JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Bernhard Gstrein, BSc
k01515230

Submitted at
Institute for Symbolic Artificial Intelligence

Supervisor
Univ.-Prof. Dr. **Martina Seidl**

Co-Supervisor
Univ.-Prof. Dr. **Armin Biere**

June 2022

Tuning the Learning of Circuit-Based Classifiers

Master Thesis
to obtain the academic degree of
Master of Science
in the Master's Program
Artificial Intelligence

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

Bernhard Gstrein

Linz, June 2022

Acknowledgement

I would like to thank Martina Seidl and Armin Biere for providing the very interesting topic of circuit-based classifiers and for useful discussion and guidance throughout. I am indebted to JKU Linz for providing great courses that gave me the necessary knowledge in AI on which I could build this thesis. Lastly, thanks and appreciation to all my family members and friends that supported me during my journey through the Master's program and thesis.

Abstract

Neural networks have proven to work remarkably well and are prevalent in our everyday lives. Two major concerns, however, are computational cost during training and inference as well as the model size. Circuit-based models that work with binary values therefore seem promising to address these concerns. We investigate the possibility of building binary classifiers by looking at two methods: LUTs and AIGs. We discuss the practicality of building such models, how they can be improved and compare them to traditional ML models in terms of accuracy and size. Our findings are that both LUTs and AIGs have potential as binary classifiers: LUTs score with their solid learning algorithm and AIGs with their tiny model size.

Kurzfassung

Neuronale Netze haben sich als bemerkenswert leistungsfähig erwiesen und sind in unserem Alltag weit verbreitet. Zwei große Probleme sind jedoch die Rechenkosten beim Trainieren und bei der Inferenz, sowie die Modellgröße. Schaltungsbasierte Modelle, die mit binären Werten arbeiten, scheinen daher vielversprechend. Wir untersuchen die Möglichkeit, binäre Klassifikatoren zu erstellen, indem wir uns zwei Methoden ansehen: LUTs und AIGs. Wir erörtern die Praktikabilität der Erstellung solcher Modelle, wie sie verbessert werden können und vergleichen sie mit traditionellen ML-Modellen in Bezug auf Genauigkeit und Größe. Unsere Feststellungen sind, dass sowohl LUTs als auch AIGs Potenzial als binäre Klassifikatoren haben, wobei LUTs mit ihrem soliden Lernalgorithmus und AIGs mit ihrer winzigen Modellgröße punkten.

Contents

List of Figures

Chapter 1

Introduction

1.1. Motivation

Neural networks trained using gradient descent work remarkably well. Small neural networks are useful for many tasks, but it is the large neural networks that perform complex tasks, such as translation, object recognition or autonomous driving. The more parameters, the more compute during training and inference is required, and the environmental impact is a concern [1].

A cumbersome aspect of neural network-powered AI is that many models are stored on a remote server and data has to be sent over the internet. The requirement of a constant and (in most cases) fast internet connection is a serious inhibitor to applying neural networks in practice. Without being able to transmit all of its input data to a server, devices with small computing power and little storage, such as autonomous lawn mowers, intelligent cameras, smart sensors, etc. cannot be used in conjunction with big neural networks.

1.2. Approach

We know that a neural network basically is just a sequence of matrix operations; our input is transformed, and we obtain a quantity of our interest which could be a probability, a number, a synthesized image, etc. We also know that a computer works with binary values, i.e., 0s and 1s. In the end, a representation of a neural network is a sequence of 0s and 1s, so is the input to the neural network and the output, which raises the question if we can *directly* work with binary values, skipping layers of abstraction. Perhaps such a scheme would be more environmentally friendly since it does not require floating point operations.

A small binary model, i.e., circuit, that requires little compute and disk space might be a good step towards making AI more prevalent and reducing costs. Extensive research already exists to make existing neural architectures smaller [2]. As opposed to deploying them on the “cloud”, small models (which also includes binary architectures) can then be applied at “edge” devices. Most existing approaches to reduce model size into binary start with regular models and gradually transfer them to binary. We will already start at binary.

1.3. Outline

Our questions of interest are summarized in the following list:

1. Is it possible to build a predictive system that works entirely in binary?
2. Is it practical to build a predictive system working in binary? Concretely, does a predictive system working with binary values...
    a) ...have a good performance in terms of accuracy?
    b) ...take up less space on disk?
    c) ...have a good performance in terms of computational efficiency?
3. How can these systems be improved?

We address (1) by looking at 2 schemes to build network-like architectures, where the former is described in a paper [3] and the latter we invent ourselves. We evaluate training and testing accuracies of each architecture (2a), calculate the size they take up on disk (2b) and discuss the computational power required to deploy these models (2c). The sub-items of (2) we discuss to our best knowledge, but might only give a hint what is possible given there is room for improvement (3).

Lastly, we would like to already give away what we achieved in this thesis. Figure 1.1 visualizes testing accuracies and size for different models. The more a point lies to the top and to the left, the better. The meaning of the model names we will introduce throughout this thesis. We took an existing approach of binary model from [3], visible as “5@1024 8-LUT network” in Figure 1.1, and were able to improve accuracy and reduce size, visible as “(5@54)×1024 2-LUT advanced ensemble” in Figure 1.1. In the lower-left corner we can see another binary model that we built, the “AIG”. It is the weakest model in terms of accuracy in Figure 1.1, however, it is tiny (just 68 Bytes) and easily translatable to a circuit on hardware, making this model a very good candidate for implementing on small electronic devices, also called “edge devices”.

