg requires $\times$100-200 less communication and the same computation to reach the target accuracy. However, there is a hidden cost in that less averaging occurs and the rebalancing effect is diminished. This becomes quite clear when training without LR decay, where overfitting is apparent.

Considering the different LR decay values, the more conservative options appear to perform best; decaying the LR too quickly causes the ANN to set in sub-optimal minima.

5.9 Client Participation and Increasing parallelism

This experiment explores the amount of multi-client parallelism that can be exploited and its effect on training. The dataset is split between 10 clients, each one holding 6000 training examples. The third model is trained with different number of participating clients per GE.

Test	A	B	C	D
total clients	10	10	10	10
clients per GE	1	3	5	10
local epochs	1	1	1	1
steps per epoch	300	300	300	300
batch size	20	20	20	20
initial LR	1e-2	1e-2	1e-2	1e-2
LR decay	0.977	0.977	0.977	0.977

TABLE 5.10: Experiment 9 parameters

Training with one client per epoch serves as the baseline. The relative reduction in communication is calculated for the other training runs.

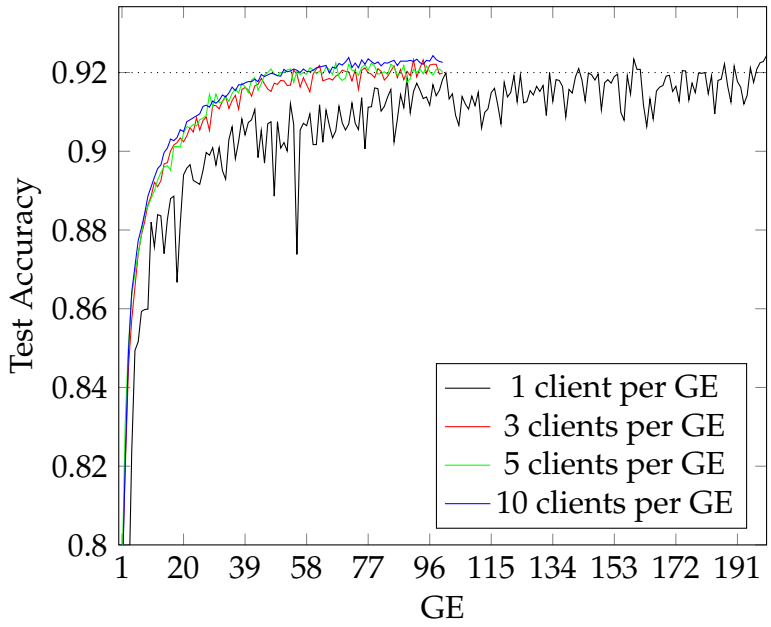

FIGURE 5.9: Experiment 9 results

client per GE	0.92 @GE
1	159
3	61 (2.6×)
5	46 (3.4×)
10	51 (3.1×)

TABLE 5.11: Experiment 9 results, with relative reduction in GEs.

Using more clients per GE substantially lowers the rounds of communication needed to achieve the target accuracy. This is consistent with others works, which show even on simulations of hundreds of clients with a small dataset each, using a bit more than half of the clients in each GE yields the best results. If all the clients are used every GE, especially when under non-IID data, the model may not converge in an acceptable solution.

5.10 Increasing computation per client

To further reduce communication, clients can perform more local updates per GE. This may be accomplished by increasing the number of local epochs (LE), reducing the batch size (B), or both. The third model is trained by 5 clients, with 3 of them participating each GE. LR and its decay are amortized

to maintain a consistent LR at each GE, regardless of the number of local updates. The goal of this experiment is to identify the behavior of the algorithm across different sets of parameters.

	Local epochs = 1		Local epochs = 3	
B	updates/GE	0.92 @GE	updates/GE	0.92 @GE
600	20	309	60	-
300	40	212	120	67
100	120	72	360	36
80	150	54	450	25
40	300	31	900	13
20	600	25	1800	7
10	1200	18	3600	15

TABLE 5.12: Experiment 10 results

According to the results of the experimental, increasing local updates directly decreases the required global updates. Unlike most works, this one concentrates on small groups of clients with large local datasets. As a result, increasing the number of local epochs produces inconsistent results due to the introduction of overfitting in the local models. Regarding the batch size, there is no cost in reducing it, providing that it is large enough to completely utilize the client's hardware parallelism.

Chapter 6

FPGA Design & Implementation

6.1 Tools Used

6.1.1 Vitis Unified Software Platform

The Vitis unified software platform[66] is a collection of tools, libraries and environments designed to ease the development of accelerated applications tailored for AMD Xilinx FPGA and Versal® ACAP hardware platforms. It includes graphical and command-line compilers, analyzers, and debuggers to build applications, analyze performance bottlenecks, and debug accelerated algorithms, developed in C, C++, or OpenCL APIs. Furthermore, it offers numerous advantages such as effortless application portability, complete simulation of hardware systems, and an open source runtime that handles host-device communication.

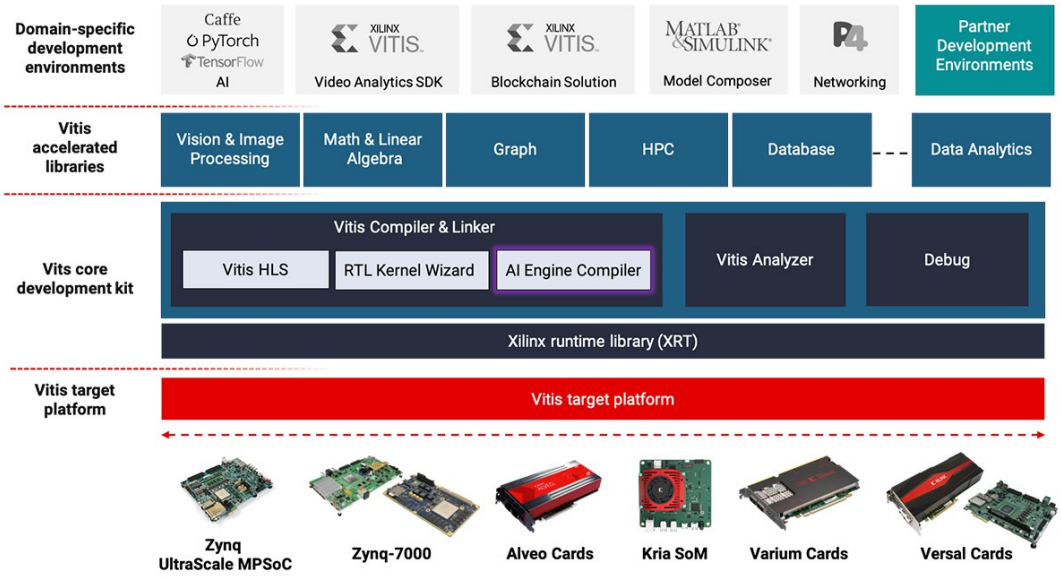

FIGURE 6.1: Vitis overview: URL.

Vitis supports hardware acceleration kernels controlled by PS or x86 kernels. The Vitis application acceleration development flow provides a framework for developing and delivering FPGA-accelerated applications using standard programming languages for both software and hardware components. The kernels can be developed through traditional RTL, C/C++ with Vitis HLS, the Vitis model composer and the AI Engine compiler.

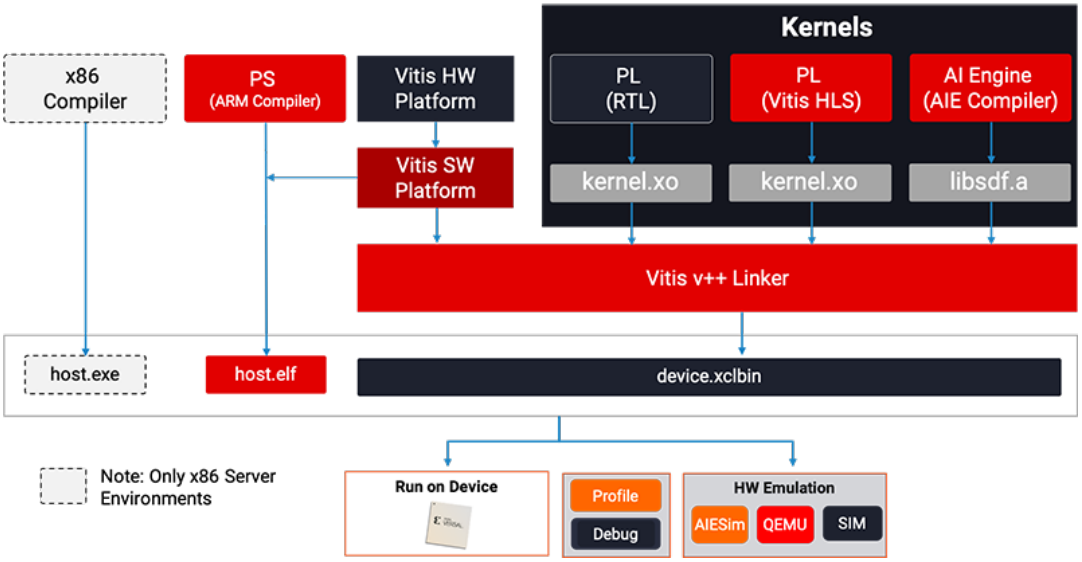

FIGURE 6.2: Vitis kernel architecture: URL.

