Université de Montréal

Learning A Graph Made of Boolean Function Nodes: A New Approach in Machine Learning

par
Mouna Mokaddem

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en Informatique

August, 2016

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Learning A Graph Made of Boolean Function Nodes: A New Approach in Machine Learning

présenté par:

Mouna Mokaddem

a été évalué par un jury composé des personnes suivantes:

PrénomPrésident NomPrésident, président-rapporteur
PrénomDirecteur NomDirecteur, directeur de recherche

Mémoire accepté le: . . . . . .

RÉSUMÉ

Dans ce document, nous présentons une nouvelle approche en apprentissage machine pour la classification. Le cadre que nous proposons est basé sur des circuits booléens, plus précisément le classifieur produit par notre algorithme a cette forme. L'utilisation des bits et des portes logiques permet à l'algorithme d'apprentissage et au classifieur d'utiliser des opérations vectorielles binaires très efficaces. La qualité du classifieur, produit par notre approche, se compare très favorablement à ceux qui sont produits par des techniques classiques, à la fois en termes d'efficacité et de précision. En outre, notre approche peut être utilisée dans un contexte où la confidentialité est une nécessité, par exemple, nous pouvons classer des données privées. Ceci est possible car le calcul ne peut être effectué que par des circuits booléens et les données chiffrées sont quantifiées en bits. De plus, en supposant que le classifieur a été déjà entraîné, il peut être alors facilement implémenté sur un FPGA car ces circuits sont également basés sur des portes logiques et des opérations binaires. Par conséquent, notre modèle peut être facilement intégré dans des systèmes de classification en temps réel.

Mots clés : Apprentissage machine, classification, classifieur, données privées, FPGA.

ABSTRACT

In this document we present a novel approach in machine learning for classification. The framework we propose is based on boolean circuits, more specifically the classifier produced by our algorithm has that form. Using bits and boolean gates enable the learning algorithm and the classifier to use very efficient boolean vector operations. The accuracy of the classifier we obtain with our framework compares very favourably with those produced by conventional techniques, both in terms of efficiency and accuracy. Furthermore, the framework can be used in a context where information privacy is a necessity, for example we can classify private data. This can be done because computation can be performed only through boolean circuits as encrypted data is quantized in bits. Moreover, assuming that the classifier was trained, it can then be easily implemented on FPGAs (i.e., Field-programmable gate array) as those circuits are also based on logic gates and bitwise operations. Therefore, our model can be easily integrated in real-time classification systems.

Keywords : machine learning, classification, classifier, private data, FPGA.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNA	Deoxyribonucleic Acid
DNN	Deep Neural Network
RF	Random Forest
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
ML	Machine Learning
MLP	Multilayer Perceptron
SNN	Spiking Neural Network
SVM	Support Vector Machine

CHAPITRE 1

INTRODUCTION

Recently, machine learning classification has been used for several tasks such as medical diagnosis, genomics predictions, face recognition and financial predictions. More particularly some machine learning models have shown impressive results on classification tasks including deep neural networks. However, the digital implementation of these models is greedy in terms of computations (i.e., floating-point or fixed-point multiplication). In this document, we will introduce a new approach in machine learning classification that generates a classifier which is based on a boolean circuit. Using bits and boolean gates enables the learning algorithm and classifier to use very efficient boolean vector operations making it suitable for consumer applications on low-power devices. Moreover, in some of the applications mentioned at the beginning of this paragraph, there exist concerns about information privacy. In other terms, it is important that the data remains confidential. And lately, more interest has been drawn, in the scientific community, to the question of how we can apply machine learning classification on encrypted data [1, 3, 10]. Our approach can address perfectly this issue since in such context, computation can be performed only through boolean circuits as encrypted data is quantized in bits. In this project, our main objective is to study the characteristics of a supervised binary classification algorithm that meets the requirements of an environment where data privacy is a necessity. Indeed, the framework we will present is based on boolean circuits, more specifically, the classifier produced by our algorithm has that form. Furthermore, assuming that the classifier was trained, it can then be easily implemented on FPGAs (i.e., Field-programmable gate array) as those circuits are also based on logic gates and bitwise operations. In fact, FPGAs have abundant logic resources (i.e., logic gates), so they can carry out and speed up the computation of our classifier. Moreover, their power efficiency and portability make them ideal for real-time classification systems.

Although our original motivation was unsupervised learning, we have decided to concentrate first on supervised learning. Once one is acquainted with the framework,

a straightforward supervised learning algorithm for binary classification emerges naturally. Studying the properties of this simple algorithm we find its performance surprisingly good, both in terms of efficiency and accuracy. It offers an interesting alternative to neural nets and support vector machines. This is despite the relative maturity of these established techniques, compared with our new framework.

The outline of this document is as follows : in chapter 2, we will set the context of our project by reviewing some basic concepts related to machine learning and some approaches that were compared to ours. Chapter 3 will be for describing the new framework we propose then presenting two fields related to the context of our approach which are FPGAs and secure multi-party computation. In chapter 4, we will introduce the algorithm describing its three main steps, greedy initialization, leaf's optimization and node's optimization followed by a discussion about conceptual similarities and differences with neural nets and random forests. Chapter 5 will be for reporting experimental results and comparison with other approaches' baselines. Finally, the last chapter will be to conclude and give potential future improvements.

CHAPITRE 2

PRELIMINARIES

In this chapter, we will present some basic concepts related to the context of our project. We will define them formally and for every concept we will discuss points that have been useful either to build our approach or to compare it to other approaches of the literature.

2.1 Machine Learning

2.1.1 Definition

Machine learning is a branch of artificial intelligence that provides computers with the ability to learn autonomously from data. It studies the design of algorithms that learn from input observations, also referred to as examples, instead of relying on hard coded rules. This field, being in the intersection of AI and statics, is about building a model that extracts knowledge from a training set (i.e., the set of examples) in order to make predictions or decisions on unseen examples. The parts of knowledge that a machine learning algorithm tries to capture from data are called patterns and/or features. We will use both words in the rest of the document to express the aforementioned meaning. There are several real-world problems that can be solved using machine learning such as pattern recognition, medical diagnosis, machine translation, self-driving cars, etc. There exists different machine learning scenarios that can be classified according to the types of training data available to the learner, the order and method by which training data is received and the test data used to evaluate the learning algorithm[20]. Instances of those scenarios are supervised learning, unsupervised learning, reinforcement learning, etc. Our approach that will be presented later in the document describes a supervised algorithm, for that reason we will focus on the supervised aspect of machine learning in the following definitions.

2.1.2 Supervised Learning

Supervised learning also known as predictive modelling is the process of making predictions using labelled data. In fact, the supervised learning algorithm is supplied a training dataset D that consists of a set of pairs $\{z_i\}_{i=1}^n = \{(x_i, y_i)\}_{i=1}^n$ where $x_i \in \mathbb{R}^d$ is an input object and $y_i \in \mathbb{R}$ the desired output of x_i (also known as target) and is required to find a decision function f that will yield a prediction $f(x_i^{test}) = \hat{y}_i$. Hence the task of learning here is to capture the mapping between the inputs and its corresponding outputs in order to make predictions on unseen data later on. In this document, we will consider examples $\{z_i\}_{i=1}^n \in D$ to be independent and identically distributed (i.i.d). When the target y_i is continuous, the process is called **regression**. When the target y_i is discrete, the learning task consists in a **classification**. For example f may predict an animal species or a handwritten digit. Our contribution, that will be presented in the following chapters, consists of an algorithm for classification based on a new approach. There are two variations of classification depending on the value(s) that the target y_i can take : binary classification is when y_i is a binary value whereas multi-category classification when y_i can take three values or more. The performance of f is estimated through a loss function $L(f,z)$ and a dataset D which defines the empirical risk $\hat{R}$ also known as the expected loss in $D \in \mathbb{R}^d$:

$$\hat{R}(f,D) = E_D[L(f,z)] = \frac{1}{n}\sum_{i=1}^{n} L(f,z_i), D = \{z_1, ..., z_n\} \tag{2.1}$$

L can be of different natures depending on the task to be considered. For classification tasks, L can be chosen to count misclassification error as :

$$L(f,(x,y)) = 1_{(f(x)\neq y)} \tag{2.2}$$

In which case, $\hat{R}(f,D)$ will measure the average classification error rate of decision function f over dataset D. So learning means finding the best function $\hat{f}^*$ in F that minimizes the empirical risk $\hat{R}$ on the training set (a regularization term $\Omega(f)$ can be

added, further details for regularization will be given later in the chapter) :

$$\hat{f}^* = \arg\min_{f \in F} J(f) \tag{2.3}$$

$$J(f) = \hat{R}(f, D_{train}) + \Omega(f)$$

Supervised learning comes in contrast of unsupervised learning that tries to extract structure from unlabelled data.

2.1.3 Training-Validation-Testing

The dataset D used to estimate the risk is finite thus the empirical risk is a biased estimator. In fact, to estimate the real risk a training set that contains an infinite number of examples should be used in order to model perfectly the distribution of D. However practically speaking, this is impossible. Moreover, when minimizing the empirical risk the model is encouraged to perform better on the trained points of the training set than on the other points of the dataset. Therefore, any performance measured on the training set will be most of the time biased (i.e., optimistic). For that reason, the dataset is in practice divided into three subsets : **training** set D_{train}, **validation** set D_{valid} and **test** set D_{test}, the last two are generally smaller than the first one. We make sure that the model will generalize well on examples other than training examples in the following way : f is trained on D_{train} until its error doesn't further improve on D_{valid} (i.e., D_{valid} is used to choose good values of hyperparameters : we will introduce the concept of hyperparameters in the next subsection), finally its performance will be measured over D_{test}. Without this division, a model could learn to perform well on D just by stocking the dataset's examples and their corresponding targets yet not be able to conserve the same performance for unseen data (i.e., the model is not able to generalize well over examples).

2.1.4 Model's complexity, Hyperparameters, Bias-variance tradeoff

There are two types of models in machine learning, **parametric** and **non-parametric** models. Parametric models are based on a fixed number of parameters θ (scalars, vectors or matrices) to characterize their choice of f in the space of functions F. Therefore, the learning task will consist in optimizing f_θ i.e., minimize $J(f_\theta)$ in order to find the best parameters. Gradient descent is a good example of an optimization algorithm used to train Artificial Neural Networks. Non parametric memory-based models use the training set for essentially memorizing it directly to model the distribution of D. Kernel SVM is a well-known example of non parametric models [4]. The work presented in this document is a parametric machine learning algorithm that trains by using greedy optimisation and hill climbing.

The number and dimensions of parameters θ define the size of the space of functions F which is referred to as the **model's complexity**. There exists a point of optimal complexity which corresponds to the lowest value of the generalization error. The number of parameters can greatly influence the quality of the classifier. Variables that are not learned by the algorithm like the number of parameters are called **hyperparameters**. In fact, they play a role, among other roles, in controlling the model's complexity. When F is big (i.e., the model's complexity is high), we have more flexibility to choose the best function $\hat{f}^*$ that minimizes the real empirical risk. However, for a high model's complexity, the learning algorithm will tend to learn too closely D_{train} but poorly model the true distribution that data comes from, i.e., it will generalize poorly on other samples from that distribution and thus the generated prediction function f will give an empirical error equal or close to zero over D_{train} but a very high one over D_{tvalid} and D_{test}. This phenomenon is called overfitting. In contrast, reducing too much the model's complexity will certainly prevent f from overfitting its training set but f won't be able to model adequately the dataset. Here the problem is referred to as underfitting. A low complexity implies a high **bias**, a high complexity implies a high **variance** . Therefore, there should be a tradeoff between the model's complexity and the generalization over examples, this is called the **bias-variance tradeoff**. The balance between the bias and

the variance is controlled by hyperparameters (i.e., also controlled by regularization, which will be introduced in the following).

