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
Associate Prof. Michail G. LAGOUDAKIS
Dr. Vassilis PAPAEFSTATHIOU
(FORTH-ICS)

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

June 21, 2022

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 επεκτίνεται 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

Abstract iii

Abstract v

Acknowledgements vii

Contents ix

List of Figures xiii

List of Tables xv

List of Algorithms xvii

List of Abbreviations xix

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Scientific Contributions . . . 2
- 1.3 Thesis Outline . . . 2

2 Theoretical Background 5
- 2.1 Artificial Intelligence & Machine Learning . . . 5
  - 2.1.1 Information management . . . 5
  - 2.1.2 Feedback mechanism . . . 6
  - 2.1.3 Representation of the learned information . . . 7
- 2.2 Deep learning . . . 7
  - 2.2.1 Artificial Neuron . . . 9
  - 2.2.2 Activation Functions . . . 10
    - Binary Step . . . 10
    - Sigmoid . . . 11
    - ReLU . . . 11
    - Softmax . . . 11

2.2.3 Artificial Neural Network architectures 11
Deep Neural Network (DNN) 12
Convolutional Neural Network (CNN) 12
rnn? 14
2.3 Training Artificial Neural Networks 14
2.3.1 Initialization 15
2.3.2 Loss Functions 16
Regression Loss Functions 16
Classification Loss Functions 17
Reward Functions 18
2.3.3 Backpropagation 18
2.3.4 Gradient Descent 20
Challenges 21
Variations 22
2.3.5 Model Overfitting 23
2.3.6 Adaptive Learning Rate? 25
2.3.7 Adaptive algoritms? 25
2.4 Federated Learning 25
2.4.1 Typical Federated Training Process 26
Task Initialization 26
Local Training 26
Model Aggregation 27
2.4.2 Federated Learning Settings 27
2.4.3 Categorization? 29
Architectures for a federated learning system 29
2.4.4 Core challenges or Unique characteristics and issues of FL 29
Expensive communication 29
Systems heterogeneity 29
Statistical heterogeneity 29
Privacy concerns 30
2.4.5 Communication cost 30
Edge and End Computation 30
Model Compression 30
Importance-based Updating 30
2.4.6 Data distribution 30
Non-identical client distributions 31
2.4.7 communication security 31

3 Related Work **33**
3.1 Training Datasets . . . 33
3.2 ANN architectures . . . 33
3.3 parallelsgd . . . 33
3.4 FedAvg . . . 33
3.5 CE-FedAvg . . . 33
3.6 Evolution of FL . . . 33
3.7 quantization? . . . 34
3.8 The FPGA Perspective . . . 34
3.9 Thesis Approach . . . 34

4 Robustness Analysis **35**
4.1 Experiment A . . . 35
4.2 Experiment B . . . 35

5 FPGA Implementation **37**
5.1 Tools Used . . . 37
5.1.1 Vivado IDE . . . 37
5.1.2 Vivado High Level Synthesis (HLS) . . . 37
5.1.3 Vivado SDK . . . 37
5.2 FPGA Platforms . . . 37

6 Results **39**
6.1 Specification of Compared Platforms . . . 39
6.2 Power Consumption . . . 39
6.3 Energy Consumption . . . 39
6.4 Throughput and Latency Speedup . . . 39
6.5 Final Performance . . . 39

7 Conclusions and Future Work **41**
7.1 Conclusions . . . 41
7.2 Future Work . . . 41

References **43**

List of Figures

2.1 Edge detection in greyscale images 6
2.2 AI Venn Diagram 8
2.3 McCulloch-Pitts Neuron 9
2.4 Deep neural network 12
2.5 A CNN sequence to classify handwritten digits 13
2.6 2D convolution 13
2.7 2D max pooling 14
2.8 Effect of learning rate in Gradient Descent 20
2.9 Local minimum and saddle point 21
2.10 Model overfitting 23
2.11 Overfitting/Underfitting 24
2.12 FL topology 26

List of Tables

2.1 FL scenarios in comparison with data center distributed learning. 28

List of Algorithms

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ANN	**A**rtificial **N**eural **N**etwork
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
FL	**F**ederated **L**earning
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GDPR	**G**eneral **D**ata **P**rotection **R**egulation
MEC	**M**ulti-access **E**dge **C**omputing
ML	**M**achine **L**earning
MSE	**M**ean **S**quare **E**rror
RNN	**R**ecurrent **N**eural **N**etwork

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 from 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.