6.1.2 Xilinx Runtime library (XRT)

The Xilinx Runtime library[67] (XRT) facilitates communication between the application code (running on an embedded Arm or x86 host) and the accelerators deployed on the reconfigurable portion of PCIe interface-based AMD Xilinx accelerator cards, MPSoC-based embedded platforms, or ACAPs. It is flexible with modifiable libraries and drivers, enabling different levels of abstractions, from high-level Python bindings to low-level C++ APIs. These APIs are common across all platforms and eliminate the need to implement hardware communication layers from scratch.

A widely used alternative are the OpenCL libraries. TBy abstracting the underlying implementations of numerous APIs, including the XRT, they offer a standard interface for managing heterogeneous devices. As a result, they enable portability across multiple devices from various providers, albeit with increased complexity due to the extra layer of abstraction. As this work is not indented to transition to other devices, the XRT is preferred.

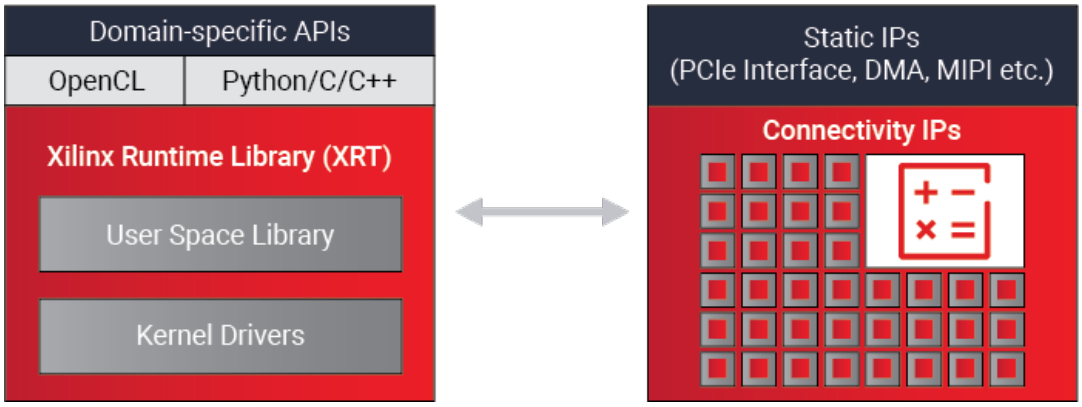

FIGURE 6.3: Xilinx Runtime Library overview: URL.

6.1.3 Vitis High Level Synthesis (HLS)

The Vitis HLS tool can synthesize a C/C++ function into RTL code for implementation in the programmable logic (PL) region of a Xilinx FPGA device. Its kernels can be easily integrated into a design utilizing OpenCL[68] code. It provides support of complex data types, math functions and AXI4-Stream interfaces for data exchange between IPs in the PL and/or Processing Subsystem (PS).

HLS is an automated design process that takes an abstract behavioral specification of a digital system and generates a register-transfer level structure that implements the given behavior. The designer is working on a high abstraction level, while the tool takes care of mechanical RTL implementation tasks.

Designer's Responsibilities
Macro Architecture
Design Intent
Constrains

HLS tool automation
FSM Generation
Operation Scheduling
Clock
Register Pipelining
Resource Sharing
Timing
Verification

TABLE 6.1: Distribution of work during HLS design.

6.2 FPGA Platforms

6.2.1 Xilinx Zynq UltraScale+ MPSoC

The Zynq® UltraScale+™ MPSoC is a family of Xilinx products that integrates a feature-rich 64-bit quad-core or dual-core Arm® Cortex®-A53 and dual-core Arm Cortex-R5F based processing system (PS) and Xilinx programmable logic (PL) UltraScale architecture in a single device. In addition, on-chip memory, multiport external memory interfaces, and a rich set of peripheral connectivity interfaces are included. [69]

6.2.2 ZCU102 Evaluation Board

The ZCU102 Evaluation Board features a Zynq® UltraScale+™ MPSoC with a quad-core Arm® Cortex®-A53, dual-core Cortex-R5F real-time processors, and a Mali™-400 MP2 graphics processing unit based on Xilinx's 16nm FinFET+ programmable logic fabric. It supports all major peripherals and interfaces, enabling development for a wide range of applications. Furthermore, its high speed DDR4 memory interfaces, variety of communication interfaces and FMC expansion ports makes it ideal for rapid prototyping.

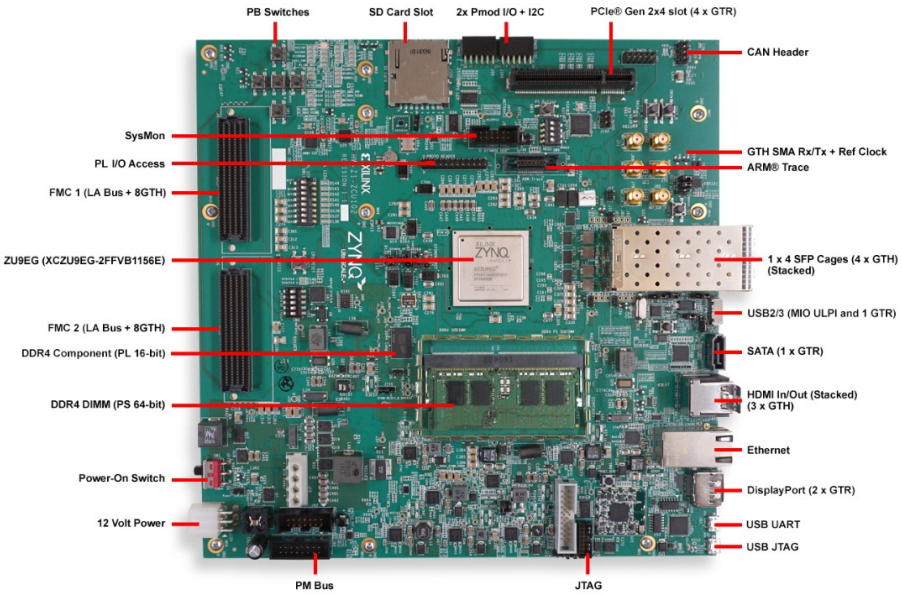

FIGURE 6.4: ZCU102 Features: URL.

Given that the thesis is based on an edge application, this platform seems to be an ideal fit for it. During the hardware design phase, the constrains and resource limitation were placed according to the specifications of this board. Nevertheless, transferring the final design to devices of similar families should require minimal effort.

6.3 Preparing the CNN for Hardware

In the preceding phase, TF with Python was utilized to implement all ANNs and training. As the Vitis Kernel Toolchain is aimed to C/C++ code, these implementations can not be synthesized to hardware by the aforementioned tools. As such, a simplified version of the most used CNN, the third in the model library, has been re-implemented with C++.

Migrating from TF to a hardware synthesizable CNN is a fairly challenging task riddled with pitfalls. This implementation is not optimized for hardware, but rather serves as a stepping stone between TF and synthesizable code. Certain practices are adopted to facilitate future transition to hardware targeted code:

- Implementation is modular and re-configurable. The code is build around template functions, each of which performs a specified task. Layers can effortlessly added, removed, or altered in size, shape and parameters.
- All data, whether input, output or internal, are produced and consumed serially and only once. This behavior is similar to the stream data format, which is widely utilized in hardware design.
- All feature maps, input gradients, variable gradients, and updated variables are logged and compared with those generated by the existing TF implementation. This approach not only evaluates functionality, but also produces test benches for future hardware implementation.

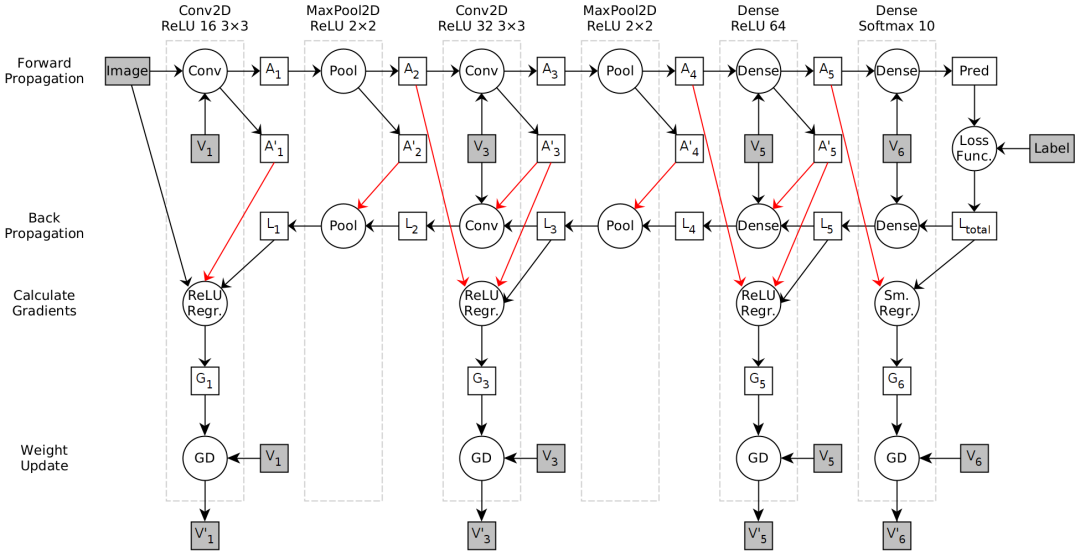

FIGURE 6.5: Dataflow diagram of the utilized CNN model.