2.1.5 Model Selection : randomized search, grid search

Hyperparameters control a model's complexity and thus the ability of the model to learn and generalize well. The procedure to select the best values of hyperparameters is referred to as model selection. As aforementioned, the model should be trained on the training set (i.e., D_{train}) to find parameters then tested on the validation set (i.e., D_{valid}) to find values of hyperparameters that minimize the error. Note here that the model should not be tested on the test set (i.e., D_{test}) until best values of hyperparameters will be found on the D_{valid}. A solution on how to construct a validation set is to divide the dataset into two parts of the corresponding proportions : the majority of examples for training set 80% D_{train} and the rest 20% D_{valid} will serve to compare different values of hyperparameters. The values of hyperparameters that give the best performance on the validation set D_{valid} are selected and then the selected model will be applied on the test set D_{test} to estimate its expected generalization performance. There are two generic approaches in order to choose the list of values of hyperparameters to be tested : **randomized search** which samples a given number of candidates from the hyperparameter space according to a specified distribution and **grid search** which considers all hyperparameter combinations on a grid. In fact in grid search, we dress a list of the to be tested values of every hyperparameter of the model then construct the list of every possible combination of those hyperparameters (i.e., constructing a grid of hyperparameters) that way we will exhaustively consider every combination. In our project, we chose to use grid search following our intuition about the values of hyperparameters that should be tested and thus we were able to dress a list of suitable candidates.

2.1.6 Regularization

The complexity of the model f and consequently the risk of overfitting over D_{train} can be reduced artificially by what is referred to as regularization. Two common methods

of regularization are : 1. add a term to the objective function. 2. inject noise. As for the former, it consists in adding to the objective function, during the training, a term that penalizes some parameters of the model under certain conditions. For the latter, an artificial noise is added to the input or/and output samples in the training process which will improve the model's robustness regarding input inaccuracies to avoid overfitting. Similarly to the "dropout" technique used when training neural networks [27], we will use the second method in our algorithm by artificially injecting noise at all levels during the training step.

2.2 Graphs and Trees

Definition 1. *A graph is a representation of a set of objects, called nodes, where some of them are interconnected. The link that connects a pair of nodes is called edge [30]. The edges may be directed or undirected. A degree of a node of a graph is the number of edges incident to the node [5]. In this document we will use directed graphs where the degree of all nodes will be the same (i.e., the degree is equal for all nodes).*

Definition 2. *A tree is a connected graph with no cycle (i.e., a walk consists of a sequence of nodes starting and ending at the same node, with each two consecutive nodes in the sequence adjacent to each other in the graph). The edges of a tree are known as branches. Elements of trees are called nodes. The nodes without child nodes are called leaf nodes.*

Definition 3. *A forest is an acyclic graph. It can be described as the disjoint union of one or more trees.*

2.3 Entropy

Definition 4. *The entropy of a discrete random variable X with a probability mass function (i.e., pmf) $p_X(x)$ is*

$$H(X) = -\sum_x p(x) \log p(x) = -\mathbb{E}[\log(p(x))] \tag{2.4}$$

The entropy measures the expected uncertainty in X. We also say that H(X) is approximately equal to how much information we learn on average from one instance of the random variable X. Customarily, we use the base 2 for the calculation of entropy.

Consider now two random variables X,Y jointly distributed according to the p.m.f p(x,y). We now define the following two quantities :

Definition 5. *The joint entropy is given by*

$$H(X,Y) = -\sum_{x,y} p(x,y)\log p(x,y) \tag{2.5}$$

The joint entropy measures how much uncertainty there is in the two random variables X and Y taken together.

Definition 6. *The conditional entropy of X given Y is*

$$H(X|Y) = -\sum_{x,y} p(x,y)\log p(x|y) = -\mathbb{E}[\log(p(x|y))] \tag{2.6}$$

The conditional entropy is a measure of how much uncertainty remains about the random variable X when we know the value of Y.

Properties. *The entropic quantities defined above have the following properties :*

- **Non negativity :** $H(X) \geq 0$, entropy is always non-negative. $H(X) = 0$ iff X is deterministic.
- **Chain rule :** We can decompose the joint entropy as follows :

$$H(X_1, X_2, ..., X_n) = \sum_{i=1}^{n} H(X_i|X^{i-1}) \tag{2.7}$$

  where $X^{i-1} = \{X_1, X_2, ..., X_{i-1}\}$ For two variables, the chain rule becomes :

$$\begin{aligned} H(X,Y) &= H(X|Y) + H(Y) \\ &= H(Y|X) + H(X) \end{aligned} \tag{2.8}$$

  Note that in general $H(X|Y) \neq H(Y|X)$

- **Monotonicity :** Conditioning always reduces entropy :

$$H(X|Y) \leq H(X) \tag{2.9}$$

- **Maximum entropy :** Let χ be set from which the random variable X takes its values, then

$$H(X) \leq \log|\chi| \tag{2.10}$$

The above bound is achieved when X is uniformly distributed.

2.4 Support Vector Machine

Support vector machines (SVMs) are supervised learning models associated with learning algorithms that perform, amongst other type of classification, binary linear classification. A binary linear classifier can be visualized as a model that splits a high dimensional input space D into two regions $C1$ and $C2$ with a **hyperplane** H referred to as a decision boundary and then classifying a new example $x_{test} \in \mathbb{R}^d$ will be according to its position from H. The hyperplane H is defined by a weight vector $w \in \mathbb{R}^d$ and a bias $b \in \mathbb{R}$ which are parameters of the model. The function that tells whether x_{test} belongs to one region or another (i.e., $C1$ or $C2$) is called **discriminative function** defined by :

$$y(x) = w^T x + b = \sum_i w_i x_i + b \tag{2.11}$$

The decision function that predict the category of x_{test} is :

$$g(x) = \begin{cases} C1, & \text{if } y(x) > 0 \\ C2, & \text{if } y(x) < 0 \end{cases} \tag{2.12}$$

The model we describe requires that D to be linearly separable (i.e., this condition is generally not respected as certain approximations may be done, we will present them below) which is not always the case but we will see later that SVMs could perform a non-linear classification by using what is called the **kernel trick**. Given a training data-

set of n examples of the form $(x_1,t_1),...,(x_n,t_n)$ where $x_i \in \mathbb{R}^d$ and $t_i \in -1,1$, SVM's model aims at finding the maximum-margin hyperplane that divides the group of points x_i whose $t_i = -1$ from the group of points whose $t_i = 1$. The margin is defined as the distance between the hyperplane and the nearest point x_i from either groups. Formally it is a signed distance given by : $\frac{t_i y(x_i)}{||w||} = \frac{t_i(w^T x_i + b)}{||w||}$, if x_i is well-classified then the margin will be positive, else it will be negative. SVM will be about finding the hyperplane that maximizes the margin which is formally expressed by :
$\arg\max_{w,b}\{\frac{1}{||w||}\min[t_i(w^T x_i + b)]\}$
To prevent data points from falling into the margin, we add the following constraint :
$t_i(w^T x_i + b) \geq 1$
Consequently, $\min[t_i(w^T x_i + b)] = 1$. Moreover, as mentioned before, some approximations are done to get around the constraint of linear separability : some terms will be added to the objective function. Those additional terms are referred to as slack variables and express that all predictions have to be within an ε range of the true predictions. We can write the optimization problem of SVM as the following :

$$\begin{aligned} &\arg\min_{w,b,\varepsilon_n} \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\varepsilon_i \\ s.t \quad & t_i(w^T x_i + b) \geq 1 - \varepsilon_i \\ & \varepsilon_i \geq 0 \\ & for \quad i = 1,...,n \end{aligned} \tag{2.13}$$

We have previously mentioned that there is a way to construct a non-linear classifier for the maximum-margin hyperplane algorithm (i.e., SVM), the kernel trick. "The resulting algorithm is formally similar, except that every dot product is replaced by a non-linear kernel function. This allows the algorithm to fit the maximum-margin hyperplane in a transformed feature space. Although the classifier is a hyperplane in the transformed feature space, it may be nonlinear in the original input space" [32]. In our project, we are using the radial basis function kernel (RBF kernel) which is defined for two examples x_i and x_j as :

$K(x_i, x_j) = \exp(-\frac{||x_i - x_j||^2}{2\sigma^2})$ where $||x_i - x_j||$ the squared euclidean distance the of two example vectors and σ is a free parameter. The reader interested in kernel method is invited to read [12].

2.5 Decision Trees

In machine learning, decision trees are non-parametric supervised learning methods that use a tree-like graph as a predictive model. They are commonly used for classification (also can be used for regression) and are referred to as **classification trees**. Decision trees can perform both binary and multiclass classification. Classification trees have a particular structure : leaves represent class labels, nodes represent attributes (i.e. unlearned feature) of input samples and branches represent conjunctions of values that an attribute can take. During training, the tree is learned from top to bottom (i.e., from the root node to the leaves) by splitting the training set into subsets based on an attribute value test. This process is recursive i.e., repeated on each derived subset and stops when the subset at a node has all the same value of the target variable, or when splitting no longer adds value to the predictions. For every iteration (i.e., recursion), an attribute is chosen according to a metric called the information gain. In fact, for all attributes calculate the information gain and choose the one that has the biggest value. The information gain of an attribute expresses the reduction of entropy brought if the attribute was chosen. Let A be a chosen attribute with k distinct values that divides the training set D_{train} into subsets $D_{train_1}, D_{train_2}, ..., D_{train_k}$ for a binary classification problem then the **expected entropy (EH)** remaining after trying attribute A (with branches $i = 1, 2, ..., k$) is :

$$EH(A) = \sum_{i=1}^{k} \frac{p_i + n_i}{p + n} H(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i}) \tag{2.14}$$

where $p + n$ are the total of examples (i.e., negative and positive examples) in the parent node, $p_i + n_i$ are examples in child i and $H(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i})$ is the entropy of the child i

given by the generic following formula :

$$H(\frac{p}{p+n},\frac{n}{p+n}) = -\frac{p}{p+n}\log\frac{p}{p+n} - \frac{n}{p+n}\log\frac{n}{p+n} \tag{2.15}$$

And then the information gain of the attribute A is defined as :

$$I(A) = H(\frac{p}{p+n},\frac{n}{p+n}) - EH(A) \tag{2.16}$$

where H is the entropy of the current node.

For an unseen example, we have to follow the simple tests given by the trained or learned tree until ending up in a leaf and then the class probability for that new example is the distribution at the leaf.

The process of building the tree top-down, one node at a time, is extremely efficient. However, the problem with it is that the tree will have variance because one can change one or more of the input examples and the structure of the tree will change, we will end up with a different tree. A solution of that problem will be random forest : build a forest of many different trees and then average them under uncertainty. We will further describe random forests in the last section of this chapter.