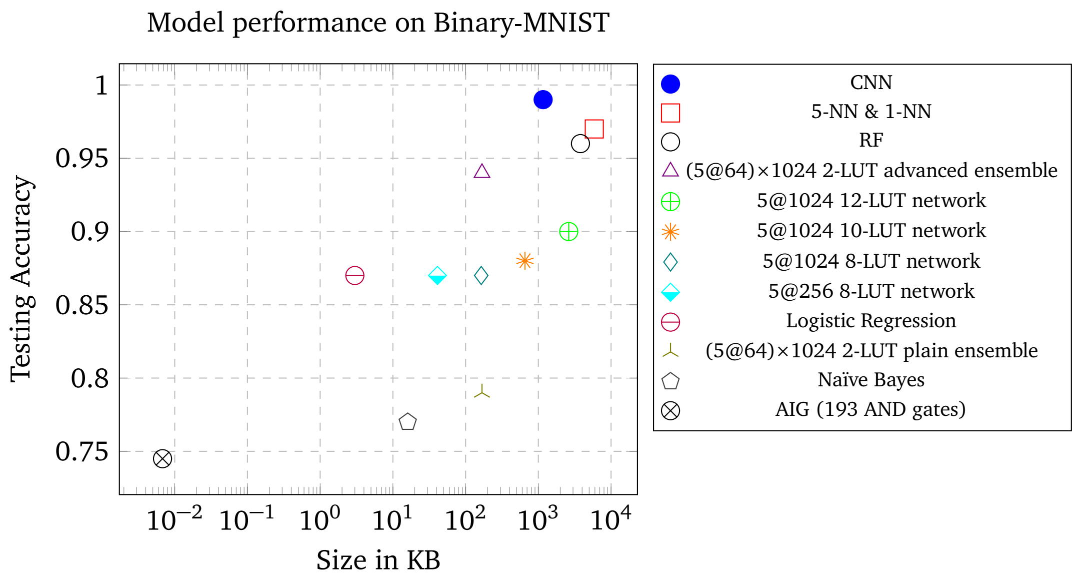


Figure 1.1.: Testing accuracy on the dataset "Binary-MNIST" and size for various models. The x-axis that represents the size has a logarithmic scale. The underlying data that was used to generate this plot can be seen in Table 6.1. Our main contributions are a (5@64)×1024 2-LUT advanced ensemble, a binary model with improved accuracy and reduced sized compared to existing approach 5@1024 12-LUT network and an AIG as classifier, a model that is tiny and fast, albeit with lower accuracy.

Chapter 2

Preliminaries

Throughout this thesis, we will mention neural networks and many other machine learning terminology that the reader might be unfamiliar with. In this section, we will give an overview of what NN are and how they are constructed. If the reader already has basic knowledge of what NN are and how they are learned, this section can be safely skipped.

In general, a neural network takes a d-dimensional vector $\boldsymbol{x}$ (we denote vectors with variables in bold) as input. As output, we obtain either a single number or another vector. The output is the *prediction* of the neural network given the input $\boldsymbol{x}$. A few examples of predicted quantities given some input could be the price of a house given various information about it (e.g., age, size, location, etc.), the probability of a dark skin area being a melanoma given a picture of it or a Japanese translation of a German sentence. A neural network arrives at the prediction by *transforming* the input. Concretely, matrix operations are being performed.

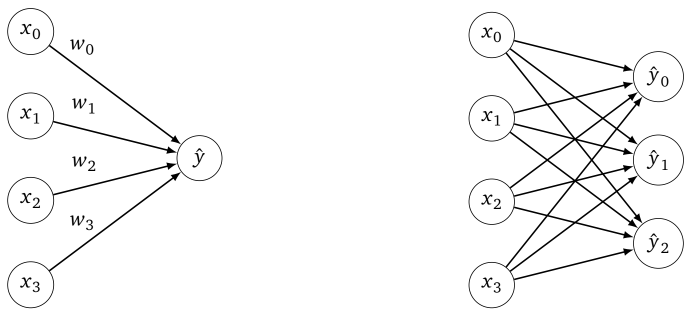
As an example, consider input $\boldsymbol{x} \in \mathbb{R}^{4\times 1}$ and “weights” $\boldsymbol{w} \in \mathbb{R}^{4\times 1}$. The operation $\boldsymbol{w}^\top \cdot \boldsymbol{x} = \hat{y}$ yields a single number, the predicted quantity. The symbol $\cdot$ denotes the matrix product and the symbol ˆ denotes any variable below it is a prediction. Writing it out, we obtain

$$x_0 w_0 + x_1 w_1 + x_2 w_2 + x_3 w_3 = \hat{y}. \tag{2.1}$$

If the weights have more than one column, e.g., if $\boldsymbol{W} \in \mathbb{R}^{4\times 3}$, then the operation $\boldsymbol{W}^\top \cdot \boldsymbol{x} = \hat{\boldsymbol{y}}$ yields a three-dimensional vector:

$$\begin{aligned} x_0 w_{00} + x_1 w_{10} + x_2 w_{20} + x_3 w_{30} &= \hat{y}_0, \\ x_0 w_{01} + x_1 w_{11} + x_2 w_{21} + x_3 w_{31} &= \hat{y}_1, \\ x_0 w_{02} + x_1 w_{12} + x_2 w_{22} + x_3 w_{32} &= \hat{y}_2. \end{aligned} \tag{2.2}$$

We can visualize the aforementioned matrix calculations using nodes and arrows, as shown in Figure 2.1. The nodes represent individual entries of the respective vectors and the arrows multiplication with the weights. In Figure 2.1b, we omit the weights on the arrows for better visibility.



(a) Scalar output. **(b)** Vectorial output.

Figure 2.1.: Matrix operations from Equation 2.1 in (a) and from Equation 2.2 in (b) visualized. Each node is a scalar and an arrow protruding from it denotes the scalar is multiplied with a weight.