Figure 6.5 depicts the basic structure of the implementation. Tasks are represented by cycles, whereas data are represented by squares. Inputs, labels, weights and any other data that must to be saved in memory have greyed-out squares. The majority of internal data are consumed immediately after being produced. Some of them, marked by red arrows, skip parts of the chain and must be temporally stored.

6.4 Vitis HLS Hardware Implementation

Even with the aforementioned techniques, adapting the code to be compatible with FPGAs is not a trivial task. To build an efficient implementation, resource usage, data access patterns, and other factors must be taken into account. All parts of the CNN are modified accordingly.

6.4.1 2D Convolutional Layers

Due to their non-serial data access patterns, multi-dimensional filter algorithms frequently conflict with FPGA design; 2-D convolution is no exception. At its core, it carries out some form of data averaging around a pixel, necessitating the access of nearby input values as seen in figure 6.6. Additionally, when calculating the adjacent outputs, some inputs are accessed again.

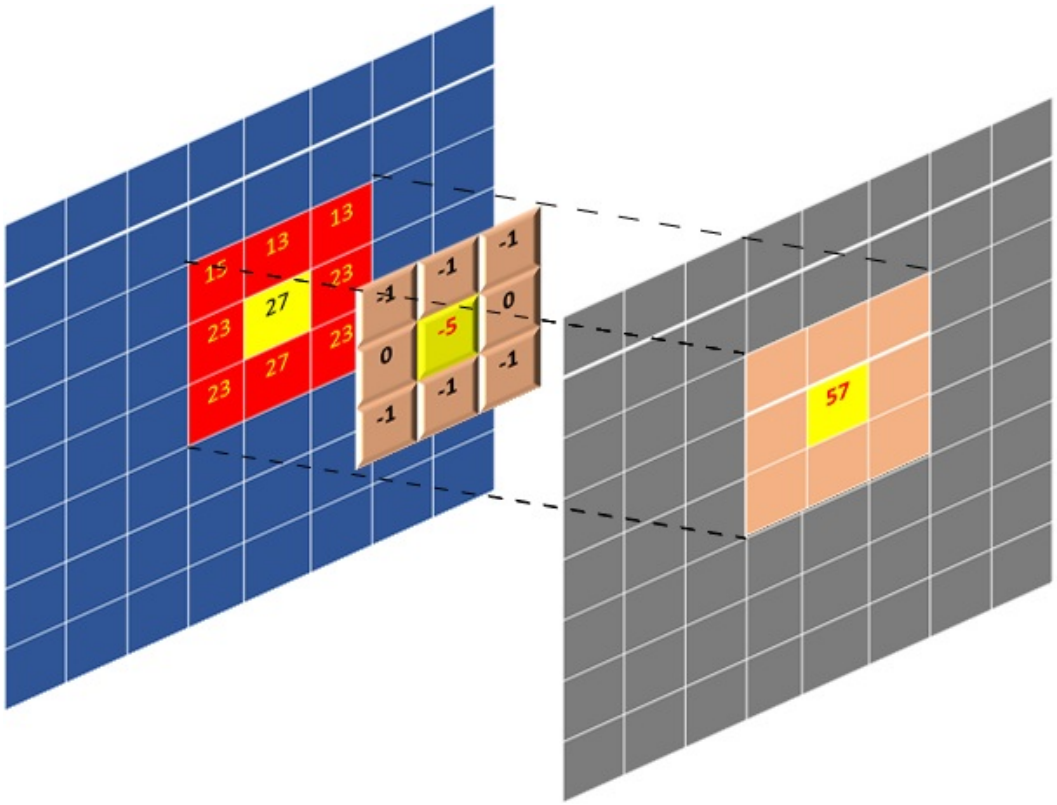

FIGURE 6.6: Convolution access pattern: Input (Blue) pixels are accessed in a non-serial pattern.URL

In a CPU-focused implementation this would be a non issue, as data caching and pre-fetching can ensure that the majority of accesses will be cache hits. Implementing this on an FPGA would produce numerous small non-burst accesses on the global memory, resulting in unacceptable performance. Thus, a different approach is required.

A unique data mover, specifically designed for the given algorithm, has been developed to reduce the number of global memory accesses. Its key concept is to construct two-dimensional input windows that are the same size as the filters and then compute the dot product of those. Its main components are buffers that store lines of the input, and a sliding window on top of them.

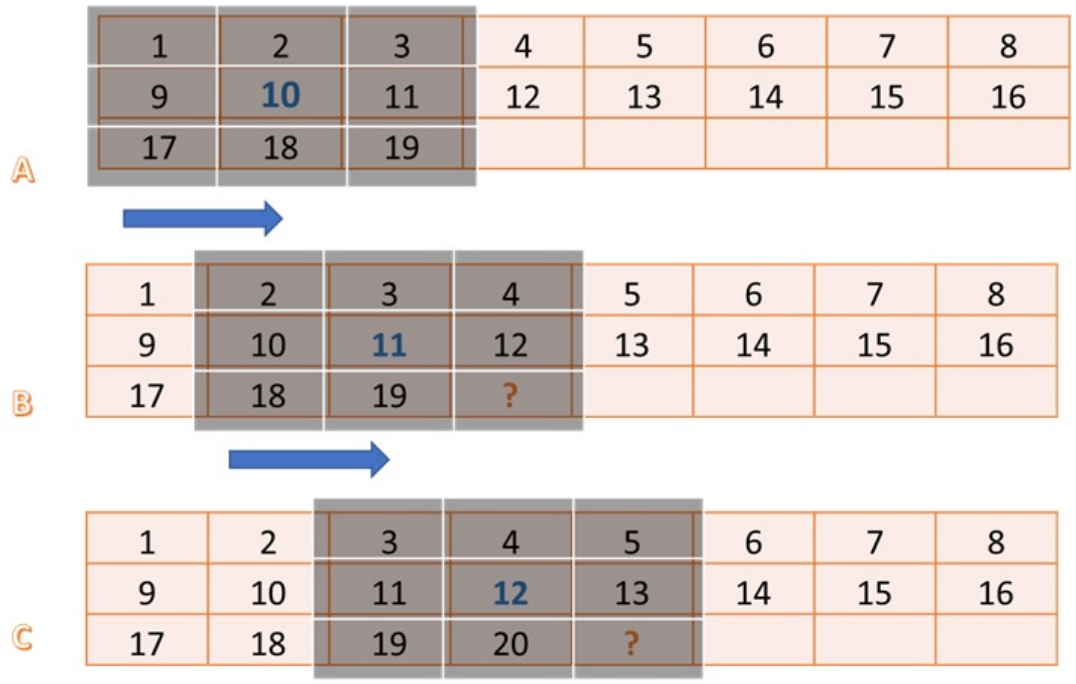

FIGURE 6.7: Line Buffers: Sifting a 3x3 window. URL

Figure 6.7 illustrates the operation of the line and window buffering scheme. A continuous stream of 3x3 windows is produced by sifting a window buffer over the top of the line buffers. Since the masked elements of the top line are already present in the window, only two line buffers are needed. Furthermore, only one new input pixel is required to produce a window, and thus an output pixel. Finally, zero padding is applied to maintain correct data with edge windows.

To complete the 2D convolution, a processing element is required. In the simplest scenario, a single channel input, the dot products between the windows and the filters are calculated and activated with the ReLU function. If there are additional input channels, the dot products are calculated in respect of each channel, which are aggregated and then activated to produce the feature map of the layer. This is done to allow computing of multiple channels in parallel, while using a data streaming paradigm.

Two output streams are produced, one float and one bool. The first one consists of the activations and is connected to the next layer. The second one indicates whether or not the kernels have activated the ReLU function. As only activated neurons convey their error backwards, this is necessary information for the back-propagation.