Beside being incompatible, these two technologies may complement each other, which is something 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 implementation is modular and platform independent.

1.2 Scientific Contributions

The main focus of this thesis is combining FL training with FPGA implementations of ANN, while exploring and overcoming their inherent conflicts.
Furthermore, it focus on the mostly unexplored FL setting of small client pools and its inherent difficulties.
Finally, it gives a real world implementation of FL that can be used as a benchmark for future works. It provides an FL implementation that is agnostic to 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 - Robustness Analysis:** Chapter 4 description
- **Chapter 5 - FPGA Implementation:** Chapter 5 description
- **Chapter 6 - Results:** Chapter 6 description
- **Chapter 7 - Conclusions and Related Work:** Chapter 7 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.

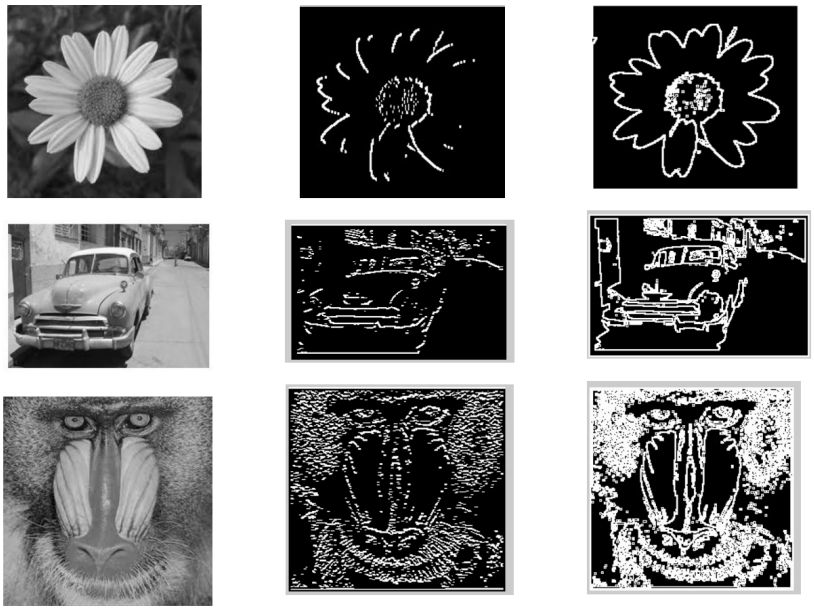

FIGURE 2.1: Edge detection in greyscale images: URL.

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.

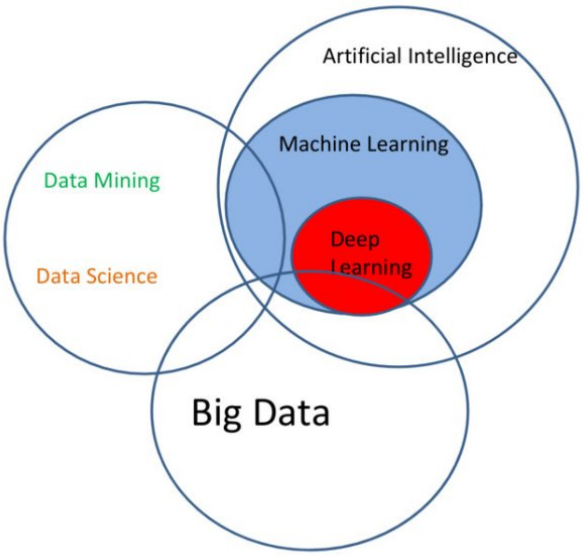

FIGURE 2.2: AI, ML, DL, Data Mining, Data Science, and Big Data: URL.

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.