The calculations performed in Equation 2.1 and Equation 2.2 deliver a predicted quantity. However, neural networks are better described by

$$\boldsymbol{W}^{[3]\top} \cdot \underbrace{h\big(\boldsymbol{W}^{[2]\top} \cdot \underbrace{h\big(\boldsymbol{W}^{[1]\top} \cdot \boldsymbol{x}\big)}_{\boldsymbol{a}^{[1]}}\big)}_{\boldsymbol{a}^{[2]}} = \hat{y}, \tag{2.3}$$

where $\boldsymbol{x} \in \mathbb{R}^{4\times1}$, $\boldsymbol{W}^{[1]} \in \mathbb{R}^{4\times5}$, $\boldsymbol{W}^{[2]} \in \mathbb{R}^{5\times3}$, $\boldsymbol{W}^{[3]} \in \mathbb{R}^{3\times1}$ and h is a point-wise non-linear function. The intermediate results $\boldsymbol{a}^{[1]}$ and $\boldsymbol{a}^{[2]}$ are commonly called "activations". Numbers in brackets are used to differentiate between variables and otherwise have no effect mathematically. Visualizing Equation 2.3, we obtain a graph structure shown in Figure 2.2. We can see that the activations (i.e., intermediate results) are also represented using nodes. An array of nodes forming one activation vector we refer to as "hidden layer".

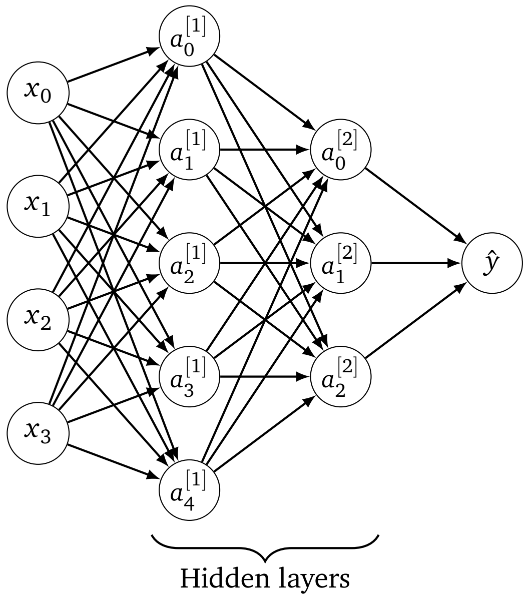


Figure 2.2.: Equation 2.3 visualized, a small neural network. The nodes between input and output represent intermediate results and are called "activations". An array of nodes forming one activation vector we refer to as "hidden layer".

When we refer to the "architecture" of a neural network, we mean the number of weight matrices, shape of weight matrices (i.e., how many rows and columns they have), types of activation functions and so on. The neural network depicted in Figure 2.2 is small and simple, but enough for this introduction. A good overview of different neural architectures can be found in [4]. In case the reader wants to refresh their knowledge about neural networks, [5] provides a good resource.

Unless an algorithm is utilized that automatically searches for a neural architecture [6], a neural architecture has to be chosen by humans. Apart from meaningful initialization [7], the "parameters" of a neural network (i.e., the individual values of the weight matrices) are not manually set, but obtained using an algorithm. The idea of this algorithm is to learn from a labelled dataset.

As an analogy, imagine a grown-up human has never seen an animal before in their entire life. Then we give that person 100 Polaroids depicting cats with the word "cat" written on the white part under the picture. In machine learning terminology, we would call the picture a "training example" and the word "cat" written under it on the Polaroid a "label". In the same fashion, we give 100 Polaroids depicting dogs to that person. The entirety of cat and dog Polaroids is the labelled dataset (unlabelled datasets are used in *unsupervised machine learning* [8]).

If we then show that person a yet unseen picture of either a cat or dog, then they will surely be able to tell which animal it is, meaning the person has *learned* to differentiate between cats and dogs. We can do the same with a neural network. We show it a certain number of d-dimensional vectors with entries ranging from 0-255 (images) and every image has a label to it, 0 for cat and 1 for dog. Given enough data, through a learning algorithm the network gets proper values in its weights, such that cat pictures fed to it should produce output 0 and dog pictures output 1.

Showing the dataset to the network and adjusting the weights accordingly we call "learning" or "fitting". If we feed a yet unseen picture to the network, it should hopefully still work the same. If the network performs badly on yet unseen pictures because its architecture is too simple, we call it "underfitting". Conversely, if it performs badly because it is complex enough to only remember peculiarities in the dataset and does not grasp the general concept, we call it "overfitting".

In order to learn the parameters of a neural network, gradient descent with backpropagation [9] at its core is almost exclusively used nowadays. Backpropagation provides the gradients of the weights with respect to some *loss function*. A loss function compares the prediction of the network and returns a high value if the prediction is bad and a low value if the prediction is good. We want to minimize the loss, thus the gradients of it with respect to the neural network weights provide us the direction where we want to go.

We update the weights by subtracting the current gradients, scaled by a parameter called "learning rate", from it. This procedure is called "gradient descent" and can be used with many loss functions for many problems which includes models that are not neural networks. Nowadays, gradient descent is refined by adding many heuristics to it, for example choosing the optimal step size per parameter, the concept of "momentum", "batching" and many more. An explanation of the terms mentioned in this paragraph and an overview of gradient descent optimization algorithms can be found in [10].

Chapter 3

Lookup Tables (LUTs)

We now introduce a scheme to build classifiers that works entirely with binary values described in [3]. The basic idea is to build a "lookup table" (LUT) based on the training data. Many lookup tables (LUTs) are created and stacked, reminding us of a neural network. The paper shows that this architecture not only generalizes on yet unseen data, but also shares some properties of neural networks. We will first start with the definition of a single LUT and then work our way up to a network of LUTs. We then discuss experimental results of [3], where we re-create some of them.