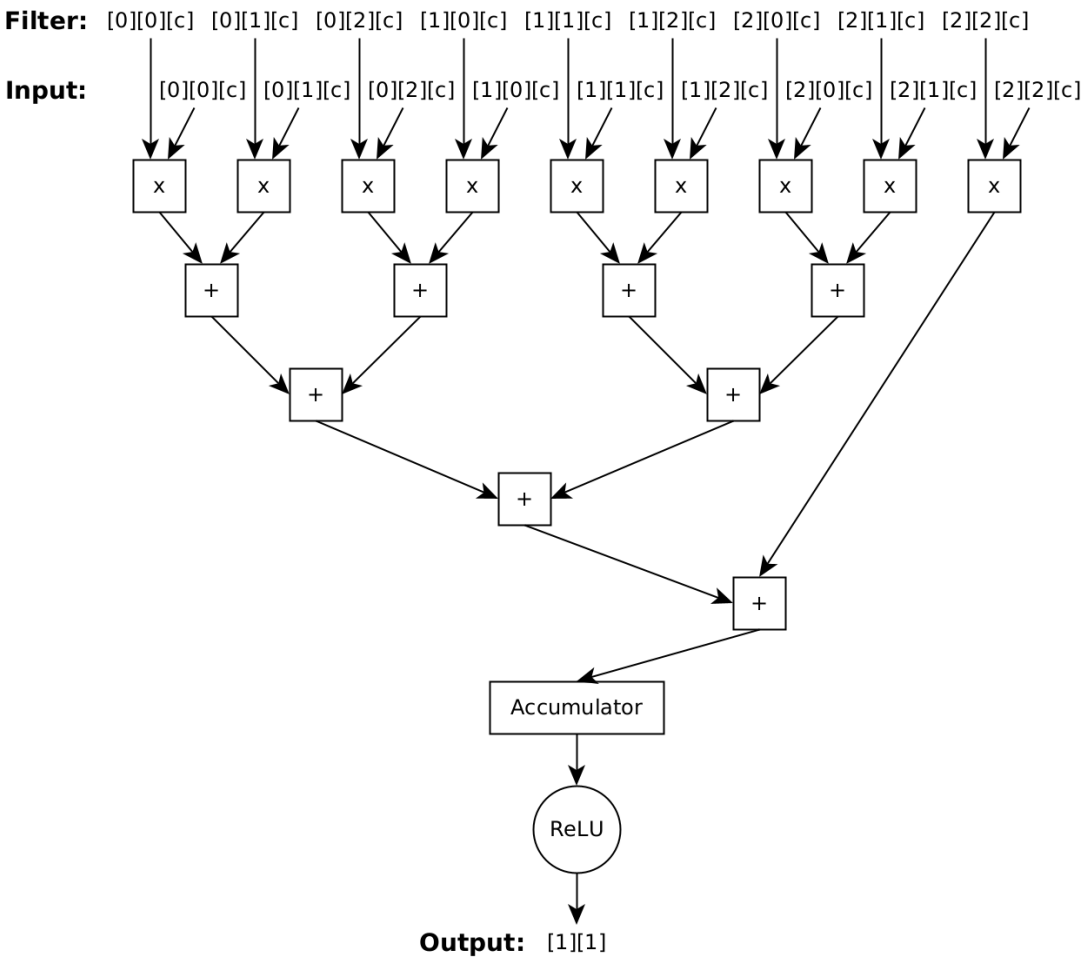

FIGURE 6.8: Order of calculations when computing the pixel [1][1] of a single filter. Iterates through all channels of the input (c).

Figures 6.8 and 6.9 illustrate the same operation, from a different point of view. The first focus on the flow of the algorithm, while the second focus on the structure of the hardware functions. The additions and multiplications tree corresponds to the dot product function, the accumulator represent the sum channels function, and the ReLU is the activation function. In figure 6.8, the input is already windowed and padded.

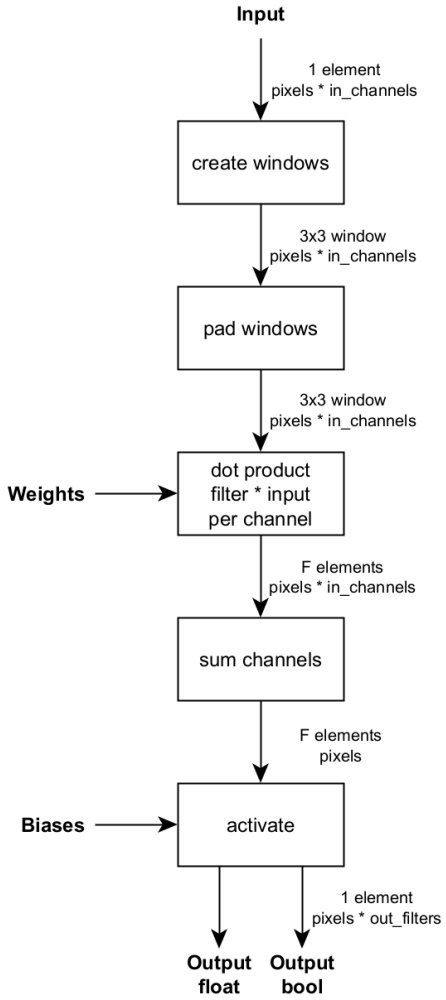

FIGURE 6.9: Block diagram of the 2D convolution forward propagation. F is the number of filters. The data types of the internal streams with the total data passed are shown.

The transformation from software to HLS hardware implementation is shown in the following pseudocode:

Algorithm 3 2D Convolution: Software implementation.

SW Implementation:

for p in pixels **do**

for o in output channels **do**

for i in input channels **do**

$out[p][o] \mathrel{+}= filter[o][i] * in[p][i]$

$out[p][o] = activate(out[p][o])$

$out[p][o]$, $filter[o][i]$ & $in[p][i]$ have the dimensions of the filter. They can be multiple dimension arrays.

Algorithm 4 2D Convolution: Software to HLS Hardware Transformation.

Step 1: Transpose output channels dimension from time to space.

for p in pixels **do**
  for i in input channels **do**
    $out[p][0] \mathrel{+}= filter[0][i] * in[p][i]$
    $out[p][1] \mathrel{+}= filter[1][i] * in[p][i]$
    ... ▷ output channel times
  $out[p][0] = activate(out[p][0])$
  $out[p][1] = activate(out[p][1])$
  ... ▷ output channel times

Step 2: Split multiplications, additions & activations in distinct functions.

Func A:

for p in pixels **do**
  for i in input channels **do**
    $A[0] = filter[0][i] * in[p][i]$
    $A[1] = filter[1][i] * in[p][i]$
    ... ▷ output channel times
    Write $(A[0], A[1], \cdots)$ to $streamA$

Func B:

for p in pixels **do**
  for i in input channels **do**
    Read $(A[0], A[1], \cdots)$ from $streamA$
    $B[0] \mathrel{+}= A[0]$
    $B[1] \mathrel{+}= A[1]$
    ... ▷ output channel times
  Write $(B[0], B[1], \cdots)$ to $streamB$

Func C:

for p in pixels **do**
  Read $(B[0], B[1], \cdots)$ from $streamB$
  $out[p][0] = activate(B[0])$
  $out[p][1] = activate(B[1])$
  ... ▷ output channel times

Algorithm 5 2D Convolution: HLS implementation.

Step 3: Make output channels dimension flexible between time & space using HLS tools.

Func A:
  for p in pixels **do**
    for i in input channels **do**
      #pragma HLS PIPELINE II=flexible
      for o in output channels **do** ▷ If II=1 loop is flattened
        $A[o] = filter[o][i] * in[p][i]$
      Write $(A[0], A[1], \cdots)$ to $streamA$

Func B:
  for p in pixels **do**
    for i in input channels **do**
      #pragma HLS PIPELINE II=flexible
      Read $(A[0], A[1], \cdots)$ from $streamA$
      for o in output channels **do** ▷ If II=1 loop is flattened
        $B[o] \mathrel{+}= A[o]$
      Write $(B[0], B[1], \cdots)$ to $streamB$

Func C:
  for p in pixels **do**
    Read $(B[0], B[1], \cdots)$ from $streamB$
    for o in output channels **do**
      $out[p][o] = activate(B[o])$

The overall scheme is designed to maximize the data reuse providing maximum parallel data to the processing element, with minimum use of memory. Back propagation and gradient calculation follow the same logic with a few minor differences:

In back propagation, the input is the activated output gradients. To activate them, the system need to remember which neurons fired during forward propagation, as shown in figure 6.5. Furthermore, the biases are not used, and the channel/filter dimensions are reversed. Finally the output are the gradients of the layer's input.

The processing element of the gradient calculation differs more. Instead of using the weights to calculate the outputs, the outputs are used to calculate

the weight gradients. Furthermore, finding the bias gradients is trivial, as they are equal with the sum of all output gradients of their filter.