2.6 Artificial Neural Networks : Multilayer Perceptron

Artificial neural networks are a family of models in machine learning inspired by known similarities with Human brain in terms of the structure (i.e., architecture) and the functional abilities. Generally speaking, both of them are composed of layers of neurons that process the information that comes from the previous layer. The first layer is the data input vectors and the last one is for prediction. An affine transformation possibly followed by a non linearity composes the layer called output layer, taking as input the learned representation of an intermediate layer called hidden layer, which itself becomes an input layer. This input layer corresponds to either a normalization of the original data, or identity. This layered architecture gives the neural network more capacity than a model of flatter architecture, with the same number of parameters. In [13], Hornik and al.

presented what they called the universal approximation theorem stating that "a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units" that is to say it can model any type of function provided that it has sufficient capacity.
Formally, a neural network with a single hidden layer is a function $\mathbb{R}^d \mapsto \mathbb{R}^m$ such that :

$$\hat{y} = f(x) = o(b^{(2)} + W^{(2)^T} h^{(1)} x) \tag{2.17}$$

$$with \quad h^{(1)}(x) = g(a(x)) = g(b^{(1)} + W^{(1)^T} x) \tag{2.18}$$

where x $\in \mathbb{R}^d$ the input vector, $b^{(1)} \in \mathbb{R}^h$ and $W^{(1)} \in \mathbb{R}^{d \times h}$ biases and weights of the hidden layer, $b^{(2)} \in \mathbb{R}^L$ and $W^{(2)} \in \mathbb{R}^{h \times L}$ biases and weights of the output layer of size L, a is the hidden layer pre-activation, $h^{(1)}$ and o are the activation functions of the hidden layer and the output layer respectively. Most of the time, they are non linear functions such as sigmoid or hyperbolic tangent. Note here that f describes a single hidden layer neural network. Deep networks can be obtained by adding more hidden layers. This model is called Multilayer Perceptron (MLP). Neural networks are trained using the algorithm of stochastic gradient descent. This algorithm is composed by two steps : the first one for calculating the output prediction (and loss) starting from an input example, is called forward propagation, the second one for calculating gradients of the loss with respect to network parameters is referred to as back propagation.

There are many types of artificial neural networks depending of the architecture of the network, e.g., recurrent neural network, convolutional neural network, spiking neural network, etc.

To help regularize the training of deep neural networks (i.e., neural networks with several hidden layers), the technique of drop out was introduced [27]. This technique is inspired by the fact that adding noise (i.e., randomly dropping units with their connections from the neural network) during training will prevent neural network from overfitting.

2.7 Ensemble methods

2.7.1 Definition

Ensemble methods are meta-learning techniques that create several base estimators with a given learning algorithm and then combine their predictions in order to improve generalizability and robustness over a single estimator [23].**Voting** and **averaging** are two of the easiest ensemble methods. Voting is used for classification and averaging is used for regression. For classification, the main advantage of ensembles of different classifiers is that it is unlikely that all classifiers will produce the same error. In fact, as long as every error is made by a minority of the classifiers, the estimator will achieve optimal classification. In particular, ensembles tend to reduce the variance of the resulting meta-classifier. So if the initial classification algorithm tends to be very sensitive to small changes in the training data, ensembles are likely to be useful to reduce variance. Two common types of ensemble methods are bagging and boosting.

2.7.2 Boosting

Boosting is a machine learning ensemble meta-algorithm that builds a powerful estimator by combining several weak estimators. A weak estimator is an estimator that performs at least slightly better than random guessing."The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction. The data modifications at each so-called boosting iteration consist of applying weights $w_1, w_2, ..., w_N$ to each of the training samples. Initially, those weights are all set to $w_i = \frac{1}{N}$, so that the first step simply trains a weak learner on the original data. For each successive iteration, the sample weights are individually modified and the learning algorithm is reapplied to the reweighted data. At a given step, those training examples that were incorrectly predicted by the boosted model induced at the previous step have their weights increased, whereas the weights are decreased for those that were predicted correctly. As iterations proceed, examples that are difficult to predict receive ever-increasing influence. "Each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence"[26]. Boosting mainly

reduces bias and thus helps to avoid underfitting.

2.7.3 Bagging

Bagging is a machine learning ensemble meta-algorithm which builds several instances of an estimator on random subsets of the original training set and then aggregate (i.e., each instance of the ensemble votes with equal weight) their individual predictions to form a final prediction : Given a training set D_{train} of size n, bagging generates m new training subsets D_i each of size n' by sampling from D_{train} uniformly and with replacement then constructing different instances f_i of the estimator on the corresponding subsets D_i. The resulting ensemble classifier then simply averages or performs a majority vote of the predictions of all trained instances. Bagging is used to reduce variance of a base estimator by introducing randomization into its construction procedure and then making an ensemble out of it. Therefore it will help to avoid overfitting.

2.8 Random Forests

As mentioned in section 2.5, decision trees suffer from a lot of variance as the structure of the learned tree is very sensitive to the input examples. Another problem to point out for decision trees is that practically, the input space is often a high-dimension space so there is a big amount of information gains that have to be evaluated for each node so the computation will be very heavy. Random forests can alleviate the two aforementioned problems by adding two sources of randomness : one in input data and the other in splitting the features. A random forest is a set of random decision trees. Each random decision tree is built as follows : Let the training data set be $\{(x_1, y_1), ..., (x_n, y_n)\}$ and let F a random forest with B trees (i.e., $b = 1, ..., B$)

1. Draw uniformly at random from the training data set a sample of size N (with replacement), this step is called bootstrapping. Basically every tree will be constructed on a different subset of the training data set.
2. Grow a random-forest tree T_b to the bootstrapped data by recursively repeating the following steps for each node of the tree until the minimum node size n_{min} is

reached.

(a) Select m variables at random from the p (i.e., the number of attributes) variables.

(b) Pick the best variable/split-point among the m.

(c) Split the node into two or more daughter nodes.

The output of the algorithm will be all the trees $\{T_b\}_1^B$.

In a forest with T trees, $t \in \{1, ..., T\}$, all trees are trained independently and possibly in parallel. During testing, each test point v is simultaneously pushed through all trees (starting at the root) until it reaches the corresponding leaves. Each tree gives a different probability of how much the point v belongs to a certain class c. We will average trees output probability class, this is given formally by the following expression :

$$p(c|v) = \frac{1}{T} \sum_{t=1}^{T} p_t(c|v) \tag{2.19}$$

The bootstrapping procedure yields better model performance because it reduces the variance of the model, without increasing the bias. In fact, although the predictions of a single tree are highly sensitive to a change in the input examples, the average of many trees is not, as long as the trees are not correlated. Because the base decision tree algorithm is deterministic, training many trees on a single training set would give strongly correlated trees, bootstrap sampling is a way of de-correlating the trees by showing them different training subsets. Furthermore, putting randomness in the selection of features (i.e., attributes) decreases the bias as random forests will be able to work with very large number of features.

CHAPITRE 3

BOOLEAN CIRCUIT

In this chapter, we will describe and formalize the new framework we propose. In contrast with approaches that use arithmetic operation on real numbers, we present a framework based on binary number (bits) and boolean circuits. In short, inputs are binary vectors of a given length, and classifiers are boolean circuits. The input data could be images, text, lists of numbers or anything else encoded as fixed-length binary vectors. The classifiers produced by all versions of our algorithm are boolean circuits. Therefore all the classifiers we present in this document are binary circuits that input binary vectors, and output a single bit representing the classification decision. We will also introduce field-programmable gate arrays (FPGAs) and we will discuss their connexions with our approach as well as artificial neural networks. At the end, we will dedicate a section that describes how our approach can be applied in the field of cryptography.

3.1 Circuit

Definition 7. *In this document, a k-gate is a function mapping k inputs bits to an output bit. We call k the arity of the gate.*

Lemma 1. *The number of possible k-gates is* 2^{2^k} *and those possible gates can be completely specified by a lookup table of* 2^k *bits.*

For example, the boolean AND, OR and XOR gate are 2-gate. There is a total of 16 boolean binary gates with 2 inputs. The total number of possible gates for a given arity is surprisingly high. In the case of arity 8, there are

$$115792089237316195423570985008687907853269984665640564039457584007913129639936$$

different gates that can be computed, but each of those is uniquely defined by a truth table of 256 bits.

00	0
01	0
10	0
11	1

TABLE 3.I – 2-gate, AND

000	0
001	0
010	0
011	1
100	0
101	1
110	1
111	1

TABLE 3.II – 3-gate, Majority

Definition 8. *In this document, we define a* ***Boolean circuit classifier*** *as a Boolean circuit whose input bits are data bits and whose output bits represent the classification decision.*

We always restrict all the gates in a classifier to have the same arity (i.e., each non-leaf node has the same number of children), as a design choice. All the circuits we consider in this document will be directed graphs such that each leaf has the same distance (i.e., in edges) from the root (decision) node.

Definition 9. *The depth of a circuit is the length of the longest path from an input bit to the output.*

The depth of a circuit is an important characteristic, especially in the context of parallel computation, where it corresponds to the time necessary to evaluate the circuit (with a sufficient number of processors).

Definition 10. *By "the levels of a circuit", we mean a list containing the number of nodes of every level of the graph (i.e., circuit).*

To describe this circuit we need to specify the inputs for each gate (which may be some bits of the input, or outputs of other gates), as well as filling in 7 truth tables (i.e., one for each gate). Suppose now that we wish to evaluate this classifier on 64000 different examples (i.e., 10 bits each). Then the dataset can be stored as a table with 64000 rows of 10 bits each. The first step of every algorithm will always be to transpose the input. In this example we obtain a table with 10 lines of 64000 bits. On a 64 bit system

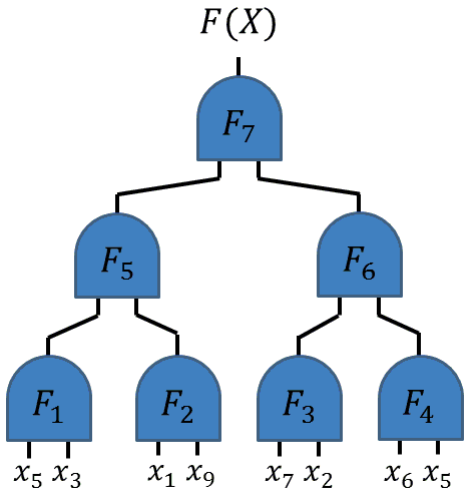

FIGURE 3.1 – A depth 3 classifier F made of 7 2-gates for data $x = (x_1; x_2; ...; x_{10})$

this requires 1000 memory words per line. Now assume that all the gates F_i are either AND, OR, or XOR gates. Evaluating this classifier on that data will require evaluating the 7 binary gates. Since the gates we have chosen can be evaluated by any reasonable system on words of 64 bits, the evaluation will require 7 vector operations on 1000 words tables, for a total of 7000 word operations. Note that here, 7000 is significantly smaller than 640,000 bits (i.e., the total size of the data). In general we will want to use more then 7 gates and more importantly might want to work with gates of arity larger then 2. Fortunately the AND, OR and NOT gates are more than enough to compute any computable function. We exploit this fact to develop a general technique for evaluating circuits with gates of any arity.

In this framework, we consider the input binary vectors as **unlearned features**, and the output of the gates, F_i as **learned features**. For each feature (i.e., a binary vector), we store its negation as well, to save operations.This representation will be useful in the implementation of gates. The **tensor product** of a list of k bits (i.e., boolean variables) is the list containing 2^k being the product (i.e., logical AND) of all possible combinations of every input and its negation.

Given that the truth table of a gate gives us its output for each input, the tensor product of the features given as inputs to the gate in question is in 1-1 correspondence

x_0	x_1	y_0	y_1	z_0	z_1	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
0	1	0	1	0	1	0	0	0	0	0	0	0	1
0	1	0	1	1	0	0	0	0	0	0	0	1	0
0	1	1	0	0	1	0	0	0	0	0	1	0	0
1	0	0	1	0	1	0	0	0	1	0	0	0	0
1	0	1	0	1	0	1	0	0	0	0	0	0	0

TABLE 3.III – $x \otimes y \otimes z = v$ for five training examples

with the truth table. The output of a gate is a feature (i.e., binary vector) where each element is the exclusive OR of every element of the tensor product position where the truth table has a value 0 and the complement (i.e., negation) of the output of the gate is the exclusive OR of every element of the tensor product position where the truth table has a value of 1. Using the fact that the output feature is composed of a vector and its complement one can spare some operations.