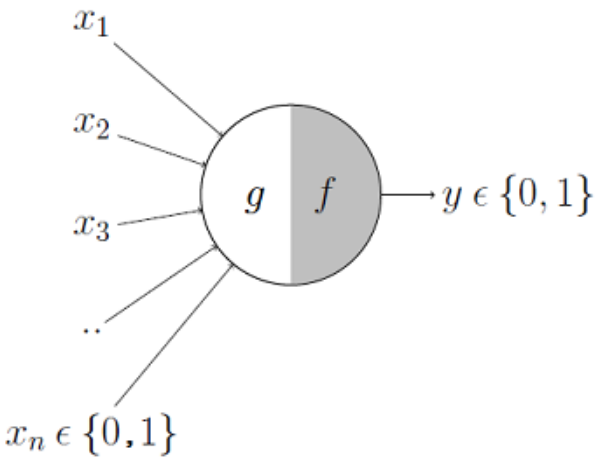

FIGURE 2.3: The McCulloch-Pitts Neuron: URL.

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]

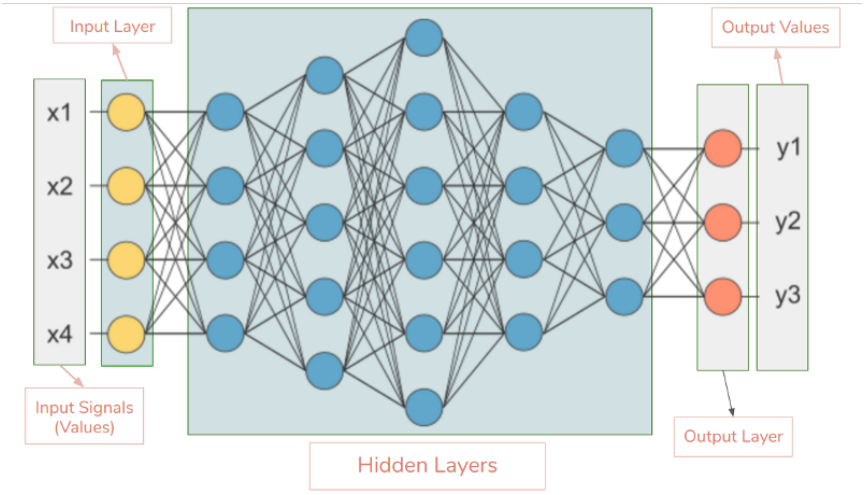

FIGURE 2.4: DNN with 5 hidden layers: URL.

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.

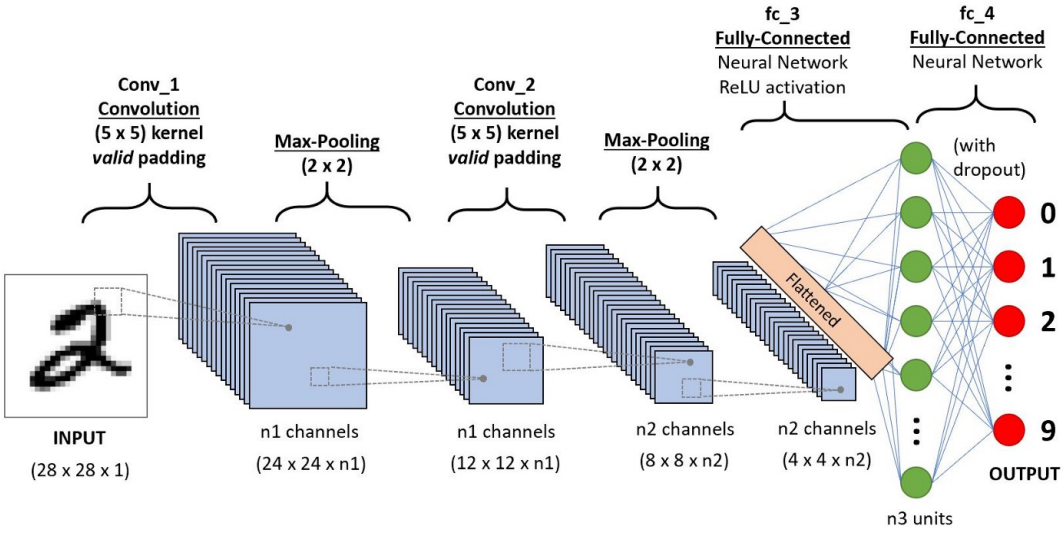

FIGURE 2.5: A CNN sequence that classifies handwritten digits using 2 convolutional layers: URL.

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.