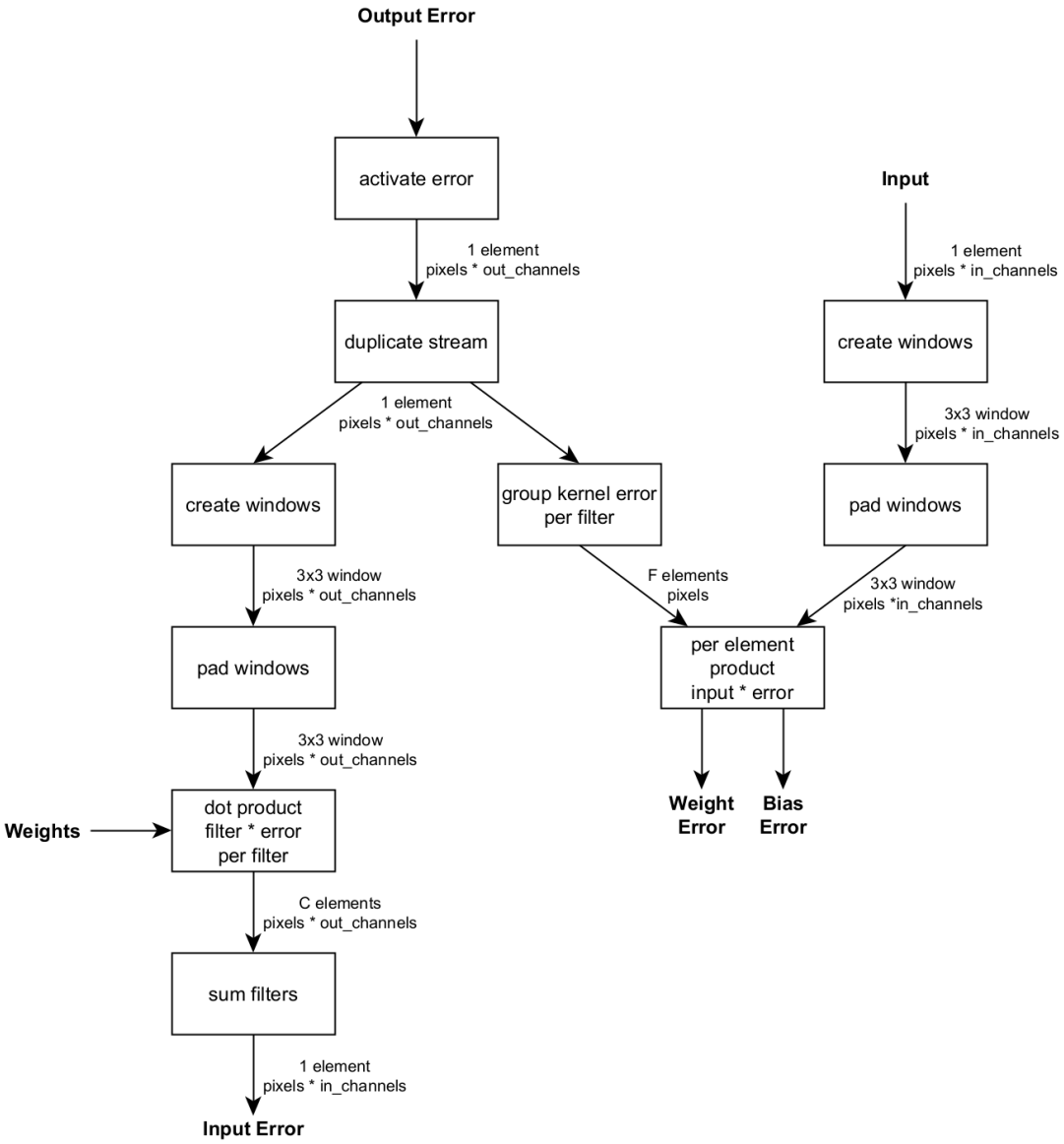

FIGURE 6.10: Block diagram of the 2D convolution back propagation. F is the number of filters, C is the number of input channels. The data types of the internal streams with the total data passed are shown.

6.4.2 2D Max-Pooling Layers

The 2D Max-Pooling layers are implemented using the same logic as the 2D convolutional layers, albeit with a few major differences. First of all the window is 2x2 is size, and with a stride of 2. As a result, each window contains exclusive data, and an output can only be obtained with four inputs.

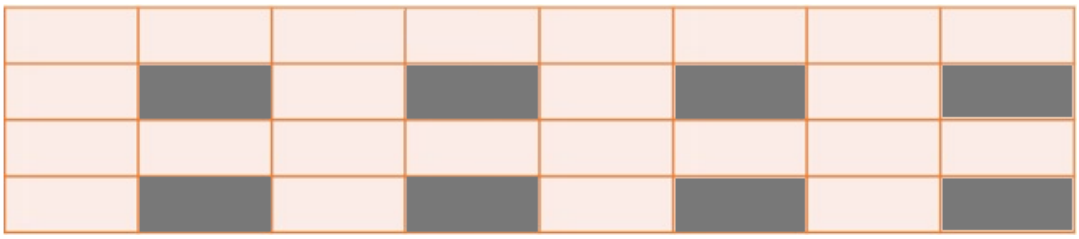

FIGURE 6.11: Line Buffers, Max-Pool: Grey pixels represent which inputs will trigger an output generation.

Figure 6.11 demonstrates the Max-Pool layer's uneven output generation, an unavoidable issue of any algorithm with stride greater than one. Following hardware does not operate while there are no data available, which is a problem in a FPGA design, as idle hardware indicates wasted hardware space. By raising the Iteration Interval (II) of the following hardware functions and properly calibrating the size of internal FIFO streams, the constant operation of the entire system is ensured.

The processing component of forward propagation is quite simple, as the output is the highest value in each window. It is important to note that the output's spatial dimensions are two times smaller than those of the input. Even more straightforward is back-propagation, in which the error back-propagates towards the maximum of each window. All other connections are assigned zero error gradients.

6.4.3 Dense & Softmax Layers

The implementations of the dense and Softmax layers are simple and fairly similar. They are made up of two components: matrix multiplication of their inputs and weights and their respective activation function. In back-propagation and gradient calculation, the output error is activated before used as the input, with the input and variable gradients being the outputs.

The most crucial aspect of their design is ensuring that the hardware functions are constantly operating. To accomplish this, a streaming architecture, that reads and writes inputs and outputs serially and only once, is used. Important to note is that the Softmax activation requires all the inputs to be received before calculating any output, meaning that for an example back-propagation can not start until forward propagation is fully completed.

6.4.4 Gradients Calculation Pipeline

A major advantage of FPGA accelerators is that multiple hardware functions operate simultaneously, if the implemented algorithm allows. This holds true for most of the design. As an example, The first maxpool layer requires four inputs to generate the first output. These inputs have being generated by the first convolutional layer before a training data-point is fully loaded and processed. Thus the first two layers can operate simultaneously.

On the other hand, the Softmax layer, which is the last step of the forward propagation, operates like a barrier. Due to the nature of the algorithm, to produce its feature map, all inputs must first be collected. As a result, for a single training data-point, the forward propagation must be completed before the back propagation begins.

As such, the sequential semantics must be preserved, and the pipeline is implemented with a dataflow region that follows the control-driven task-level parallelism paradigm. This means that a subsequent function can start before the previous finishes and multiple functions can start and operate simultaneously. All tasks and channels are instantiated and connected explicitly. Furthermore, the inputs and outputs of the tasks are of stream type or stable memory arrays.

In this paradigm, the task with the highest latency typically determines the overall latency. Due to the existence of the Softmax barrier, for a single data input, forward propagation tasks can not operate simultaneously with tasks after it. As a result, the minimum overall latency equals the highest task latency before the barrier plus the highest task latency after it.

Figure 6.12 shows in detail the developed dataflow region that generates the weight and bias gradients. The heavy use of auxiliary data transformation functions, such as create windows and stream, is evident. These functions consume almost no hardware when synthesized, and add near zero latency.

Furthermore, several data streams skip hardware functions and layers. This could introduce stalls and ultimately deadlocks. To address this issue, hardware functions are implemented as free running pipelines (FRPs), when possible. Such implementations significantly reduce the possibility of a stall, by continuing to operate even when no input data are available or the output streams are full.