Lemma 2. *On an architecture with words of m bits, the evaluation of a k-gate when the size of the data set is n (a multiple of m) is* $(2^k + (2^{(k-1)} + 1))n/m$ *gates.*

Prior to the implementation we have made in python, some preliminary experiments have shown that on one core, using no parallelism (i.e., other than the fact that 32-bit words are used), using C# we can evaluate 27 million 4-gates over 32 bits. This means that a 5000 gate circuit with 5000 inputs can be evaluated in a second. Of course parallelism can be used both at the vector level and the circuit level, for significant speed-ups.

3.2 FPGA

An FPGA (i.e., field programmable gate array) is an integrated circuit that can be reprogrammable after manufacturing : it is an array of gates (i.e., programmable logic blocks) with programmable interconnect and logic functions that can be redefined after manufacture [11]. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete blocks

of memory [31].

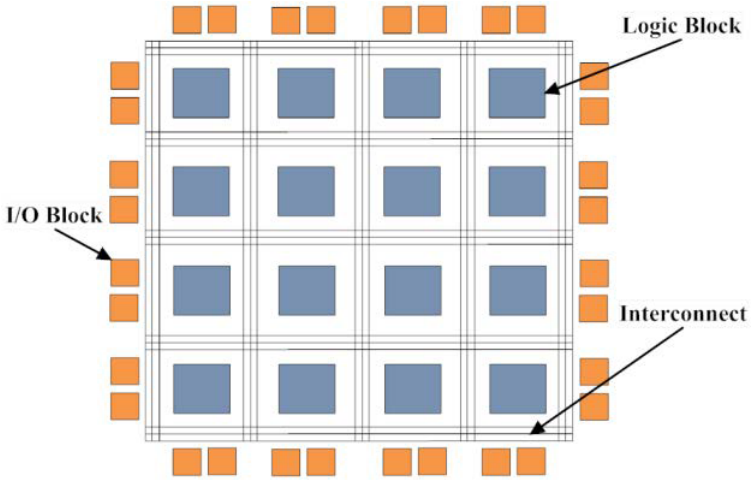

FIGURE 3.2 – FPGA's architecture

There are 4 main advantages of FPGAs :

1. **Performance** Taking advantage of hardware parallelism, FPGAs break the paradigm of sequential execution and thus exceed the computing power of digital signal processors (i.e., processors that are used for real-time calculation on a real-time stream of inputs like image and audio processing, face and gesture detection, ..., etc.) [14].
2. **Time to market** FPGA technology offers flexibility and rapid prototyping capabilities. An idea or a concept can be tested and verified in hardware without going through a long fabrication process of custom integrated circuit like ASIC design. A user can then implement incremental changes and iterate on an FPGA design within hours instead of weeks [29].
3. **Cost** System requirements often change over time, the cost of making incremental changes to FPGA designs is negligible when compared to the large expense of respinning a custom integrated circuit like ASIC.
4. **Reliability** While software tools provide the programming environment, FPGA circuitry is truly a "hard" implementation of program execution. Processor-based

systems often involve several layers of abstraction to help schedule tasks and share resources among multiple processes. The driver layer controls hardware resources and the OS manages memory and processor bandwidth. For any given processor core, only one instruction can execute at a time, and processor-based systems are continually at risk of time-critical tasks preempting one another. FPGAs, which do not use OSs, minimize reliability concerns with true parallel execution and deterministic hardware dedicated to every task [14].

Currently, many fields related to machine learning have been using FPGA technology because they require high computation performance as computer vision [8, 15], data mining [2] and deep learning [24].

The framework we propose is a boolean circuit classifier having its nodes logic gates and its input binary vectors so it can be easily implemented on an FPGA (i.e., which usually has programmable gates of arity $= 6$). Therefore, our algorithm can be integrated on real-time applications that require real-time classification of stream of inputs.

3.3 Artificial Neural networks and FPGA

Current artificial neural networks (ANN) are demonstrating impressive performance on a number of real-world computational tasks including classification, recognition and prediction. Neural networks and especially deep neural networks computations usually consume a significant amount of time and computing resources and they can be implemented either in software or directly in hardware such as FPGAs. Software implementations leverage costly high-end and power-hungry CPUs and GPUs to perform costly floating-point operations. FPGA-based implementations have the potential to greatly accelerate neural networks workload with low-power consumption making them an attractive solution for real-time embedded systems. But they cannot afford to carry out the expensive high-precision floating point operations that the software implementations use. FPGA approaches are thus typically restricted to using low-precision integer or fixed-point operations, or sometimes even just binary operations. In this section, we will briefly present FPGA-based implementations of some types of ANNs including convolu-

tional, spiking neural nets and restricted boltzmann machine. The reader should be aware that our intention is not to give an exhaustive list of studies addressing the hardware implementation of artificial neural nets on FPGA but merely pointers to recent relevant research.

Several works on artificial neural nets have addressed the issues of embeddability and power-efficiency using FPGA implementations. A popular architecture is the Neuflow by Pham and al. [25], the approach presents an FPGA-based stream processor for embedded real-time vision with convolutional neural networks (CNN). Neuflow is able to perform all operations necessary in ConvNets and it uses 16 bit fixed-point arithmetic. In 2014, Neuflow was further improved and renamed as nn-X [9]. A more recent study, by Zhan and al. [33], addresses the problem of under-utilization of either logic resources or memory bandwidth an FPGA platform provides. They started with the following observation : for any CNN algorithm implementation, there are a lot of potential solutions that result in a huge design space for exploration and so it is not obvious to find out the optimal solution. The approach proposes an analytical design scheme that quantitatively analyses the computing throughput and required memory bandwidth for any solution of a CNN design and then identify the solution with best performance and lowest FPGA resource requirements. In the previous work as well as similar works [8, 24], "the hardware does not compute layers simultaneously. As a result, it is necessary to use memory as a buffer to store the computation results and serve the input of the next layer. This can result in memory access efficiency problems" [18]. To solve this problem, Li and al. in 2016 [18] proposed an implementation that manages the computation of all layers (i.e., from beginning to end) in a parallel manner. Another work, that was proposed by Li and al. [17], is a stochastic implementation of a two-layer restricted boltzmann machine classifier, which classifies the handwriting digit image recognition dataset, MNIST (i.e., we will give more details about this dataset in the last chapter), completely on a single FPGA. However the stochastic architecture proposed failed to provide an acceptable misclassification error compared to the software-based designs. Maria and al [19] suggested the use of stacked autoencoders for real-time objects recognition in power-constrained autonomous systems. They showed that within a limited number of nodes and layers

(i.e., in order to consume lower power), they were able to achieve not far from state-of-the-art on CIFAR-10 dataset (i.e., classification of images from 10 objects categories). In 2012 Moore and al. proposed a scalable, configurable real-time system named Bluehive [21] which is an FPGA architecture for very large-scale NNs (i.e., 64k spiking neurons per FPGA with 64M synapses). This work has incorporated fixed-point arithmetic to implement the computation of their neuron models. Another FPGA-based spiking network was proposed more recently (i.e., in 2014) by Neil and al. [22]. It is called Minitaur and it implements a spiking deep network which records 92% accuracy on MNIST.

As a conclusion, we can say that the usefulness of ANNs in real-time embedded applications can be improved if architectures for their hardware implementation on FPGAs can be customized in order to provide an attractive tradeoff between power consumption and performance.

3.4 Cryptography

In cryptography, there are many tasks that can be done. We are particularly interested in tasks that are related to the subfield of **secure multi-party computation**. Secure mutli-party computation scheme creates methods that allow parties to conduct a computation based on their inputs while keeping them private. The scenario can also be described as having parties that will give their private information to a trusted third party who will calculate functions on them and then share the results (i.e., data remain secret). The medical field is a good example where we can apply secure multi-party computation e.g., a group of biologists investigating about a genetic disease want to create statistics to help improving the diagnosing process, they have access to a database containing DNA patterns but information in the database is private (i.e., data of patients). In such case they can use secure multi-party computation to build statics without that information being disclosed. It can also be used in financial market : given two companies A and B that want to expand their market share in some region. Naturally, A and B do not want to compete against each other in the same region, so they need to have a strategy to know whether their regions overlap or not without giving away location information [6].

Here secure multi-party computation can be of a good use because it helps to solve the problem while maintaining the privacy of their locations.

In the context of third trusted party cryptography, there are several types of protocols that exist. In fact, we have the standard secure multi-party computation which was described above. Another interesting protocol is **zero-knowledge proof** called also zero knowledge protocol. The zero knowledge protocol is a method that allows one party, which is referred to as *the prover*, to prove to another party, which is referred to as *the verifier*, that a given statement is true without disclosing any information apart from the fact that the statement is indeed true. The verifier in such a scheme does not learn the secret information that the prover used for proving the statement and therefore he will not be able to prove it to anyone else. As an example of zero-knowledge protocol, one party can prove to another party that he has the password of a strongbox without having to reveal the password. A more recent protocol is **homomorphic encryption** where the encryption scheme allows one to compute a function on a plain-text by handling the ciphertext only. In simple terms, a homomorphic encryption scheme enables to perform computations on the ciphertext without decrypting it. Such scheme permits, among other things, the chaining together of different services without exposing the data (i.e., plein text) to each of those service. For examples, a chain of different services could calculate 1) the tax 2) the currency exchange rate 3) shipping, on an encrypted transaction without exposing the unencrypted data to each of those services [28].

The reader may ask how the previously described techniques are related to our approach. In fact, all those techniques perform computations only on boolean circuits. Therefore any function to be calculated on private data should be first transformed into a boolean circuit. Let's return to the aforementioned example of biologists that investigate about a genetic disease, they might want to train the database containing the DNA patterns so that at the end given a DNA of a new patient, they will be able to say if he or she has that genetic disease or not. Basically what is needed here is a boolean circuit classifier. Neural nets have been showing impressive results recently for the task of classification. However, it is very expensive to transform a neural network into a boolean circuit because there will be an astronomical number of gates assuming that the num-

ber of gates to multiply two n-bit integers is $\Omega(n^2)$ gates. The main advantage of our approach is that the classifier it builds is indeed a boolean circuit so any of the techniques mentioned above can be used with no harm. Once the classifier is trained, it can be applied on any private information to get a result.

CHAPITRE 4

PROPOSED ALGORITHM

In the last chapter, we set the framework of our approach : our classifier will be a boolean circuit that is a directed acyclic graph whose nodes are logic gates and leaves are bits chosen from instances of the dataset. In this chapter we will present the binary classification algorithm that first initializes the graph (i.e., finds truth tables of its gates) by a greedy local optimization, second adds hill climbing to optimize the choices of input dimension for all leaves (i.e., leaves' optimization) and finally performs nodes' optimizations to help some gates of the graph learn better features. Moreover we will discuss some theoretical concepts related to the choices we made in building the classifier. Without loss of generality, we will take the example of a dataset composed of images even though, as aforementioned, our algorithm could be applied on any type of data.