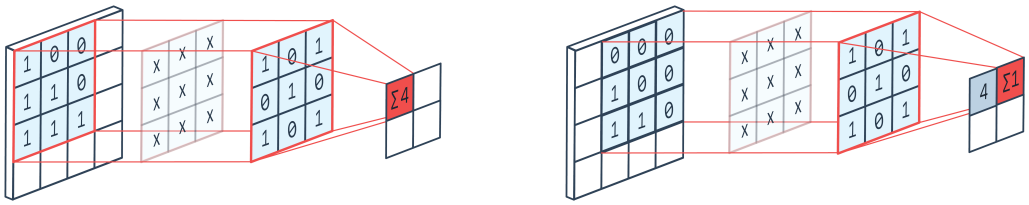

FIGURE 2.6: 2D convolution: URL.

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.

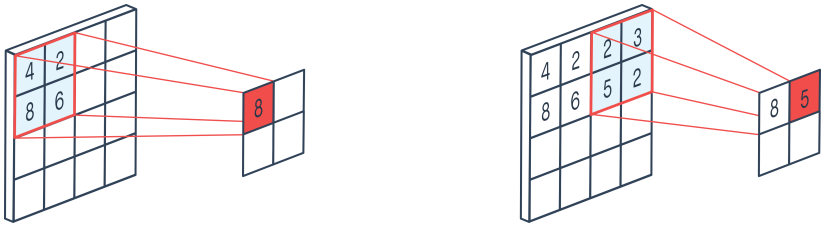

FIGURE 2.7: 2D max pooling: URL.

rnn?

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

[2] Also called fitting.

learning algorithms, such as Maximum Likelihood, Maximum A Posteriori, 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 layerδ^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.

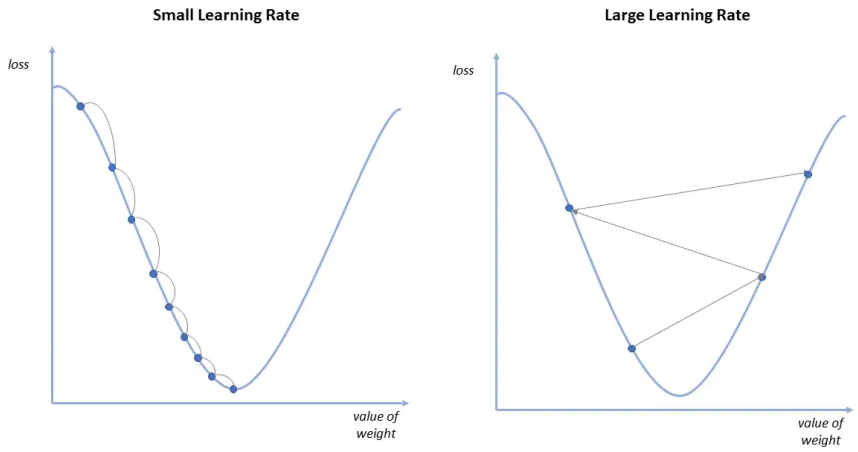

FIGURE 2.8: Effect of learning rate in Gradient Descent: URL.

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.

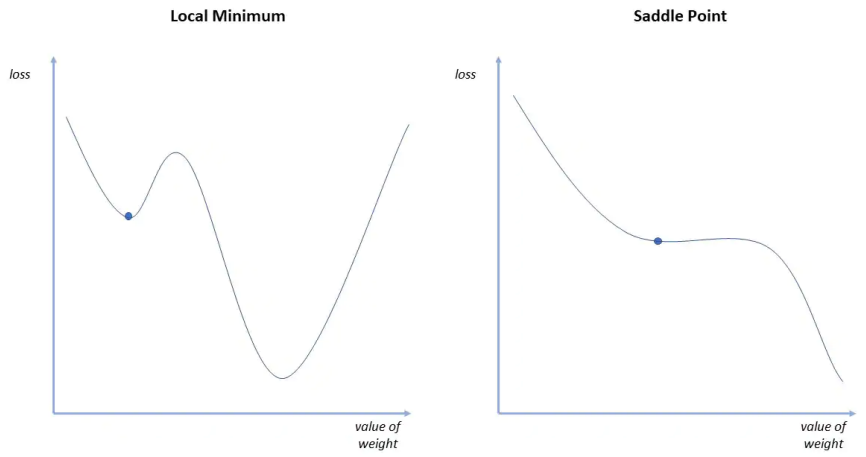

FIGURE 2.9: Local minimum and saddle point: URL.