3.1. Single LUTs

We start with a simple example. The task is to classify a 2-bit input $\boldsymbol{x}$ into either 0 or 1. Since $\boldsymbol{x}$ has two bits, $\boldsymbol{x} \in \{00, 01, 10, 11\}$. Given training data $(\boldsymbol{X}, \boldsymbol{y})$, where $\boldsymbol{X}$ has N rows and two columns and $\boldsymbol{y}$ has N rows and one column, we are given the task of creating a LUT f:

$$f(\boldsymbol{x}) = \begin{cases} ? & \text{if} \quad \boldsymbol{x} = 00, \\ ? & \text{if} \quad \boldsymbol{x} = 01, \\ ? & \text{if} \quad \boldsymbol{x} = 10, \\ ? & \text{if} \quad \boldsymbol{x} = 11. \end{cases} \tag{3.1}$$

If the training data has 90 examples of $\boldsymbol{x} = 01$ with the label $y = 1$, and 10 examples of $\boldsymbol{x} = 01$ with the label $y = 0$, we classify any new unseen $\boldsymbol{x}$ of value 01 with label 1. Our LUT f now looks like this:

$$f(\boldsymbol{x}) = \begin{cases} ? & \text{if} \quad \boldsymbol{x} = 00, \\ 1 & \text{if} \quad \boldsymbol{x} = 01, \\ ? & \text{if} \quad \boldsymbol{x} = 10, \\ ? & \text{if} \quad \boldsymbol{x} = 11. \end{cases} \tag{3.2}$$

As we have seen, the basic idea is to count occurrences: for any bit pattern, we count how many times it has the label 0 and how many times it has the label 1 and plug into the LUT the value which occurs most often.

But what if a bit pattern has an equal number of examples for $y = 0$ and $y = 1$ or the bit pattern does not occur in the training set at all? We call such circumstances "ties" and in order to break them, we assign in the LUT a random entry for that bit pattern. Going back to our example, suppose for the pattern 11 there are 50 training examples with label 1 and 50 training examples with label 0. We randomly sample a value from $\{0, 1\}$, obtaining 0. The LUT now looks like this:

$$f(\boldsymbol{x}) = \begin{cases} 0^* & \text{if} \quad \boldsymbol{x} = 00, \\ 1 & \text{if} \quad \boldsymbol{x} = 01, \\ ? & \text{if} \quad \boldsymbol{x} = 10, \\ ? & \text{if} \quad \boldsymbol{x} = 11, \end{cases} \tag{3.3}$$

where we have annotated the random LUT entry with the symbol $*$. The remaining bit patterns 10 and 11 are learned in the same fashion. Figure 3.1 demonstrates constructing a 3-bit LUT.

Training set

x	y
000	0
000	1
000	1
001	1
100	0
110	0
110	1

bit pattern	$\sum_{y=0}$	$\sum_{y=1}$
000	1	2
001	0	1
010	0	0
011	0	0
100	1	0
101	0	0
110	1	1
111	0	0

bit pattern	f
000	1
001	1
010	0^*
011	1^*
100	0
101	1^*
110	1^*
111	0^*

Figure 3.1.: Constructing a LUT given 3-bit features $\boldsymbol{X}$ with labels $\boldsymbol{y}$. We want to create a LUT which assigns either 0 or 1 to a bit pattern. For each bit pattern, we count how many times $y = 0$ and $y = 1$ and choose the value which occurs more often as final classification. In the case of ties, we assign a random value, denoted by the symbol $*$.

In [3], it is shown that a learned LUT is the best we can do given the training set, meaning no other learning scheme will yield a classifier with higher accuracy. However, single LUTs become impractical with increasing arity, i.e., the number of bits taken as argument. For example, a very small image of size 28×28 has $28 \cdot 28 = 784$ entries. A 784-LUT would need to have $2^{784} \propto 10^{236}$ entries which is computationally infeasible.

Instead of random tie breaking, we could also sample randomly. Suppose there are 10 examples in the dataset with $\boldsymbol{x} = 01$ with six examples having the label $y = 0$ and four examples $y = 1$. During inference, if we encounter $\boldsymbol{x} = 01$, we could classify it with $y = 0$ using a probability of $6/10$, as opposed to always assigning $y = 0$. We could also make the probability linear or exponential in the difference. In this thesis, we do not utilize such a scheme, but it would be interesting to try out in the future.

3.2. Network of LUTs

In this section, we will look at how to construct a network of multiple LUTs according to [3] which will be able to handle more bits. Consider the dataset from Figure 3.1. Instead of constructing a LUT that utilizes all bits, we now take **random** subsets of the columns. Suppose $\boldsymbol{x} = \{x_0, x_1, x_2\}$, where x_j are the individual entries of $\boldsymbol{x}$, then we consider $\{x_0, x_1\}$ and $\{x_0, x_2\}$. We obtain two matrices with seven rows (same as before), but only two columns instead of three. With these two matrices we can construct two LUTs, using the **same** label vector y. The matrices and new LUTs are shown in Figure 3.2.