4.1 Data Preprocessing

In machine learning, an input example is an n-dimensional vector of features. Features can be categorical or numerical or a mix of both. For example, when representing images, the feature values might correspond to the RGB luminosity or grey level values of the pixels of an image (i.e., numerical). Most approaches of the literature consider the aforementioned features of an example for the learning process and then iterate over all examples. For our algorithm, regardless of the type of data, examples have to be transformed into binary vector representation. For categorical features, this can be done by a one-hot vector representation whereas for numerical features, the corresponding binary representation can be stored. In some cases, it might be useful to keep only the most significant bit(s) of the binary representation. At the end, all examples will be represented by a set of binary fixed-length vectors.

4.2 Greedy Initialization

Building the best classifier for the training set amounts to finding the simplest circuit capable of classifying correctly all its examples is obviously intractable as for a k input features there exists 2^{2^k} possible truth table (i.e., gates). We simplify this optimization problem several times, obtaining a greedy algorithm that produces a surprisingly good classifier which is extremely efficient to train. This algorithm will optimize gates (i.e., parameters) of the circuit locally in an attempt to reach an optimum and thus a good quality of classification. Before proceeding with the greedy algorithm, there are numerous hyperparameters that have to be set :

— Set the **arity** of all gates to be equal to k.
— Set the topology of our circuit to be a graph of **depth** d and of **levels** $[n_1, n_2, ..., n_d]$ where n_i is the number of nodes of level i beginning from the top of the graph. We always consider random connections between levels.

Note that if we fix the number of nodes of each level such that every non-root node has only a unique father then our classifier is a **full tree** of depth d.
Last, we let the leaves' inputs each be a random coordinate of the input space. Finally, we use greedy local optimization instead of global optimization. The greedy algorithm is quite simple. Each leaf is associated to a randomly chosen input coordinate (binary feature), and will receive the bit vector containing the value of this feature for all examples of the training set. To specify the circuit we have to specify the truth table of each of its gates. We choose these truth tables in a greedy way starting with the gates connected to the leaves (i.e., data bits) and climbing up the graph until the root gate, which outputs the classification decision.

So how do we choose the truth tables for the gates ? Every gate is greedily built ("trained") to output the correct class (target) as much as possible, given only k input bits it receives from the layer above : the task will be to find the logic gate that outputs the predicted feature that is closest to the target. Since it is a binary classification so

the target is unique (i.e., denoting the presence or absence of one of the two categories) and the same for each node of the graph. To accomplish this task, we proposed two schemes (i.e., the third one "random gates" was simply an intuition) : the optimisation based on probability finds the best gate, amongst all possible gates, that minimizes the classification error whereas the optimisation based on information gives the best possible gate in terms of information. We will detail both alternatives in the next subsections.

The reader might recall that the number of different gates of arity k is double exponential in k and might worry that this optimization would be intractable. That optimization can be simplified, as follows : In the case of a k-gate, we have 2^k possible inputs to worry about and we have to specify the answer of the gate on each of those, nothing less, but also nothing more. We can optimize a gate independently for each possible input vector, e.g. the best answer for a 4-gate on input $(0,0,0,0)$ can be computed independently from input $(0,0,0,1)$ and so on. This means for a k-gate we only need to compute 2^k values, not 2^{2^k}. This is reasonable to do when k is small (we mostly use gates of size 2 to 10 in this work).

We start this process of specifying gates one by one with the gates at the bottom of the graph, which take dimensions of training data (leaf nodes) as input. Once these gates' truth tables are known, we also know their output on each example, which we use to compute the second-level-gates' truth tables, and so on, moving up the graph until we learn all the gates. The output (i.e., feature) of the root node will be the final classification decision. As shown in figure 4.1 of 2-gate classifier, the algorithm learns the truth table T_1 by processing the two leaves x_1 and x_{100} then the corresponding gate outputs the Feature F_1, the same is done for x_{10} and x_{240} to output F_2. F_1 and F_2 will be input to learn the truth table T_5 and so on until the root node F_{11} which outputs the classification decision.

Until now, we haven't specified how we can compute a gate that will output a feature which is the closest to the target ? In fact, there are three approaches that refine the choice of the gate in the greedy initialization. In the following, we will discuss each of them and point out the one that was adopted and the reason for that.

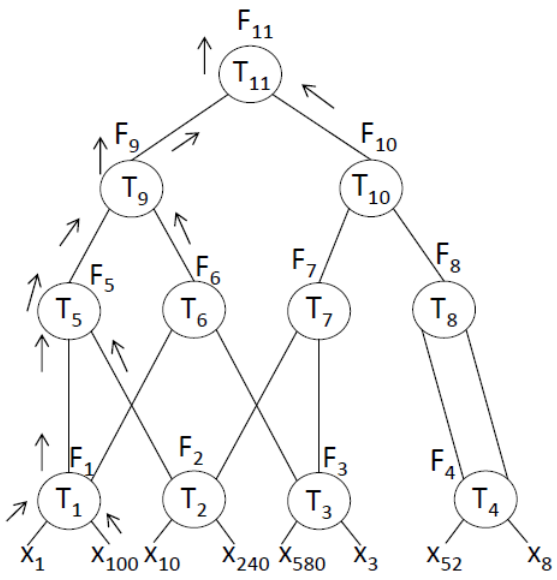

FIGURE 4.1 – Greedy algorithm

4.2.1 Maximizing probability

The idea of maximizing probability is very intuitive. We can compute the best output bit for a given input bit pattern simply by counting how many examples that produce this input for this gate belong to each category. If there are more such examples belonging to category 0, we chose that element of the truth table to be 0, otherwise, we set it to 1. This procedure will be repeated for every configuration of the input , e.g for a 4-gate, it will be done for $(0,0,0,0)$, $(0,0,0,1)$, $(0,0,1,0)$, $(0,0,1,1)$ and so on (i.e., $2^4 = 16$ different configurations).

Table 4.I shows an example of a binary classifier of 2-gates which takes a total of 12 training examples. B_i are input binary vectors and F_i are binary vectors representing features found by the classifier (i.e., F_1 the output of input B_1 and B_2, F_2 the output of input B_3 and B_4, F_3 the output of input F_1 and F_2). T is the target. For arity = 2, we have $2^2 = 4$ different configurations, $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$ .Taking as an example the configuration $(0,0)$ in B_1 B_2 we found that, according to the target T, category 0 comes more often than category 1 (3 vs. 1 as shown by table 4.II) so the value of the truth table for this configuration will be 0. Continuing similarly with the other configurations, the

value of the truth table is $[0,0,1,1]$ denoting 0 for $(0,0)$, 0 for $(0,1)$, 1 for $(1,0)$ and 1 for $(1,1)$. Then we can assign the value of the feature F_1 that is the result of the learning process from B_1 and B_2. As shown by Table 4.I the accuracy of the classifier improves as more and more features are learnt (i.e., first F_1 and F_2 then F_3) going from 75% to 83%. Maximizing probability is efficient and gives a good classifier but it not the best alternative. In fact, we obtained better final results by instead greedily maximizing information.

B1	B2	B3	B4	F1(B1, B2)	F2(B3, B4)	F3(F1, F2)	T
0	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0
0	1	0	0	0	0	0	0
0	0	0	1	0	1	1	0
0	1	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	1	1	1
1	0	1	0	1	0	1	1
1	1	0	1	1	1	1	1
0	1	1	1	0	1	1	1
1	0	0	1	1	1	1	1
1	1	0	0	1	0	1	1
uf [1]	uf	uf	uf	9 [2]	9 [3]	10 [4]	12 [5]
				75% [6]	75%	83%	
						83%	

TABLE 4.I – Example of maximizing probability

1. uf : unlearned feature
2. 9 : 9 out of 12 examples of the feature F1 are well classified
3. 9 : 9 out of 12 examples of the feature F2 are well classified
4. 10 : 10 out of 12 examples of the feature F3 are well classified
5. 12 : 12 examples of the target T
6. 75% : the percentage of well classified examples of the feature F1, other percentages of the table 4.I are expressing the same thing

T= 1	T= 0	B1	B2	F1
1	3	0	0	0
1	2	0	1	0
2	1	1	0	1
2	0	1	1	1

TABLE 4.II – Truth table of the gate F1

T= 1	T= 0	B3	B4	F2
1	2	0	0	0
2	1	0	1	1
1	3	1	0	0
2	0	1	1	1

TABLE 4.III – Truth table of the gate F2

T= 1	T= 0	F1	F2	F3
0	4	0	0	0
2	1	0	1	1
2	1	1	0	1
2	0	1	1	1

TABLE 4.IV – Truth table of the gate F3

4.2.2 Maximizing information

In this section we will argue that maximising the success probability is not in general the best thing to do when initializing a gate.

When working with probability, we notice the following facts : given two classifiers, the first one, shown by table 4.V, always outputs the category 0 when its input is 0 while it has a probability of 50% to output category 0 and 50% to output category 1 when its input is 1, the second one, shown by table 4.VI, is more symmetric having a probability of 80% and 20% to output category 0 and 1 respectively when its input is 0 and when its input is 1, it has a probability of 20% and 80% to output category 0 and 1 respectively. Calculating the classification error rate of each classifier, we found that the first classifier's error rate is 25% and the second one's error rate is 20% so we can say that the second classifier is better than the first one. This intuition is correct when those classifiers are used to take the final classification decision but in the case where several of those classifiers are combined (i.e., each gate in a graph can be considered to be a classifier, and so a classifier is a combination of several classifiers) to render a decision this intuition is wrong. For example, when combining three independent instances of each of those classifiers, we find the opposite to be true. First, let us give the details of calculation for each classifier. For the second one, we base our calculation on its

classification capability over category 0 : we can have two possibilities $0-0-0$ (i.e., first, second and third instances all output category 0) or two 0s and one 1 i.e., $0-0-1$, $0-1-0$ or $1-0-0$ so the average classification error rate for the first classifier over the three instances is $1-(0.8^3+0.8^2*0.2*C_3^1)=0.104$ where C_3^1 is a -1-combination over 3. For the first classifier, we base our calculation on its classification capability over category 1 : in contrast with the second classifier, no matter how many instances output category 0, if there is at least one instance that outputs category 1 then the input must be 1 so the only remaining possibility where the output is of category 0 is where first, second and third instances all output category 0 i.e., $0-0-0$ so the average classification error rate for the first classifier over the three instances is $1-(1-0.5*0.5^3)=0.0625$ admitting that both categories 0 and 1 have equal probability of 0.5. The reader may notice that even if the second classifier yields a better classification error rate than the first one, the first classifier gives a better classification error rate on average. This could be explained by the fact that the first classifier gives more information than the second classifier, i.e., when outputting category 1, it is impossible that the input is 0 it can only be 1 but for the second classifier there is always a probability of 20% that the input is 0.

I/O	0	1
0	1.0	0.0
1	0.5	0.5

TABLE 4.V – Classifier 1

I/O	0	1
0	0.8	0.2
1	0.2	0.8

TABLE 4.VI – Classifier 2

The observation, we made previously, leads us to choose information instead of probability in the greedy initialization. More specifically, we worked with the average entropy of the error rate. First, we count how many examples that produce this input for this gate belong to each category (i.e., as done when maximizing probability) and then we sort them in decreasing order according to one of the two categories (i.e. we chose category 1). Second we will try to divide input configurations according to the average entropy into two groups, one group will be assigned category 1 and the other one will be assigned category 0. In other terms, we will look for the best cut (i.e., the best position) where the average entropy calculated for category 1 is minimal then the first input

configurations will be given 1 in the truth table whereas the rest will be given 0. The algorithm describing steps for searching the best cut is shown below.