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.

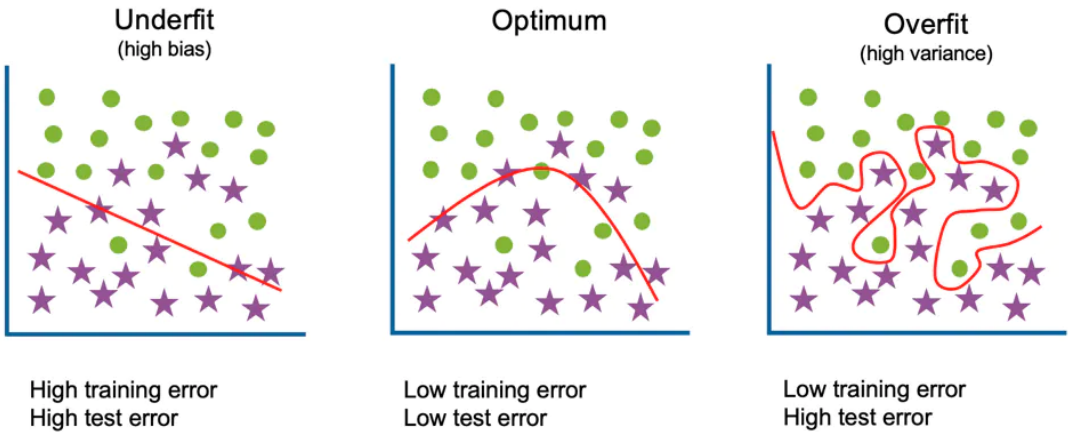

FIGURE 2.10: Model overfitting and its opposite, model underfitting: URL.

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.

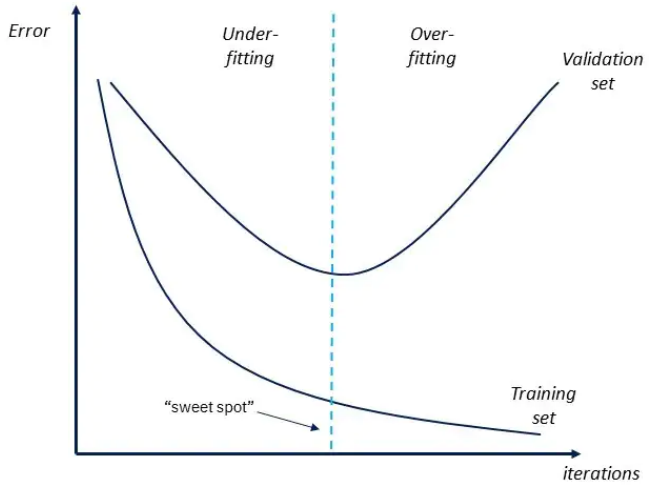

FIGURE 2.11: Border between overfitting and underfitting: URL.

- 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.

2.3.6 Adaptive Learning Rate?

2.3.7 Adaptive algoritms?

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.

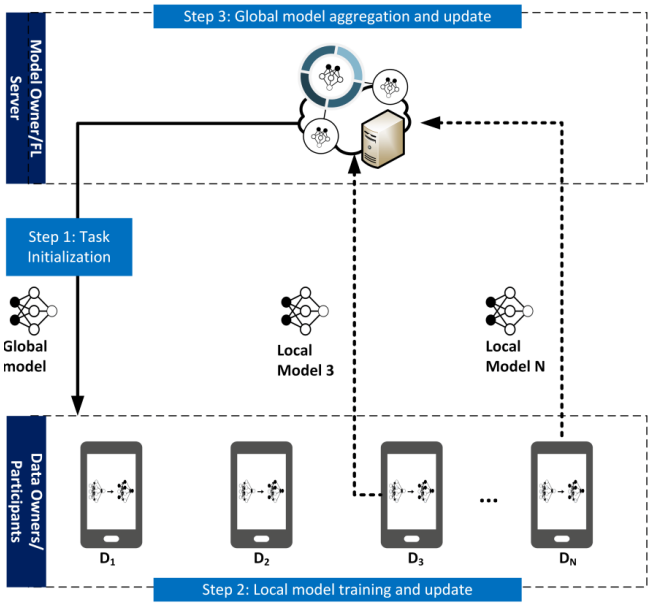

FIGURE 2.12: Typical FL topology: URL.