Training set

$\{x_0, x_1\}$	y
00	0
00	1
00	1
00	1
10	0
11	0
11	1

bit pattern	$\sum_{y=0}$	$\sum_{y=1}$	f_0
00	1	3	1
01	0	0	1*
10	1	0	0
11	1	1	1*

Prediction on training set

$\{x_0, x_1\}$	$f_0(\boldsymbol{x})$
00	1
00	1
00	1
00	1
10	0
11	1
11	1

Training set

$\{x_0, x_2\}$	y
00	0
00	1
00	1
01	1
10	0
10	0
10	1

bit pattern	$\sum_{y=0}$	$\sum_{y=1}$	f_1
00	1	2	1
01	0	1	1
10	2	0	0
11	0	0	1*

Prediction on training set

$\{x_0, x_2\}$	$f_1(\boldsymbol{x})$
00	1
00	1
00	1
01	1
10	0
10	0
10	0

Figure 3.2.: Two subsets of columns of the original dataset from Figure 3.1 that are the basis for two new LUTs. Note that the label vector y is the **same** for both LUTs.

We now have two separate LUTs, but we would like to construct a network. In order to do that, we apply each LUT on the dataset it was trained on and stack the predictions horizontally. The stacked predictions, along with the original label column $\boldsymbol{y}$, form a new dataset on which we can train a third LUT. This process is visible in Figure 3.3.

$\{x_0, x_1\}$	$f_0(\boldsymbol{x})$
00	1
00	1
00	1
00	1
10	0
11	1
11	1

$+$

$\{x_0, x_2\}$	$f_1(\boldsymbol{x})$
00	1
00	1
00	1
01	1
10	0
10	0
10	0

$=$

$f_0(\boldsymbol{x})$	$f_1(\boldsymbol{x})$
1	1
1	1
1	1
1	1
0	0
1	0
1	0

$\{f_0(\boldsymbol{x}), f_1(\boldsymbol{x})\}$	y
11	0
11	1
11	1
11	1
00	0
10	0
10	1

bit pattern	$\sum_{y=0}$	$\sum_{y=1}$	f_2
00	1	0	0
01	0	0	1*
10	1	1	0*
11	1	3	1

Figure 3.3.: Training of f_2, the final LUT. Predictions of f_0 and f_1 on the original dataset are stacked horizontally and form a new dataset, along with the original label vector $\boldsymbol{y}$.

With the final LUT visible in Figure 3.3, we have a *network* which might not be evident by just looking at tables. Figure 3.4 provides a visualization.

bit pattern	f_0	f_1	f_2
00	1	1	0
01	1*	1	1*
10	0	0	0*
11	1*	1*	1

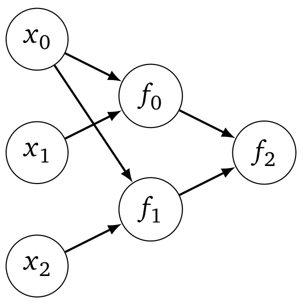


Figure 3.4.: A small 2-LUT network. The inputs x_j get fed into the first two LUTs and their predictions is the input to the last LUT. Note that unlike in neural networks, the neurons are not fully connected, but connections are sparse.

The aforementioned LUT network has one *hidden layer*, meaning one layer of LUTs between the input and output. Increasing the layer size results in a more complex architecture. Unlike in a neural network, where all parameters are updated at each training step, hidden layers in a LUT network are built successively and a layer does not change anymore once it is built.

Connections exist only between adjacent layers. A LUT network with zero hidden layers is a single LUT.

The examples so far were very small. In practice, we will consider much higher dimensional datasets. There are several parameters which we have to choose when constructing a LUT network, which are

1. the arity of the individual LUTs,
2. the number of hidden layers, and
3. the number of LUTs per hidden layer.

One important thing to note is that the output layer is just one LUT because in the end we want to obtain either 0 or 1. Figure 3.5 shows a 2-LUT network with five inputs, three hidden layers and four LUTs per hidden layer. We can see that the LUT that gives us the final prediction, f_{12}, has two connections to the previous layer (as every LUT in the network). That means two LUTs in the final layer do not contribute to the prediction at all. It is also possible that LUTs in other layers or even inputs do not contribute to the final prediction. In Figure 3.5, everything that does not influence the final prediction is dotted. Algorithm 1 shows the just described scheme.

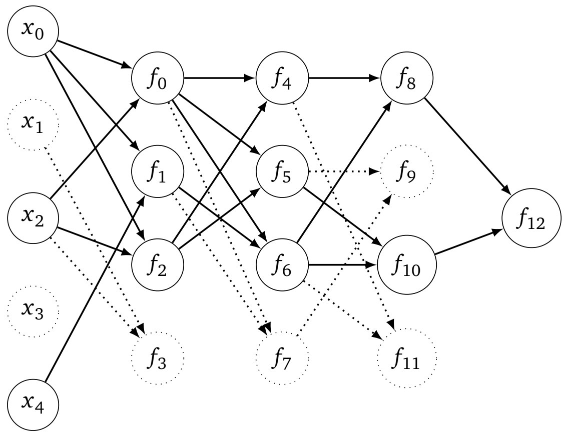


Figure 3.5.: Generic 2-LUT network with five inputs, three hidden layers and four LUTs per hidden layer. A single LUT gives the final prediction and it takes two inputs, as all LUTs in the network. That means there are parts that do not contribute to the final prediction at all which are shown dotted.