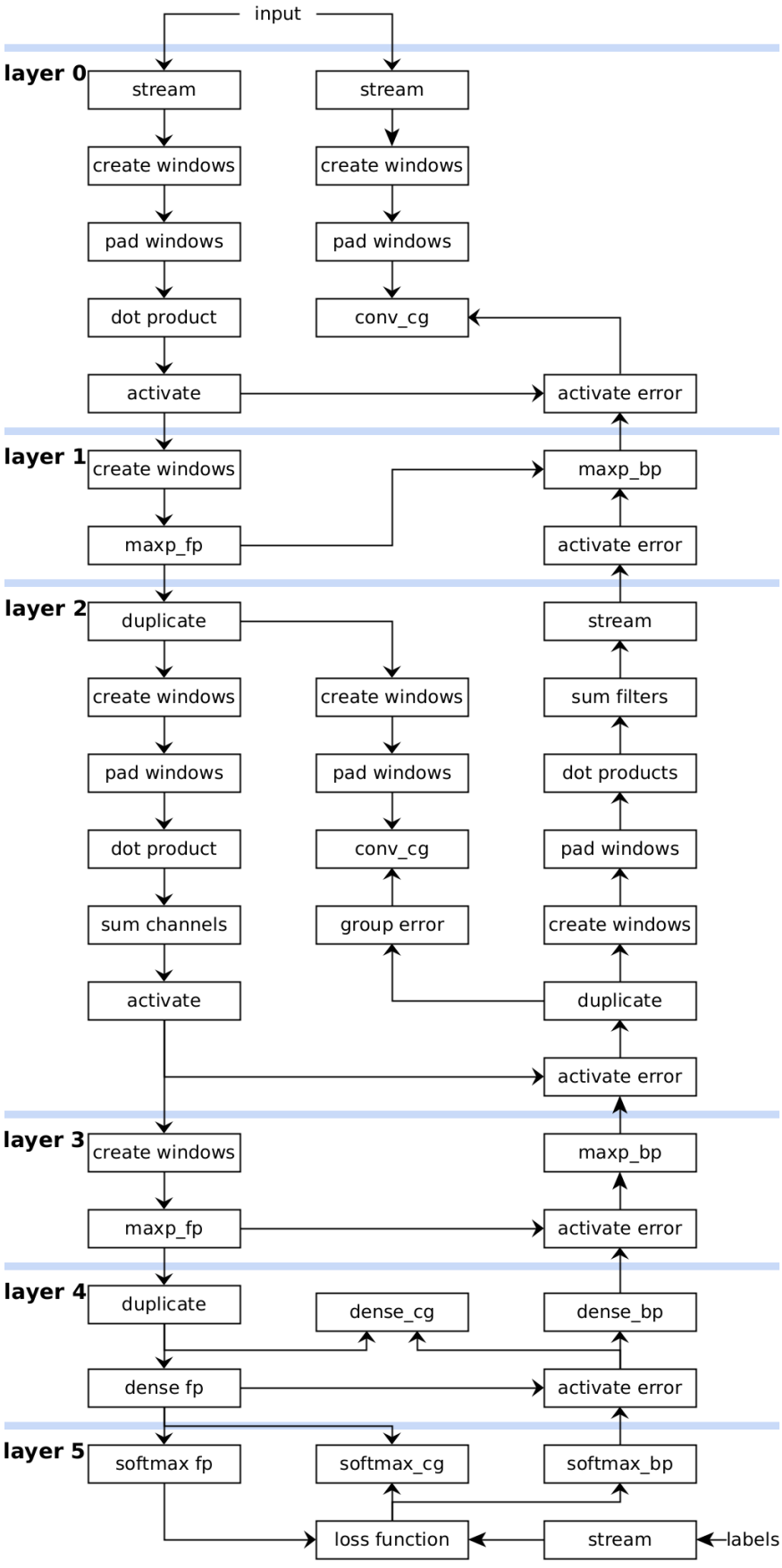

FIGURE 6.12: Block diagram of the pipeline that generates the gradients. Inputs/Outpouts are not shown. Arrows represent data streams.

Using FRPs comes with multiple restrictions, a strict coding style is required, and MAXI ports are not supported. For hardware functions that read or write data from MAXI ports or can not adhere to other restrictions, flushable pipelines (FLPs) are used instead. They achieve the same goal as FRPs, but by instantiating multiple copies of the pipeline and executing them independently. As a result, the design is robust against any unpredictable stalls that MAXI ports may introduce.

6.4.5 Hardware Streams

All communication between the hardware functions is facilitated with the stream implementation provided by the Vitis HLS library "hls_stream.h". Hardware functions stall when an output stream is full, making the overall architecture inefficient. It is crucial to prevent this by determining the proper depth of the streams.

In most cases, this is trivial as they link sequential functions in a dataflow region where the consumer can instantly begin utilizing any data written by the producer. Then the major factor of the depth is the II of the connected functions. For most stream, a depth of two is sufficient.

More consideration is required about the streams that skip parts of the function chain. Due to the Softmax barrier, forward propagation of a sample is completed before its back propagation begins, thus all their data are produced before any of them is consumed. As such, their minimum depth is equal with the data produced by a single input sample.

To determine the ideal depths, an iterative optimization approach has typically been used. The Vitis HLS environment offers a variety of simulation tools that generate a number of useful statistics, such as the amount of clock cycles that each function stalls and whether or not a stream becomes full. Monitoring these when testing, enables calibrating the depths to ensure the stable flow of the pipeline, while not wasting hardware space in unnecessarily large streams.

6.4.6 Batching Inputs

As already explained, not all hardware functions can run concurrently for a single input sample. This issue is mitigated through batching input samples, where while an input runs through back-propagation, the next one is used in forward propagation.

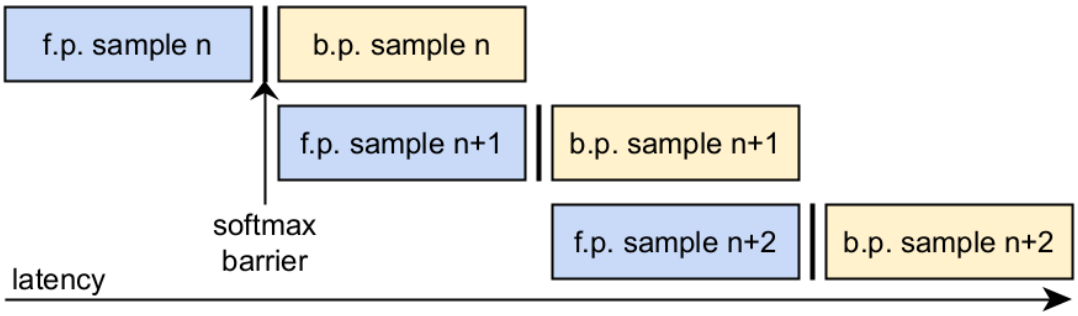

FIGURE 6.13: Latency of the pipeline under batching.

With both sub-regions of the the pipeline having the same latency, the expected overall latency for a dataset without batching is defined as:

$$L_{dataset} = \sum^{samples} (L_{fp} + L_{bp}) = 2 \cdot samples \cdot L_{fp/bp} \tag{6.1}$$

With batching enabled, the latency of the dataset is transformed as:

$$\begin{gathered} L_{dataset} = \sum^{\substack{batches \\ per\ dataset}} \sum^{\substack{samples \\ per\ batch}} (L_{fp/bp} + 1) = \\ \frac{samples}{size_{batch}} \cdot size_{batch} \cdot (L_{fp/bp} + 1) = \\ samples \cdot (L_{fp/bp} + 1) \end{gathered} \tag{6.2}$$

With Vitis HLS, implementing batching on each individual hardware function is quite trivial. Encapsulating their C++ definitions in a loop of the same size as the desired batch, is sufficient.

Further though must be given to the size of the streams connecting functions of the forward propagation with functions after the barrier. The producer functions will block until the consumer functions clear some space in the stream if the minimal depth is used as mentioned in the preceding section. Depending on the minimal latency between the producer and the consumer, this can be avoided by increasing the depth by 0.5 to 1 times.

6.4.7 Updating Variables

Based on the produced gradients, the variables of the ANN are updated in a second dataflow region. Due to the independence of all weights and gradients, this process is relatively straightforward. The classic SGD with momentum optimization algorithm is used and the learning rate is supplied by the driver program. Thus, latency and hardware usage are the only criteria for the applied parallelism in this specific region.

6.4.8 Data Movement & Storage

Under the Vitis flow, AXI streams are unavailable for the ZCU102, thus memory mapping is used to transfer data from general memory to the PL and vice versa. These data are the input samples and labels, as well as the variables of the ANN. Appropriate data mover functions have been developed.

In Vitis HLS, arrays are implemented as continuous memory spaces with one or two ports, and only a limited amount of data can be read or written per cycle. To increase data accesses per cycle, the arrays are partitioned with the appropriate HLS directive *ARRAY_PARTITION*. The HLS tool splits the initial arrays to smaller ones, whose size and shape depend on the parameters of the corresponding directives.

The weights of the ANN are accessed in multiple functions of the first dataflow region and require special treatment. These arrays must be designated as shared using the directive *STREAM* with the type parameter set to *SHARED* in order for the design to be syntesizable. The tool then recognizes there are numerous consumers and multiplies the ports accordingly, without duplicating the array data.

For the weights of the second convolutional layer, this is insufficient. They are accessed by two functions with different access patterns. To resolve this, both access patterns could be satisfied by increasing the partition factor and dimensions. This solution generate a huge amount of access ports, increasing hardware consumption unacceptably. More appropriate solution is creating two arrays with unique partitioning each. Albeit more memory is needed, the overall hardware usage is lower.

To generate the gradients, the ANN's inputs, data and labels, are sent from global memory to the PL via AXI Master Adapter ports. In a dataflow region, each channel must have a single consumer, thus two ports are needed

to propagate the input data to the two hardware functions that consume it. All ports are independent from one another by having their own dedicated bundles, thus enabling the simultaneous reading of all inputs.

The gradients are produced in the first dataflow region and consumed in the second, thus persistent saving in on-chip memory is necessary. The related memory is not shared since their producers and consumers are in different regions, and can be implemented as streams or arrays, whichever is more convenient. The same applies for the momenta of the gradients, as they may be produced and consumed in same dataflow region, but in different iterations of it.

6.4.9 Top function

To hold everything together, a top level function has been developed, whose signature operates as the API between PL and PS. Furthermore, it contains all the definitions of the memory structures, as well as the instantiations of the data movers and dataflow regions. Finally, to train on multiple batches, the dataflow regions are enclosed in a loop.

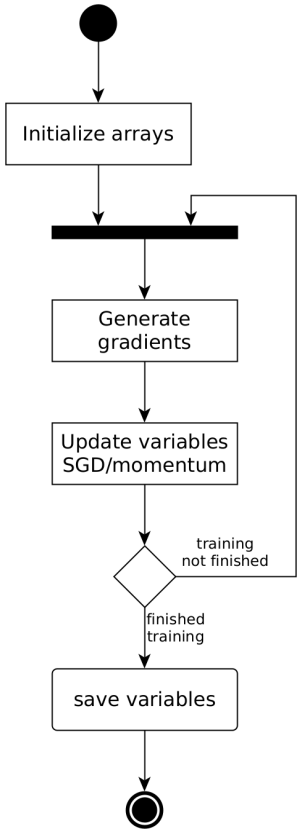

FIGURE 6.14: Block diagram of the accelerator's top function.