2.4.1 Typical Federated Training Process

FL training is a continuous process. Each iteration is referred as a global epoch 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. 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 [37] 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 Categorization?

horizontal - vertical - transfer learning?

Architectures for a federated learning system

2.4.4 Core challenges or Unique characteristics and issues 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.

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.

Systems heterogeneity

In FL, the computational and communication capabilities of the clients may greatly vary. The client devices may differ in architecture (CPU, GPU, FPGA) with different resources and capabilities. Furthermore, they may use different networks (4G, 5G, wifi, etc.) with varying reliability and bandwidth. The devices can also have different levels of willingness to participate

Heterogeneous devices

Statistical heterogeneity

due to data privacy concerns, the FL servers are unable to check for training data quality

Privacy concerns

2.4.5 Communication cost

Edge and End Computation

decrease the number of communication rounds, adding computation: increase parallelism by increasing participants per global epoch increase computation per global epoch, more local epochs smaller minibatch FedAvg increases the accuracy eventually since model averaging produces regularization effects similar to dropout, which prevents overfitting the heterogeneity among edge devices, e.g., in computing capabilities, is often not considered inroduce straggler effect when too many local updates are implemented between communication rounds, the communication cost is indeed reduced but the convergence can be significantly delayed

Model Compression

Commonly used in DL purpose is to reduse size of updates Structured updates restrict participant updates to have a prespecified structure sketched updates refer to the approach of encoding the update in a compressed form before communication with the server like model or gradient compression using sparsification,quantization, subsambling etc lossy compression on server updates no acceptable level of performance on client updates work, update errors can be averaged out for participant to server uploads federated dropout?

Importance-based Updating

only the important or relevant updates are transmitted in each communication round. besides saving on communication omitting can even improve the global model performance. such strategies include if loss greater than previous update, do not send uploads only relevant local model, local update is first compared with the global update lower accuracy and convergence guarantees

2.4.6 Data distribution

iid - non iid

Non-identical client distributions

Feature distribution skew (covariate shift) B Label distribution skew (prior probability shift) B Same label, different features (concept drift) B Quantity skew or unbalancedness B Violations of independence B Dataset shift? B

statistical challenges?

2.4.7 communication security

Secure aggregation - untrusted server Differential privacy - gan attacks data poisoning attacks model poisoning attacks

somewhere talk about stragglers and participant selection

Chapter 3

Related Work

tf and tff

3.1 Training Datasets

Common datasets and FL datasets from TF.

3.2 ANN architectures

nns in library

3.3 parallelsgd

3.4 FedAvg

applied in googles gboard

fedavg algorithm

3.5 CE-FedAvg

3.6 Evolution of FL

Fig 5 from A review of applications in federated learning

3.7 quantization?

3.8 The FPGA Perspective

3.9 Thesis Approach

Chapter 4

Robustness Analysis

Maybe this need its own chapter
Developed FL architecture. Server - Client architecture, communication protocol, tf embeddment. Interface - code agnostic of NN design and training.

The experiments from the reviews. Add a prologue to show why they exist. Remake the supplementary ones with the latest settings.

4.1 Experiment A

4.2 Experiment B

Chapter 5

FPGA Implementation

5.1 Tools Used

5.1.1 Vivado IDE

5.1.2 Vivado High Level Synthesis (HLS)

5.1.3 Vivado SDK

5.2 FPGA Platforms

Chapter 6

Results

6.1 Specification of Compared Platforms

6.2 Power Consumption

6.3 Energy Consumption

6.4 Throughput and Latency Speedup

6.5 Final Performance

Chapter 7

Conclusions and Future Work

7.1 Conclusions

7.2 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, 17–19 Jun 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] Andrew Hard et al. *Federated Learning for Mobile Keyboard Prediction*. 2018. DOI: 10.48550/ARXIV.1811.03604. URL: https://arxiv.org/abs/1811.03604.