Algorithm 1 Constructing a LUT network according to [3]

```
Given binary features $\boldsymbol{X}$ (binary meaning entries are either 0 or 1) with $N$ rows and $d$
columns and binary label vector $\boldsymbol{y}$ with $N$ entries, construct a LUT network that takes
a $d$-dimensional binary vector and classifies it with either 0 or 1.

Choose hyperparameters:
    Arity (number of arguments) $\delta$ of individual LUTs, where $\delta \leq d$
    Number of hidden layers $L$
    Number of LUTs per hidden layer $l_1, l_2, \ldots, l_L$
for $i = 1, \ldots, L$ do
    for $j = 1, \ldots, l_i$ do
        Randomly choose $\delta$ columns from the previous layer
        for bit pattern $1, \ldots, 2^\delta$ do
            Count how many times $y = 0$
            Count how many times $y = 1$
            Choose the label that occurs most often
            In case of a tie, make a random choice
        end for
    end for
    Propagate, obtaining dataset with new features, but same labels $\boldsymbol{y}$
end for
Create LUT in the last layer
```

3.3. Computing the size of LUT networks

As much as we are interested in the performance in terms of accuracy of LUT networks, we are also interested on the size they take up on disk. Computing the parameter size of LUT networks is straight-forward. LUT networks are built of individual LUTs that are essentially tables with binary entries, meaning each entry takes up one bit of space. A d-LUT has 2^d entries. For a d-LUT network with L hidden layers and $l_{1,\ldots,L}$ LUTs per hidden layer, we can compute the parameter size z of a LUT network in bytes as follows:

$$z = \frac{2^d + \sum_{i=1}^{L} 2^d l_i}{8}, \tag{3.4}$$

where 2^d is added because there is a final LUT after the last hidden layer. Eight bits make up one byte. We can see that the relation between arity and size on disk of LUT networks is exponential, which Figure 3.6 visualizes for LUT networks of varying arity with five hidden layers and 1024 LUTs per hidden layer. Thus, when disk space is limited, we have to choose an arity that is as small as possible. From Equation 3.4 we can also see that the relation between LUTs per hidden layer and LUT network size is linear, so the number of LUTs per hidden layer is not as much of a concern as the arity.

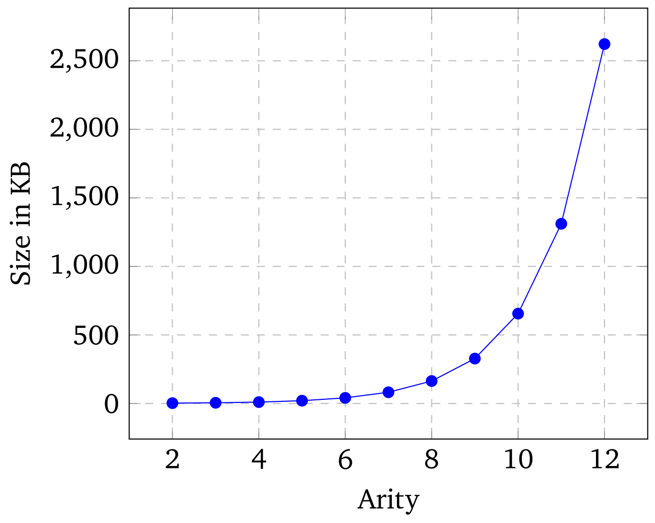


Figure 3.6.: Size in KB of LUT networks with five hidden layers and 1024 LUTs per hidden layer dependent on the arity. We can see that the relation is exponential. Thus, whenever disk space is a concern, we have to choose the arity appropriately.

3.4. Binary-MNIST

As in [3], we use the MNIST dataset [11] for our experiments. It consists of 28x28 grayscale images of handwritten digits from 0-9 and has 60000 training examples and 10000 testing examples. Figure 3.7 visualizes 16 random images from the MNIST dataset. Before we can use this dataset, we have to modify it. The labels range from 0-9, but a LUT network is only able to output 0 or 1 as classification. Thus, we define the task to distinguish between numbers 0-4 (label 0) and numbers 5-9 (label 1).

The inputs to a LUT network also have to be either 0 or 1, so we must binarize the dataset. We transform the data such that the minimum value becomes 0 and the maximum value becomes 1. Each entry is changed to 0 if it is smaller than 0.5 and 1 if it is bigger or equal 0.5. Figure 3.8 shows a part of an original and binarized image.

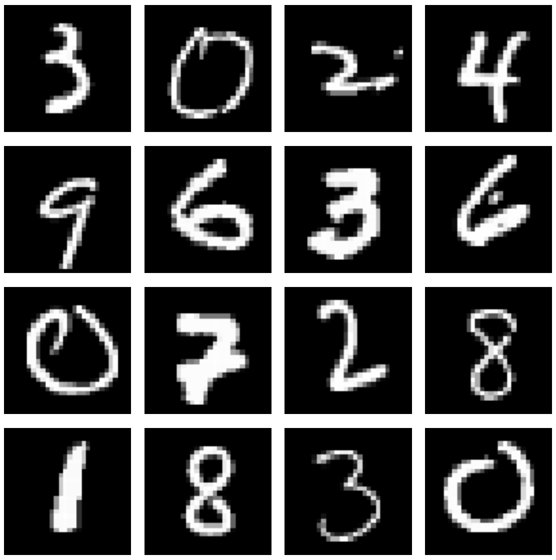


Figure 3.7.: Random selection of images taken from the MNIST dataset [11]. The MNIST dataset consists of 28x28 grayscale images of handwritten digits from 0-9 and has 60000 training examples and 10000 testing examples.

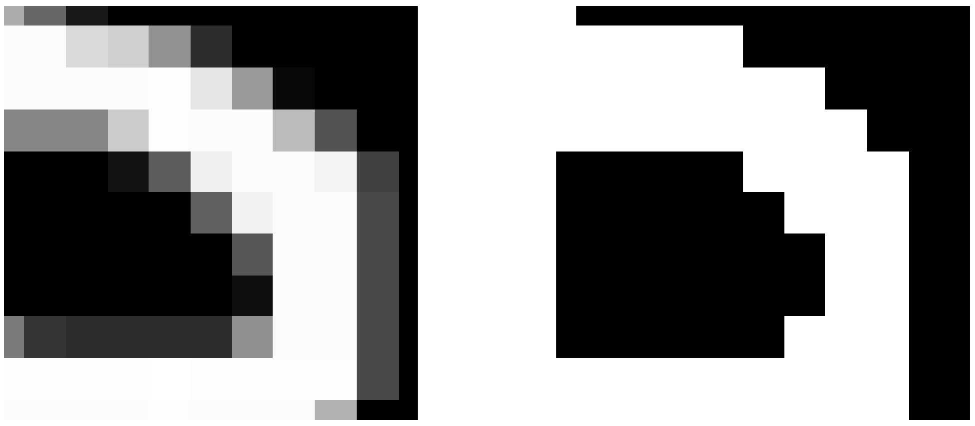

(a) Original. **(b)** Binarized.