Table 4.VII and table 4.VIII illustrate an example of applying the algorithm for searching the best cut. Table 4.VIII represents the second step where the table is sorted in a decreasing order. Then using the formula described in the algorithm below, the best cut will be as follows : the first three lines belong to category 1 whereas the last one belongs to category 0, the cut is shown by a bold horizontal line in the table 4.VIII. Therefore the truth table of such gate will be $[1,0,1,1]$.

Let $tot1$ be the total of examples belonging to category 1, N the total of examples, c the number of configuration of the inputs (e.g., 4 for a 2-gate) and the function $H(x)$ is the function calculating the entropy of variable x.

input : A list l of 2 sublists : $l[0]$ contains the number of examples belonging to category 1 for each input configuration, $l[1]$ contains the number of examples belonging to category 0 for each input configuration.
output: The position of the best cut

```
1  list = [ ];
2  for i ← 0 to c − 1 do
3  |   v1 = v1 + l[0][i];
4  |   v2 = v2 + l[1][i];
5  |   tot = max(v1 + v2, 1);
6  |   tot = min(tot, N − 1);
7  |   val = (tot/N) ∗ H(v1/tot) + (N − tot)/N ∗ H((tot1 − v1)/(N − tot));
8  |   list.append(val);
9  end
10 pm = PosMin(list);
11 return pm;
```

Algorithm 1: Algorithm for finding the best cut

I/T	0	1
00	10	15
01	7	6
10	9	10
11	5	8

TABLE 4.VII – Input data occurrence counts

I/T	0	1
00	10	15
10	9	10
11	5	8
01	7	6

TABLE 4.VIII – Input data counts sorted according to rightmost column (target $T = 1$) with the best cut

4.2.3 Implementation

For implementation's efficiency we consider using what we call tensor product (i.e., introduced formally in 3.1) that will serve to produce markers. In fact, calculating the tensor product of the input of a gate will yield a set of binary vectors where each vector represents the marker of one configuration of the input : each vector denotes the presence of one configuration (e.g., for $k = 2$: configuration (0,0) or (0,1) or (1,0) or (1,1)) by having 1s in the positions of examples inputs having that configuration and 0s otherwise. Naturally the number of markers will be the number of possible configurations which is 2^k. All the binary vectors (i.e., markers) are mutually exclusive. For example for a 2-gate having vectors $F1$ and $F2$ as inputs, the tensor product yields the list of markers M that has $2^2 = 4$ elements, the first vector indicating the presence of the configuration $(0,0)$, the second $(0,1)$, the third $(1,0)$ and the last one $(1,1)$. That way, instead of iterating over the gates' inputs several times to count for every configuration how many belong to category 0 or 1, we can rely on markers and simply do a bitwise AND between a marker and the target to count how many times the target was one when the input was in the configuration corresponding to the marker. And for target equal to 0, we can do a bitwise AND between the marker and the bitwise negation of the target (we have mentioned in chapter 2 that we keep in memory the bitwise negation of all learned features and the target also). Below an example showing two vectors $F1$ and $F2$ and M reunites markers for different configurations, the first for the configuration (0,0), the second for (0,1), the third for (1,0) and the fourth for (1,1).

$$F1 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad F2 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad M = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

4.3 Leaf optimization

We have observed that the previous greedy algorithm's result is highly dependent on the choice of leaves' inputs. Repeating independently with different random choices of input dimensions is not very efficient. In this section we propose to use simple hill climbing to optimize the choice of input dimension for all leaves.

Again, the idea of this technique is quite simple. Choose one leaf uniformly at random and change it to take a different input dimension, also chosen uniformly at random. Then greedily retrain the gates affected by the change. If the new classifier is better, we keep the new value of the leaf otherwise we return to the previous value. Of course, this process must be repeated an appropriate number of times to get significant improvements. For some graphs, especially trees this can be done very efficiently because only the gates along the path from that leaf to the root need to be re-evaluated and since each node in a tree has a unique parent so the number of evaluations of gates is quite small. For example for a k array tree this value is the base-k logarithm of the number of leaves and even for very large trees this value remains small. For a k-gate graph the value is bigger especially if nodes of different levels tend to have a lot of parents. As shown in figures 4.2 and 4.3, the number of gates to be re-evaluated in the graph shown by figure 4.3 is greater than the one in the tree figure 4.2 (5 vs 4).

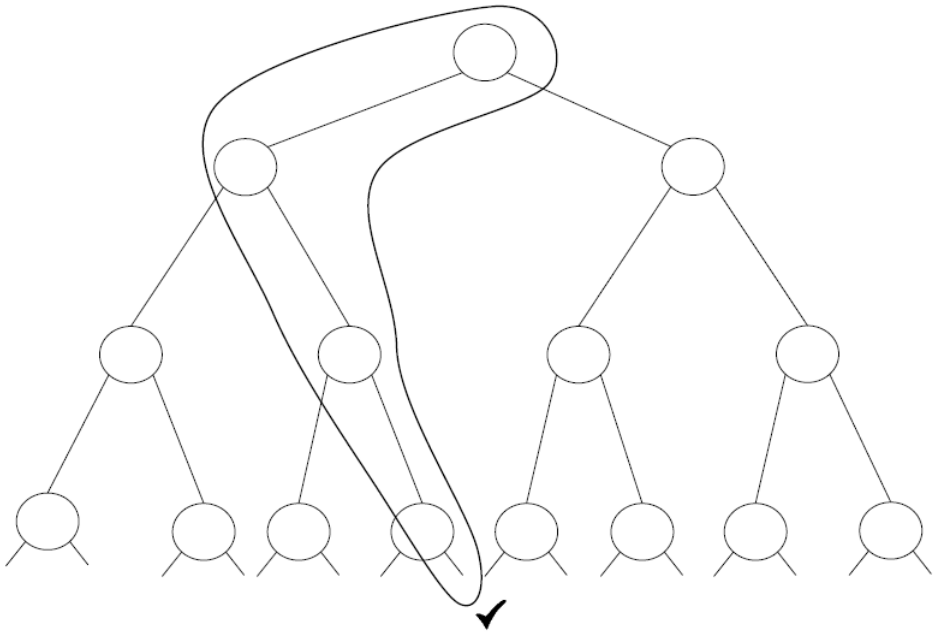

FIGURE 4.2 – Leaf optimization in a tree

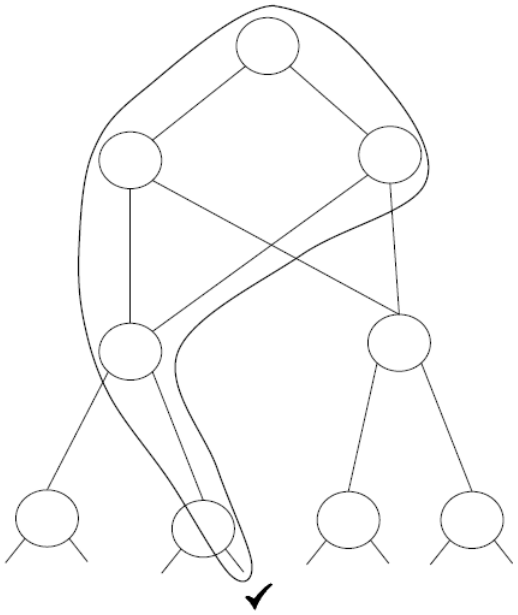

FIGURE 4.3 – Leaf optimization in a graph

As an alternative to switching a leaf with another input dimension (i.e., leaf) chosen uniformly at random, we can bias that choice according to a certain heuristic. The heu-

ristic deployed consists in building a matrix called correlation matrix where we calculate the distance (i.e., hamming distance) between two leaves (input dimensions) x and y and the correlation value assigned will be :

$$Correlation(x,y) = 2 \times |\frac{Nb\ Examples - Hamming\ distance(x,y)}{Nb\ Examples} - 0.5| \qquad (4.1)$$

For the optimization, we choose between the k^{th} first correlated leaves with the leaf to be switched.

4.4 Node optimization

Greedy initialization followed by leaf's optimization does not yield the best possible classifier. To improve the classifier's accuracy, we can further optimize the gates of the graph now taking into account the computation of subsequent nodes. Intuitively, we optimize a gate by choosing the best gate while all other gates remain unchanged, to maximize the classifier's quality. In node's optimization, a gate will be optimized to produce the most accurate final output classification at the root node. This technique is quite powerful and if not used carefully, can result in severe over-fitting. The idea is to choose a node then for every configuration of the input see if it is better to output (i.e., feature) a 0 or a 1 according to the root node's feature consequently generated. Most important steps of the node's optimization are as following :

1. Choose randomly a graph's level.
2. Choose randomly the node to be optimized.
3. Suppose that for a given configuration of the input (i.e., this step will be applied to every configuration) the gate outputs a 0 and then percolate to the top of the graph through evaluating (i.e., calculating the feature) all gates that are directly/indirectly (e.g., father of father) linked to the node to be optimized.
4. Repeat step 2 but suppose that the gate outputs a 1.

5. Compare the consequently generated root node's feature with the target. If the output error is smaller when the gate outputs 0 then the new value of the gate for the given configuration is 0 else 1.

That way the training set accuracy of the classifier always improves. Figures 4.4 and 4.5 show an example of a node's optimization. The node chosen to be optimized is N3. We will focus only on the configuration $(0,0)$ of the gate's input. Figure 4.4 shows the gain when putting 1 as the output of the gate for that configuration which is equal to 1. Figure 4.5 shows the gain for putting 0 which is equal to 0. We notice that the gain when putting 1 is greater than the one when putting 0 so for gate $G3$ it is better to keep 1 as the answer of the gate for the configuration $(0,0)$. Note that the gain is calculated by counting the number of 1s of the binary vector resulted when doing a bitwise AND between the root node's learned feature and the target.