6.5 Host Program

6.5.1 Driver Architecture

The overall application uses Linux system calls, such as socket read and write. Thus a bare metal implementation is inadequate and a host program in the PS is required to drive the developed hardware design in the PL. This is achieved with the use of the XRT native C++ APIs [70].

The flow of the driver code is as follows:

- Open the device.
- Load the compiled and linked binary (XCLBIN) onto the device.
- Open the kernel loaded to the device with the XCLBIN.
- Create buffer objects to transfer data to and from the inputs and outputs of the kernel.
- Write data to the input buffers.
- Execute the kernel.
- Read data from the outputs buffers.

6.5.2 Memory Management

Since the target device is a SoC, both host and kernel programs can read and write in the same memory space. By facilitating data transfers between PL and PS through the map API provided by the XRT library, no data movements are necessary other than filling the global buffers. The training dataset is stored in binary files and is directly transferred in global memory; no intermediately buffers are used in the user space of the host program.

While the same methodology could be used when moving the ANN's weights and biases, the codebase would get too inflexible and brittle. They are a part of the byte-stream that is received from the socket connected to the FL server; copying it immediately in the global memory may introduce issues regarding endianess and alignment. Thus, an intermediary buffer is utilized to contain the incoming byte-stream, and the relevant sections of it are copied to the global memory using the write API of the XRT library.

Diagram 6.12 demonstrates how the image inputs are needed twice. They are stored twice in global memory, in two distinct memory banks, to prevent

any bus or AXI conflicts. As a result, both hardware functions can read them concurrently without experiencing any additional latency.

6.5.3 Incorporating the driver in the FL client

Uniting the Federated Learning and the driver codebases is trivial. First off, the server is, by design, agnostic to the implementation of the training. As it does not interface with any hardware kernel, there is no need for any code changes when compiled for another device, such as the ZCU102.

In the case of the client, all that is required is swapping the calls to the integrated Python interpreter with those to the aforementioned driver. The following diagram shows the architecture of the FL client on the ZCU102, with an emphasis on data transfers.

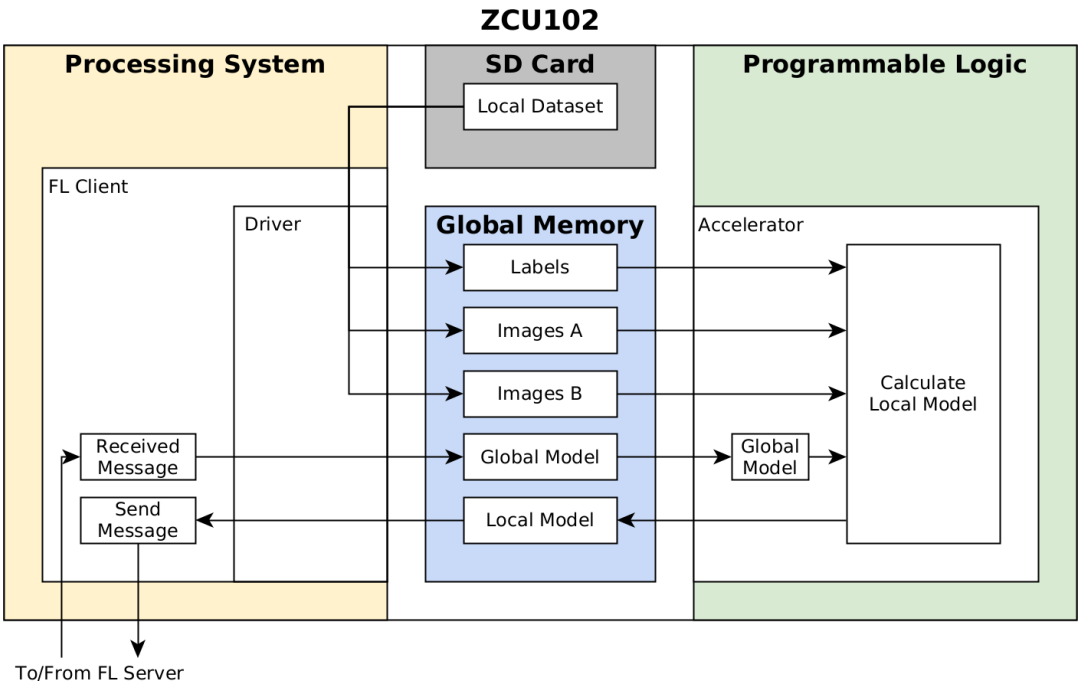

FIGURE 6.15: Architecture of the FL client on the ZCU102.

Chapter 7

Results

7.1 Specification of Compared Platforms

7.2 Power Consumption

7.3 Energy Consumption

7.4 Throughput and Latency Speedup

7.5 Final Performance

Chapter 8

Conclusions and Future Work

8.1 Conclusions

8.2 Future Work

Bibliography

[1] Yun Chao Hu et al. *Mobile Edge Computing A key technology towards 5G*. Tech. rep. 11. 06921 Sophia Antipolis CEDEX, France: European Telecommunications Standards Institute, Sept. 2015. URL: https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf.

[2] European Parliament and Council of the European Union. *REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)*. May 2016. URL: http://data.europa.eu/eli/reg/2016/679/oj (visited on 05/25/2022).

[3] Chau A., Hertzberg S., and Dodd S. *The California Consumer Privacy Act of 2018*. June 2018. URL: https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375 (visited on 05/25/2022).

[4] H. Brendan McMahan et al. "Communication-Efficient Learning of Deep Networks from Decentralized Data". In: (2016). DOI: 10.48550/ARXIV.1602.05629. URL: https://arxiv.org/abs/1602.05629.

[5] Jed Mills, Jia Hu, and Geyong Min. "Communication-Efficient Federated Learning for Wireless Edge Intelligence in IoT". In: *IEEE Internet of Things Journal* 7.7 (Aug. 2020), pp. 5986–5994. DOI: 10.1109/jiot.2019.2956615. URL: https://doi.org/10.1109/jiot.2019.2956615.

[6] Kang Wei et al. "Federated Learning With Differential Privacy: Algorithms and Performance Analysis". In: *IEEE Transactions on Information Forensics and Security* 15 (2020), pp. 3454–3469. DOI: 10.1109/tifs.2020.2988575. URL: https://doi.org/10.1109/tifs.2020.2988575.

[7] Stuart J. Russell and Peter Norvig. "Introduction". In: *Artificial Intelligence: A modern approach*. 2nd ed. Pearson Education, Inc., 2003, pp. 31–32. ISBN: 0137903952; 9780137903955; 0130803022; 9780130803023.

[8] Stuart J. Russell and Peter Norvig. "Learning from Observations". In: *Artificial Intelligence: A modern approach*. 2nd ed. Pearson Education,

Inc., 2003, pp. 649–651. ISBN: 0137903952; 9780137903955; 0130803022; 9780130803023.

[9] Mohamed Elgendy. "Feature extraction". In: *Deep learning for vision systems*. 1st ed. New York, NY: Manning Publications, Dec. 2020, p. 27. ISBN: 1617296198; 9781617296192.

[10] Fuzhen Zhuang et al. *A Comprehensive Survey on Transfer Learning*. 2019. DOI: 10.48550/ARXIV.1911.02685. URL: https://arxiv.org/abs/1911.02685.

[11] Dan Geiger and David Heckerman. *Advances in Probabilistic Reasoning*. 2013. DOI: 10.48550/ARXIV.1303.5718. URL: https://arxiv.org/abs/1303.5718.

[12] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* 5.4 (Dec. 1943), pp. 115–133. DOI: 10.1007/bf02478259. URL: https://doi.org/10.1007/bf02478259.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[14] Yu Zhang et al. *Pushing the Limits of Semi-Supervised Learning for Automatic Speech Recognition*. 2020. DOI: 10.48550/ARXIV.2010.10504. URL: https://arxiv.org/abs/2010.10504.

[15] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. *A Survey of the Usages of Deep Learning in Natural Language Processing*. 2018. DOI: 10.48550/ARXIV.1807.10854. URL: https://arxiv.org/abs/1807.10854.

[16] Jonathan A. Weyn, Dale R. Durran, and Rich Caruana. "Can Machines Learn to Predict Weather? Using Deep Learning to Predict Gridded 500-hPa Geopotential Height From Historical Weather Data". In: *Journal of Advances in Modeling Earth Systems* 11.8 (Aug. 2019), pp. 2680–2693. DOI: 10.1029/2019ms001705. URL: https://doi.org/10.1029/2019ms001705.

[17] Jason Riordon et al. "Deep Learning with Microfluidics for Biotechnology". In: *Trends in Biotechnology* 37.3 (Mar. 2019), pp. 310–324. DOI: 10.1016/j.tibtech.2018.08.005. URL: https://doi.org/10.1016/j.tibtech.2018.08.005.

[18] Ritika Wason. "Deep learning: Evolution and expansion". In: *Cognitive Systems Research* 52 (Dec. 2018), pp. 701–708. DOI: 10.1016/j.cogsys.2018.08.023. URL: https://doi.org/10.1016/j.cogsys.2018.08.023.