Figure 3.8.: Part of sample from the original (a) and binarized (b) MNIST dataset. The original image is the number three and here we show a zoomed-in version (the upper right corner) to make the binarization visible. A LUT network can only do binary classification, so we have to binarize the labels as well. Numbers from 0-4 get the label 0 and numbers from 5-9 get the label 1.

3.5. Experiments

In the following, we repeat one experiment from [3]. We construct a LUT network with five hidden layers and 1024 LUTs per hidden layer. Every LUT in the network takes eight bits as input. Construction of the network is performed using the training set with 60000 samples and after training we validate on the test set with 10000 samples. The code is written by ourselves in Python using the package NumPy [12] for matrix operations.

Similar to other machine learning packages, a LUT network object with provided arguments is first created and methods like `.train()` and `.predict()` allows the user to work with the LUT network at a high level. The code is available on GitHub [13]. We obtain a training accuracy of 0.89 and a test accuracy of 0.87 which are the exact results from [3]. Note that the results are way above 0.5 meaning that learning had definitely taken place.

The way the LUT network is defined, every individual LUT already gives a prediction on the example passed to the network. Hence, we can compute the accuracy of every LUT in the network. Figure 3.9 visualizes the training accuracy dependent on the layer. Each point represents the mean training accuracy over the respective layer and the total height of the error bars are two standard deviations. We can see that with increasing layer number, the accuracy goes up and the standard deviation goes down until it reaches zero at layer six because there is only one LUT in the last layer. Increasing performance with increasing depth reminds us of neural networks where adding more layers is a heuristic to increase performance.

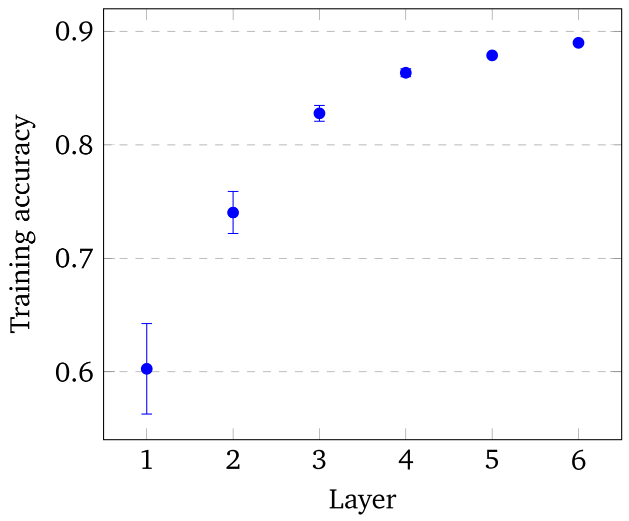


Figure 3.9.: Training accuracy dependent on the layer for an 8-LUT network with five hidden layers (six is the output layer with just one LUT) and 1024 LUTs per hidden layer, trained on the Binary-MNIST dataset using our own code [13]. The points represent the mean over the respective layer with two standard deviations as total height of the error bars. Similar to neural networks, we can see that performance increases with increasing depth. These results are our own and are almost identical to [3].

The arity of the LUTs (denoted by δ in Algorithm 1) is a crucial hyperparameter. In another experiment, [3] varies the arity from two to 16 with steps of two and observes the effect on training and testing accuracy. The rest of the LUT network is the same as before. Increasing the arity sufficiently drives the training accuracy near perfection. As with machine learning models in general, the testing accuracy goes down with too much complexity, meaning we are overfitting. We get the same results when we vary the arity from two to eight using our own code. Above that point, our implementation consumes too much memory, and we are unable to fully recreate the experiment. Figure 3.10 visualizes the results.

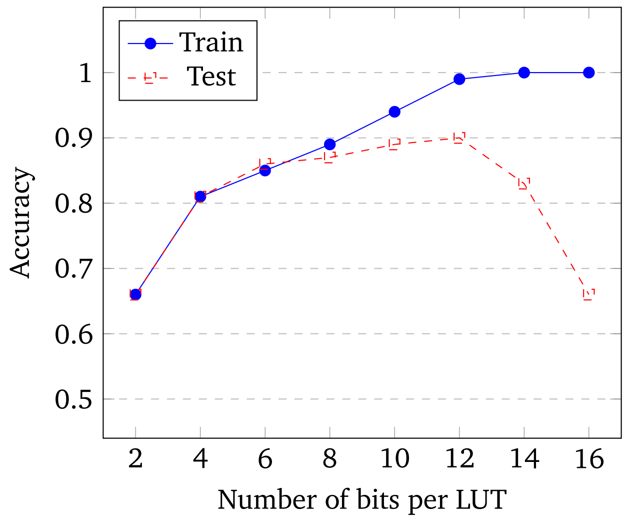
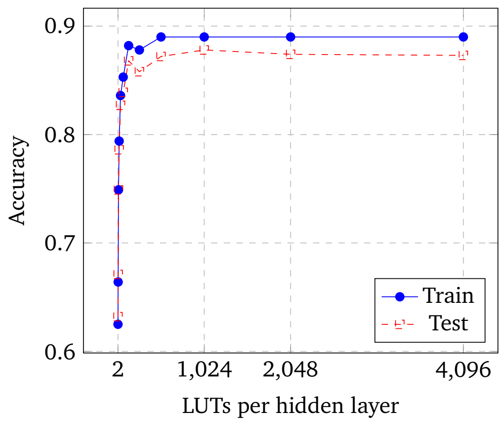


Figure 3.10.: Training and testing accuracy on Binary-MNIST when the arity of a LUT newtwork with five hidden layers and 1024 LUT per hidden layer is varied. The underlying data to produce this plot is taken from [3].