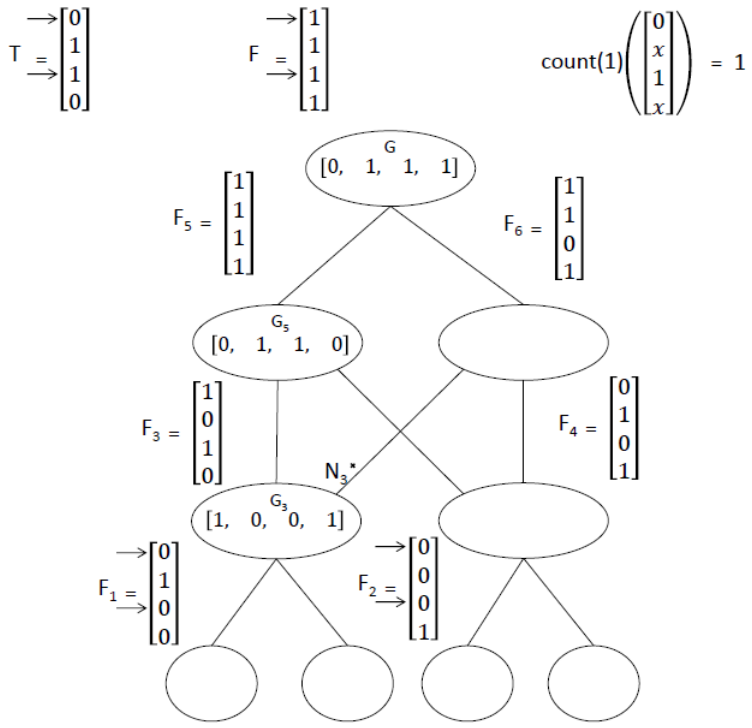

FIGURE 4.4 – Node's optimization counting ones

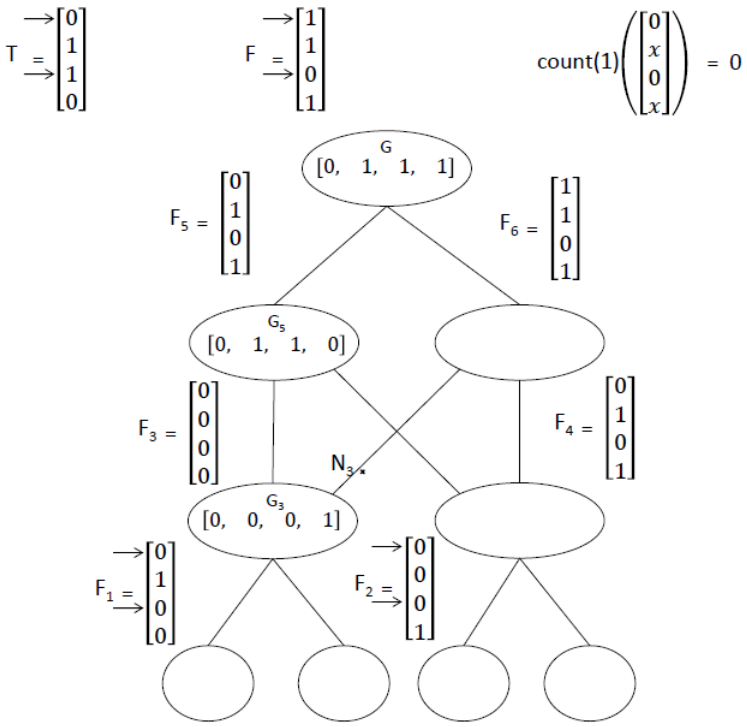

FIGURE 4.5 – Node's optimization counting zeros

The reader may have noticed that there is a considerable variance between levels (of a graph) in terms of number of nodes. This will affect the choice of nodes to be optimized noting that a level having for example two nodes will see each of its nodes optimized twice more than a level having four nodes if the the node's choice is made uniformly at random. To solve this problem, we proposed to give explicitly probabilities to choose levels of a graph for optimization. That way, the choice of a node is at random but with some distribution depending on the depth.

4.5 Regularization

We have seen in chapter 1 section 2.1.6 that one solution to the problem of overfitting is regularization. Many regularization techniques used in other frameworks, in particular the regularization technique that consists in adding a term in order to penalize the complexity of the model, are difficult to apply in our framework. Therefore the regularisation technique we have used is based on the injection of noise, similarly to what is done in

the "dropout" technique for training neural networks [27]. Adding noise to the input when optimizing a gate is a way to regularise. Fortunately, we do not have to go through the process of really injecting noise and we can do that analytically. More precisely, injecting noise to the input can be seen as flipping one or more bit(s) in a binary vector. Hence the noise will be accounted for by the number of flipped bits and the probability of flipping one bit. This can be simulated in our framework by considering **neighbours** in the gate optimization's process. In fact, when counting for one configuration of the input, instead of relying only on that configuration, we will henceforth take into account the h-neighbours. The h-neighbours are defined as the configurations that differ from the configuration in question in terms of hamming distance, e.g., considering the configuration of a 4-gate $(0,0,0,0)$, $(0,1,0,0)$ is a neighbour of hamming distance = 1, $(0,1,0,1)$ is a neighbour of hamming distance = 2. Setting the hamming distance to 3 for example means that we will consider 1-neighbours, 2-neighbours and 3-neighbours. Let ε be the probability of flipping a bit. Then by setting h and ε (i.e., which will be then additional hyperparameters for controlling regularization), we will control the quantity of noise we will be injecting into our model and therefore reduce overfitting. As an example, if we have a 4-gate and we are injecting an $h = 2$ noise with probability ε and let us consider the configuration $(0,0,0,0)$: we will have 4 neighbours of distance $= 1$, 6 neighbours of distance $= 2$ so we will assign to the configuration itself a factor of $(1-\varepsilon)^4$, the factor $\varepsilon \times (1-\varepsilon)^3 \times C_4^1$ for a 1-neighbour and $\varepsilon^2 \times (1-\varepsilon)^2 \times C_4^2$ for a 2-neighbour.

This technique of regularization was used directly in both the greedy and node's optimizations. However it is indirectly used in leaves' optimizations when re-evaluating gates after changing a leaf.

4.6 Other variants of the algorithm using ensembles of graphs

We investigated two variants of the algorithm which use classifier ensemble methods similar to boosting and bagging metaheuristics to better control bias and variance. The reader should be aware that we didn't apply the textbook version of boosting or bagging, to our algorithm, but rather we got inspired by the basic ideas of both of them and adapted

them to fit our framework.

4.6.1 2-stage combination of classifiers

We build a set of relatively small subgraphs (i.e., subgraphs of 3 to 4 levels, every subgraph is trained on different data since the leaves' inputs each is a random coordinate of the input space) by obviously greedy initializing them. Then we apply some leaf optimizations followed by node optimizations to every subgraph independently until the average classification error rate of all subgraphs no further improves. Building and optimizing subgraphs are efficient as the subgraphs are independent and thus their implementations can be distributed over many machines. Next, we will consider the subgraphs' resulting top-level output predictions (i.e., features of the root node) as inputs for a new graph that we called top graph. In contrast with what we described before, in this chapter, about standard graphs which take "raw" inputs (i.e., instances of the training set or what we referred to as unlearned features), the top graph takes as input learned features resulted by subgraphs previously built. After building top graph, we perform node optimizations until top graph's classification error rate no longer improves. Finally, we link subgraphs to top graph and we consider them to be a single graph and then apply node optimizations until its classification error rate no further enhances. An interesting issue to point out here is the difference between node optimizations performed in the first step (i.e., on subgraphs) and the ones performed in the last step (i.e., on the single big graph) : for the former, node optimizations are done in a subgraph-independent basis i.e., the optimization of a node in one subgraph is completely independent of a node's optimization in another subgraph as each optimization is done according to the root node of the corresponding subgraph. However, for the latter, the subgraphs are no longer considered to be independent but rather a part of the big graph and therefore every node will be optimized according to the root node of the big graph. Using boosting enabled us to produce a more powerful classifier (i.e., bias has decreased) and thus we obtained a significant improvement of the classification error rate compared to results obtained with the framework that is based on a single graph.

4.6.2 Bagging

For bagging, we rely on the same graph's building process as the 2-stage combination of graphs, i.e., we pass through the same steps as described in the previous subsection. However, the only difference resides in the training of the subgraphs (i.e., in the first step). In fact, we consider only a subset of examples of the training set sampled with replacement when training the subgraphs. Then the predictions of all resulting classifiers (i.e., subgraphs) are combined to train (i.e., greedy initialization, leaf and node optimization) a big classifier at the top just like in the variant of the 2-stage combination of graphs. This will help the model to less relying on all examples of the training set in the learning process. Consequently, the model less overfits and the classifier performs better on the testing set.

4.7 Connexion with other approaches : Multilayer Perceptron, Random Forest

In this section, we attempt to compare basic ideas of our approach with other approaches mainly multilayer perceptron and random forest. We will consider giving common aspects that other approaches share with ours and also different points where they diverge.

4.7.1 Connexion with Multilayer Perceptron

Multilayer perceptron (MLP) and our approach share some common aspects. In fact, they rely on the same architecture, i.e., the graph structure : both have nodes, referred to as units or neurons for MLP and gates for our approach. Fixing the number of hidden layers of MLP can be seen as fixing the number of levels in our approach, also specifying hidden units for each hidden layer is equivalent to fixing the number of nodes of each level of the graph in our approach. Both approaches try to extract non-linearities of the data in a layer-based framework by applying a non-linear transformation (i.e., a logic function for our approach and hyperbolic tangent or other non-linear function for MLP) to every node/neuron of a level/layer and repeating that for every level/layer. However, in MLP the graph is weighted and more particularly its learning algorithm aims at learning

the connexions' weights of the graph whereas in our approach the learning algorithm learns truth tables of different gates of the graph. Also, the computations carried out by a node are very different : a typical MLP neuron computes a real-valued weighted sum of its (usually real-valued) inputs followed by a real activation function, whereas a node in our approach computes an arbitrary boolean function of its k input bits. Another thing to point out is the number of connexions in the graph : while the number of connexions entering each neuron of the network (i.e., graph) can be very large (receiving connections from all neurons of the previous layer in a fully connected MLP), the number of connexions entering each node of the graph in our approach is not and is fixed to a number which we referred to, previously, as arity. MLP and our approach use different learning algorithms : as for the former, it applies stochastic gradient descent to perform backpropagation, for the latter it uses hill climbing to learn gates. With contrast to MLP that uses vectors of real numbers as the input data, our approach uses vectors of bits.

It is difficult to compare exactly the number of binary operations required for the evaluation of one gate in our algorithm compared to the evaluation of one neuron. That being said, a rough estimate still highlights the big advantage of our approach. For example, the number of logical operations for an arity 6-gate is around 64 operations and the multiplication of two 32 bit floats requires at least 5000 gates. A Relu neurone with 128 connections would require at least 640000 logical operations.

4.7.2 Connexion with Random Forest

It seems also natural to compare the 2-stage combination of classifiers version of our algorithm that uses trees (i.e., a very particular case of graphs) to random forests (RF). As common points between our approach (i.e., using only trees) and random forests, we can point out that both are based on a set of subtrees that learn from a randomly chosen subset of examples. Also, both approaches learn by considering features aka., attributes of different input examples. On the other hand, there are many different aspects that separate the two approaches : first, leaves for RF represent class labels whereas for our approach they represent input dimensions. Second, a node in a RF tree partitions the examples based on a single input attribute but for our approach they represent logic gates

that compute a boolean function of k features from the layer below. For RF, the branches of the tree represent conjunctions of values that an attribute can take whereas in our approach the tree has unlabelled branches. Moreover, while the learning algorithm in RF learns the tree from top (i.e., the root node) to bottom, the learning algorithm of our approach learns the tree from bottom to top. More particularly, the learning algorithm in RF aims at splitting the training set into subsets based on an attribute value test whereas in our approach it consists in finding the best gate according to an objective function. Although, both approaches can use entropy in the building process of the trees, they use it in a very different way : for random forests when building a tree, the feature (i.e., attribute) selected is the one that has the biggest value of information gain, information gain expresses how much entropy about data has been reduced when selecting that feature. However, for our approach, when building the tree, the logic gate chosen is the one that minimizes the average entropy of the classification error rate.

CHAPITRE 5

EXPERIMENTAL RESULTS

In this chapter, we will present results obtained by the classifier generated by our approach. We will begin by describing the two datasets (i.e., CONVEX and MNIST) we worked with and specify the technique used for hyperparameters selection. Next, we will report results given by the classifier showing the effect of 2-stage combination of classifiers, optimizations and bagging for the improvement of its quality. Finally, we will give a summary of our observations on the behaviour of the algorithm : we will discuss strong and weak points of the approach and compare it to the SVM technique.

5.1 Methodology

We experimented our approach with 2 datasets : **MNIST**, **CONVEX**. MNIST dataset [16] is a database of handwritten digits, it has 60000 images. All images are grayscale of size 28×28 pixels, falling into 10 classes corresponding to the 10 digits. CONVEX dataset [7] is a database of images showing convex and non-convex sets (i.e., regions). It has 58000 images. All images, just like MNIST, are grayscale of size 28×28 pixels, falling into 2 classes corresponding to the 2 different sets. The reader should be aware that we used for our experiments a binarized version of MNIST and CONVEX.