[19] *History of Neural Networks*. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/index.html (visited on 06/01/2022).

[20] Jason Brownlee. *How to Choose an Activation Function for Deep Learning*. Jan. 2021. URL: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/ (visited on 06/04/2022).

[21] IBM Cloud Education. *Neural Networks*. Aug. 2020. URL: https://www.ibm.com/cloud/learn/neural-networks (visited on 06/04/2022).

[22] *Convolutional Neural Networks for Visual Recognition*. 2022. URL: https://cs231n.github.io/convolutional-networks/ (visited on 06/05/2022).

[23] James D. McCaffrey. *Neural Network Glorot Initialization*. June 2017. URL: https://jamesmccaffrey.wordpress.com/2017/06/21/neural-network-glorot-initialization/ (visited on 06/11/2022).

[24] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (May 13–15, 2010). Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[25] James D. McCaffrey. *Neural Network Glorot Initialization*. Visual Studio Magazine. May 2019. URL: https://visualstudiomagazine.com/articles/2019/09/05/neural-network-glorot.aspx (visited on 06/11/2022).

[26] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *CoRR* abs/1502.01852 (2015). arXiv: 1502.01852. URL: http://arxiv.org/abs/1502.01852.

[27] Andrew Jones. *An Explanation of Xavier Initialization*. Feb. 2015. URL: https://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization (visited on 06/11/2022).

[28] *Loss functions*. Peltarion. URL: https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions (visited on 06/11/2022).

[29] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[30] *Backpropagation*. Wikipedia. Mar. 2022. URL: https://en.wikipedia.org/wiki/Backpropagation (visited on 06/13/2022).

[31] *Gradient*. Wikipedia. May 2022. URL: https://en.wikipedia.org/wiki/Gradient (visited on 06/13/2022).

[32] *Gradient Descent*. IBM Cloud Education. Oct. 2020. URL: https://www.ibm.com/cloud/learn/gradient-descent (visited on 06/14/2022).

[33] *Gradient descent*. Wikipedia. June 2022. URL: https://en.wikipedia.org/wiki/Gradient_descent (visited on 06/14/2022).

[34] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, Jan. 2013, pp. 1139–1147. URL: https://proceedings.mlr.press/v28/sutskever13.html.

[35] *Overfitting*. IBM Cloud Education. Mar. 2021. URL: https://www.ibm.com/cloud/learn/overfitting (visited on 06/17/2022).

[36] Wei Yang Bryan Lim et al. "Federated Learning in Mobile Edge Networks: A Comprehensive Survey". In: *IEEE Communications Surveys & Tutorials* 22.3 (2020), pp. 2031–2063. DOI: 10.1109/COMST.2020.2986024.

[37] Peter Kairouz et al. *Advances and Open Problems in Federated Learning*. 2019. DOI: 10.48550/ARXIV.1912.04977. URL: https://arxiv.org/abs/1912.04977.

[38] Li Li et al. "A review of applications in federated learning". In: *Computers & Industrial Engineering* 149 (2020), p. 106854. ISSN: 0360-8352. DOI: https://doi.org/10.1016/j.cie.2020.106854. URL: https://www.sciencedirect.com/science/article/pii/S0360835220305532.

[39] Andrew Hard et al. *Federated Learning for Mobile Keyboard Prediction*. 2018. DOI: 10.48550/ARXIV.1811.03604. URL: https://arxiv.org/abs/1811.03604.

[40] Qiang Yang et al. "Federated Machine Learning: Concept and Applications". In: (2019). DOI: 10.48550/ARXIV.1902.04885. URL: https://arxiv.org/abs/1902.04885.

[41] Tian Li et al. "Federated Learning: Challenges, Methods, and Future Directions". In: *IEEE Signal Processing Magazine* 37.3 (May 2020), pp. 50–60. DOI: 10.1109/msp.2020.2975749. URL: https://doi.org/10.1109/msp.2020.2975749.

[42] Michael Sprague et al. "Asynchronous Federated Learning for Geospatial Applications". In: Mar. 2019, pp. 21–28. ISBN: 978-981-10-0665-4. DOI: 10.1007/978-3-030-14880-5_2.

[43] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2015. DOI: 10.48550/ARXIV.1510.00149. URL: https://arxiv.org/abs/1510.00149.

[44] Nikko Strom. "Scalable distributed DNN training using commodity GPU cloud computing". In: *INTERSPEECH*. 2015.

[45] Nikhil Joshi. *Model Skewing Attacks on Machine Learning Models*. Payatu. Feb. 2021. URL: https://payatu.com/blog/nikhilj/sec4ml-machine-learning-model-skewing-data-poisoning (visited on 06/27/2022).

[46] Elie Bursztein. *Attacks against machine learning — an overview*. May 2018. URL: https://elie.net/blog/ai/attacks-against-machine-learning-an-overview/#:~:text=impacted%5C%20your%5C%20users.-,Feedback%5C%20weaponization,this%5C%20fact%5C%20to%5C%20their%5C%20advantage. (visited on 06/27/2022).

[47] Christian Szegedy et al. *Intriguing properties of neural networks*. 2013. DOI: 10.48550/ARXIV.1312.6199. URL: https://arxiv.org/abs/1312.6199.

[48] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. *Deep Models Under the GAN: Information Leakage from Collaborative Deep Learning*. 2017. DOI: 10.48550/ARXIV.1702.07464. URL: https://arxiv.org/abs/1702.07464.

[49] *What is Secure Multiparty Computation?* Inpher. URL: https://inpher.io/technology/what-is-secure-multiparty-computation/ (visited on 07/01/2022).

[50] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. 2017. DOI: 10.48550/ARXIV.1708.07747. arXiv: 1708.07747. URL: http://arxiv.org/abs/1708.07747.

[51] Yann LeCun and Corinna Cortes. *MNIST handwritten digit database*. 2010. URL: http://yann.lecun.com/exdb/mnist/.

[52] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[53] Christian Szegedy et al. *Going Deeper with Convolutions*. 2014. DOI: 10.48550/ARXIV.1409.4842. URL: https://arxiv.org/abs/1409.4842.

[54] Daniel Povey, Xiaohui Zhang, and Sanjeev Khudanpur. *Parallel training of DNNs with Natural Gradient and Parameter Averaging*. 2014. DOI: 10.48550/ARXIV.1410.7455. URL: https://arxiv.org/abs/1410.7455.

[55] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[56] *TensorFlow Federated: Machine Learning on Decentralized Data*. URL: https://www.tensorflow.org/federated (visited on 07/08/2022).

[57] *Python/C API Reference Manual*. Python Software Foundation. July 2022. URL: https://docs.python.org/3/c-api/index.html (visited on 07/11/2022).

[58] *Embedding Python in Another Application*. Python Software Foundation. July 2022. URL: https://docs.python.org/3/extending/embedding.html (visited on 07/11/2022).

[59] *TensorFlow in other languages*. Google Inc. Aug. 2019. URL: https://github.com/tensorflow/docs/blob/master/site/en/r1/guide/extend/bindings.md (visited on 07/11/2022).

[60] *socket(2) — Linux manual page*. man7.org. Mar. 2021. URL: https://man7.org/linux/man-pages/man2/socket.2.html (visited on 07/08/2022).

[61] *Berkeley Software Distribution*. Wikipedia. May 2022. URL: https://en.wikipedia.org/wiki/Berkeley_Software_Distribution (visited on 07/08/2022).

[62] Li Blanca and Lu Peter. *Normalize Data component*. Microsoft. Apr. 2021. URL: https://docs.microsoft.com/en-us/azure/machine-learning/component-reference/normalize-data (visited on 07/09/2022).

[63] *TensorFlow Datasets, A collection of ready-to-use datasets*. TensorFlow. (Visited on 07/09/2022).

[64] Richard Durstenfeld. "Algorithm 235: Random Permutation". In: *Commun. ACM* 7.7 (July 1964), p. 420. ISSN: 0001-0782. DOI: 10.1145/364520.364540. URL: https://doi.org/10.1145/364520.364540.

[65] *Fisher–Yates shuffle wiki*. Wikipedia. Apr. 2022. URL: https://en.wikipedia.org/wiki/Fisher%5C%E2%5C%80%5C%93Yates_shuffle (visited on 08/08/2022).

[66] *Vitis unified software platform*. AMD Xilinx. 2022. URL: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html#overview (visited on 10/20/2022).

[67] *Xilinx Runtime Library*. AMD Xilinx. 2022. URL: https://www.xilinx.com/products/design-tools/vitis/xrt.html (visited on 10/20/2022).

[68] *OpenCL overview*. The Khronos® Group Inc. 2022. URL: https://www.khronos.org/opencl/ (visited on 11/14/2022).

[69] *Zynq UltraScale+ MPSoC Data Sheet: Overview*. Xilinx Inc. May 2021. URL: https://docs.xilinx.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview (visited on 11/14/2022).

[70] *XRT Native APIs*. Xilinx Inc. Oct. 2022. URL: https://xilinx.github.io/XRT/master/html/xrt_native_apis.html (visited on 02/13/2023).