We have chosen to work with MNIST-01234-56789 (i.e., we train on 9000, we validate on 1000 then we train on 10000 and we test on 1000) where the main task of the model will be to recognize digit set $\{0,1,2,3,4\}$ against $\{5,6,7,8,9\}$. It is relatively easy to learn a model that generalizes well on test data for the MNIST dataset, nevertheless we have taken a smaller training set because a smaller training set will make it easier to learn the training set but more difficult to get a good generalization on test data. For CONVEX dataset, we train on 7500, we validate on 500 then we train on 8000 and we test on 4000. It is known to be hard to learn a model that generalizes well on test data for the dataset CONVEX but preliminary experiments were showing that good results

can be obtained with our algorithm.

The evaluation of our approach was accomplished using the test set. As it will be discussed in the next section, optimization may cause overfitting of the training set. For that reason, we might want to stop optimization once reaching such point. Validation set was used to select optimal number of optimization steps.

We tested different variations of our approach including simple greedy algorithm, greedy algorithm with leaf and node optimization and the 2-stage combination of classifiers and bagging versions of the algorithm. We compared our results with support vector machine with radial basis function kernel (RBF-SVM). For RBF-SVM, best values of hyperparameters and best classifier's quality were selected/produced using respectively the aforementioned validation sets of MNIST and CONVEX. For hyperparameters exploration for RBF-SVM, we used the grid search technique. Best classification error rates obtained on MNIST and CONVEX by different variations of our algorithm as well as RBF-SVM are reported in table 5.I.

	RBF-SVM	Greedy with 2-stage combination of classifiers	Greedy with 2-stage combination of classifiers and optimizations	Greedy with bagging	Greedy with bagging and optimizations
MNIST	2.4%	15%	13.4%	19.7%	9.1%
CONVEX	20.68%	20.4%	19.25%	20.6%	16.3%

TABLE 5.I – Test set classification error on binarized MNIST (0,1,2,3,4 versus 5,6,7,8,9) and CONVEX

5.2 Summary of our experimental observations

We ran many experiments to understand the effects of the different hyperparameters of our algorithm. We will present our observations in the next paragraph. One of the problem we have faced is the large number of hyperparameters. For that reason, we have chosen to concentrate on the effect of a subset of them and leave the others for future exploration (i.e., we have fixed them to specific values when experimenting). Nonetheless, we will highlight their potential effects in the following discussion.

Preliminary results showed that 2-stage combination of classifiers variant of the algorithm (i.e., when the classifier produced is composed by two levels of graphs) always helps giving better results than the ones obtained by applying the primary version of the algorithm in which the classifier is a simple graph. Therefore, in all experiments presented in the next sections, we have been using 2-stage combination of classifiers variant of the algorithm with other optimization techniques depending of the version of the algorithm explored.

Watching the behaviour of the algorithm through various experiments, we have made the following observations :

— MNIST is hard for our algorithm (with optimisation). We cannot beat RBF-SVM.

— CONVEX is easy for our algorithm (with or without optimisation). We can beat RBF-SVM.

— Increasing arity significantly increases the running time and the complexity of the model. It can lead to overfitting. Arity 4 and 6 are reasonable choices in general. In fact, Arity 6 is good for FPGAs as such circuits are composed by 6-input gates. Whereas Arity 4 is a better fit for cryptography as it achieves a trade-off between complexity and number of gates.

— Arity 8 is not efficient but can be good. Arity 10 is too much overfitting. Beyond arity 10 the running time and model's complexity are clearly high.

— The greedy algorithm is better than RBF-SVM for the CONVEX dataset.

— For the greedy algorithm, graphs are better than trees in learning. In fact, taking a graph means that nodes have many parents and thus the running time will increase proportionally to the number of parents. This is will affect more optimizations (i.e., leaf and node optimizations) because they are performed several times in our algorithm, the greedy initialization is done only once when constructing the graph at the beginning. When optimization is done, the gain in speed of using trees is essential.

— For the greedy algorithm, the bigger the graph is (i.e., number of levels and number of nodes in every level) the better is the classification accuracy. In other words, having a larger graph always helps both in learning the model and in

testing on unseen data. However, at some point the quality improvement is not proportional to the increasing number of computations when building the graph.

— With the greedy algorithm, bagging seems to be very useful on CONVEX but not on MNIST.

— Leaf optimization helps to improve the classification error rate and hardly overfits. Effects of the use of correlation matrix heuristic on leaf optimization is left to future experiments.

— Node optimization enhances the classifier's quality as well. However, it rapidly causes overfitting.

— Regularization by analytically injecting noise decreases significantly overfitting but it can also cause underfitting. It is a tricky parameter to adjust.

— We think that fixing different values of regularization hyperparameters for different levels of the graph could lead to better results. However, due to lack of time, we could not explore that side of the algorithm on time to include it in this document.

— Bagging helps notably especially when a significant optimization (i.e. leaf and node optimization) is done.

— The algorithm is probabilistic in all of its aspects (i.e., greedy initialization, leaf and node choices for optimization, bagging selection of subsets of examples). The large number of random choices makes the variance of the accuracy of the algorithm relatively small and repeated experiments yield similar results.

CHAPITRE 6

CONCLUSION

Through this project, we have brought a novel approach for supervised binary classification. In this approach, the classifier is a boolean circuit thus it is suitable for fields including FPGAs and secure multiparty computation. Experiments we reported have shown impressive results especially for the CONVEX dataset. Unfortunately, we haven't observed similar notable results for the MNIST dataset as we didn't manage to beat RBF-SVM. Leaf optimization and bagging have proven to bring significant improvements to the classification error rate. Node optimization is a powerful tool but sometimes it causes overfitting. To reduce overfitting, we used regularization whether in the greedy initialization or/and in the node and leaf optimizations but regularization hyperparameters are tricky, they may cause underfitting if not well-selected. One of the biggest problems we faced was hyperparameters optimal values selection as we count numerous ones in our algorithm. In the future, we will put more emphasis in exploring the effects of hyperparameters that we hardly varied in our experiments such as regularization for different levels of the graph or the use of correlation matrix in the leaf optimization. We think also that we should conduct experiments on other datasets to highlight varieties of problems or structures that can be learned both accurately and efficiently by our approach. Once we complete exploring strength and weakness of our approach, our intention is to publish an article reporting all our work.

It is worth noting that we elaborated a mutlicategory classification version of our approach. However, due to lack of time we were not able to test it through experiments. We decided to focus rather on the binary classification version in order to build a solid baseline and thus be more pragmatic when exploring the multicategory version later on.

The approach and the main algorithm (i.e., greedy initialization and leaf optimization) as well as preliminary results were completed by Prof. Alain Tapp before I started working on the project. Then we collaborated together to accomplish the node optimization, boosting and bagging versions of the algorithm (and the correlation matrix

heuristic) and the multicategory version of the algorithm.

BIBLIOGRAPHIE

[1] Riyad Alshammari and A Nur Zincir-Heywood. Machine learning based encrypted traffic classification : Identifying ssh and skype. *CISDA*, 9 :289–296, 2009.

[2] Zachary K Baker and Viktor K Prasanna. Efficient hardware data mining with the apriori algorithm on fpgas. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 3–12. IEEE, 2005.

[3] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. *IACR Cryptology ePrint Archive*, 2014 : 331, 2014.

[4] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3) :273–297, 1995.

[5] Reinhard Diestel. Graph theory (3rd ed'n). 2005.

[6] Wenliang Du and Mikhail J Atallah. Secure multi-party computation problems and their applications : a review and open problems. In *Proceedings of the 2001 workshop on New security paradigms*, pages 13–22. ACM, 2001.

[7] Dumitru Erhan. Recognition of convex sets, 2007. URL `http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/ConvexNonConvex`. [Online ; accessed 22-July-2016].

[8] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. Cnp : An fpga-based processor for convolutional networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 32–37. IEEE, 2009.

[9] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 682–687, 2014.

[10] Thore Graepel, Kristin Lauter, and Michael Naehrig. Ml confidential : Machine learning on encrypted data. In *International Conference on Information Security and Cryptology*, pages 1–21. Springer, 2012.

[11] Shinichiro Haruyama. Fpga in the software radio. *IEEE communications Magazine*, 109, 1999.

[12] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola. Kernel methods in machine learning. *The annals of statistics*, pages 1171–1220, 2008.

[13] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5) :359–366, 1989.

[14] National Instruments. Introduction to fpga technology : Top 5 benefits, 2012. URL `http://www.ni.com/white-paper/6984/en/#`. [Online ; accessed 14-June-2016].

[15] Seunghun Jin, Junguk Cho, Xuan Dai Pham, Kyoung Mu Lee, S-K Park, Munsang Kim, and Jae Wook Jeon. Fpga design and implementation of a real-time stereo vision system. *Circuits and Systems for Video Technology, IEEE Transactions on*, 20(1) :15–26, 2010.

[16] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11) :2278–2324, 1998.

[17] Bingzhe Li, M Hassan Najafi, and David J Lilja. An fpga implementation of a restricted boltzmann machine classifier using stochastic bit streams. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 68–69. IEEE, 2015.

[18] Ning Li, Shunpei Takaki, Yoichi Tomiokay, and Hitoshi Kitazawa. A multistage dataflow implementation of a deep convolutional neural network based on fpga for high-speed object recognition. In *2016 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI)*, pages 165–168. IEEE, 2016.

[19] Joao Maria, Joao Amaro, Gabriel Falcao, and Luís A Alexandre. Stacked autoencoders using low-power accelerated architectures for object recognition in autonomous systems. *Neural Processing Letters*, 43(2) :445–458, 2016.

[20] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2012.

[21] Simon W Moore, Paul J Fox, Steven JT Marsh, A Theodore Markettos, and Alan Mujumdar. Bluehive-a field-programable custom computing machine for extreme-scale real-time neural network simulation. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 133–140. IEEE, 2012.

[22] Daniel Neil and Shih-Chii Liu. Minitaur, an event-driven fpga-based spiking network accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12) :2621–2628, 2014.

[23] David Opitz and Richard Maclin. Popular ensemble methods : An empirical study. *Journal of Artificial Intelligence Research*, pages 169–198, 1999.

[24] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2, 2015.

[25] Phi-Hung Pham, Darko Jelaca, Clement Farabet, Berin Martini, Yann LeCun, and Eugenio Culurciello. Neuflow : Dataflow vision processing system-on-a-chip. In *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1044–1047. IEEE, 2012.

[26] sickit-learn developers. Ensemble methods, 2014. URL `http://scikit-learn.org/stable/modules/ensemble.html`. [Online ; accessed 18-August-2016].

[27] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout : a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1) :1929–1958, 2014.

[28] Craig Stuntz. What is homomorphic encryption, and why should i care ?, 2010. URL `http://community.embarcadero.com/blogs/entry/what-is-homomorphic-encryption-and-why-should-i-care-38566`. [Online ; accessed 13-July-2016].

[29] Mike Thompson. Fpgas accelerate time to market for industrial designs. *EE Times*, 2004.

[30] Richard J Trudeau. Introduction to graph theory (corrected, enlarged republication. ed.), 1993.

[31] Wikipedia. Field-programmable gate array — wikipedia, the free encyclopedia, 2016. URL `https://en.wikipedia.org/w/index.php?title=Field-programmable_gate_array&oldid=722326962`. [Online ; accessed 14-June-2016].

[32] Wikipedia. Support vector machine — wikipedia, the free encyclopedia, 2016. URL `https://en.wikipedia.org/w/index.php?title=Support_vector_machine&oldid=727504473`. [Online ; accessed 29-June-2016].

[33] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.