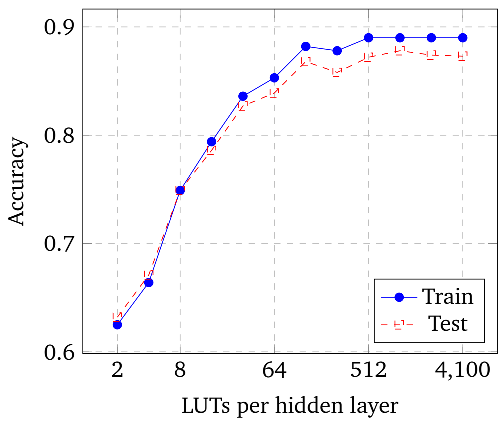
3.6. Varying the number of LUTs per hidden layer

Throughout this thesis, we will stick with LUT networks with five hidden layers and 1024 LUTs per hidden layer to stay consistent with [3]. We will now investigate what effect changing only the number of LUTs per hidden layer has. We use an 8-LUT network with five hidden layers and evaluate the accuracy at 2^i LUTs per layer, where $i = 1, \ldots, 12$. The result can be seen in Figure 3.11, where Figure 3.11a has an arithmetic scale and Figure 3.11b has a logarithmic scale.

Going above 512 LUTs per hidden layer does not increase the accuracy anymore. In Section 3.3 we have shown that the number of LUTs per hidden layer is linearly proportional to the LUT network size on disk. So although not as important as the arity, which influences the LUT network size exponentially, reducing the number of LUTs per hidden layer seems like a quick and easy way to reduce LUT network size, especially if the performance does not drop.



(a) Arithmetic scale.



(b) Logarithmic scale.

Figure 3.11.: Accuracy of an 8-LUT network with five hidden layers dependent on the number of LUTs per hidden layer, where (a) provides an arithmetic scale and (3.11b) provides a logarithmic scale. Increasing the number of LUTs per hidden layer above 512 has no effect on the accuracy.

3.7. Summary

We introduced the concept of LUTs, which are classifiers working entirely in binary, from [3] by walking through a small example. A single LUT cannot grasp complex information and is limited in arity due to computational constraints, thus we considered LUT networks that are made of many individual LUTs. Using a binarized version of the MNIST dataset, which we call Binary-MNIST, we recreate one experiment from [3]. Our results match the ones from the paper. We also made ourselves aware that the arity influences the LUT (and thus LUT-network) size exponentially and that the number of LUTs per hidden layer can sometimes be reduced, resulting in a smaller size and almost no performance drop.

Chapter 4

Improving LUT-based Architectures

The LUT networks considered so far were able to learn something, but in order to be viable for real-world ML use, we have to improve the accuracy. Therefore, we now take the idea of LUT networks further and try to improve the accuracy, where we take three main approaches:

- modifying existing LUT networks or the LUT network learning algorithm,
- enhancing the dataset using feature-engineering, and
- combining many LUT networks of low arity (ensembling).

We will see that we are indeed able to improve the accuracy, where we obtain the best results with an ensembling technique. With an appropriate architecture choice, we are also able to keep model size low.

4.1. Majority vote

So far the final prediction came from a single LUT that takes its inputs from the last hidden layer. Considering the same architecture as in the previous section, the last hidden layer has 1024 LUTs. If the number of bits per LUT is eight, then for the final prediction only eight of all 1024 LUTs will be used. Since every LUT is trained with the same label vector $\boldsymbol{y}$, each of the 1024 LUTs could theoretically be used for a final prediction too. That motivates us to try a *wisdom of the crowd* technique, a majority vote mechanism that takes into account the predictions of all LUTs in the last hidden layer. The prediction that occurs most often will be chosen as the final one.

The complete scheme can be seen in Algorithm 2. We conduct an experiment using a LUT network with five hidden layers and 1024 LUTs per hidden layer, varying the arity from two to 10 with steps of one, predicting with and without majority vote each time. The result can be seen in Figure 4.1. We can see that for a low arity, using a majority vote significantly boosts the

Algorithm 2 LUT network majority vote

Given binary vector $\boldsymbol{x}$ and trained LUT network according to Algorithm 1 with l_L LUTs in the last hidden layer, perform a majority vote to predict.

```
Initialize empty list $\alpha = [\ ]$
for $i = 1, \dots, j_L$ do
    Append prediction of $\text{LUT}_i$ on $\boldsymbol{x}$ to $\alpha$ (either 0 or 1)
end for
if $\sum_{\alpha_i=0} > \sum_{\alpha_i=1}$ then
    Return 0
else if $\sum_{\alpha_i=0} < \sum_{\alpha_i=1}$ then
    Return 1
else
    Return random choice of $\{0, 1\}$
end if
```

training and testing accuracy. For high arity, a majority vote has less effect, but still enhances the testing performance a little. Interestingly, at five bits per LUT all accuracies coincide. Since using a majority vote requires almost no extra computational effort and at worst the results stay the same, it seems to be a good technique for enhancing the network's performance.

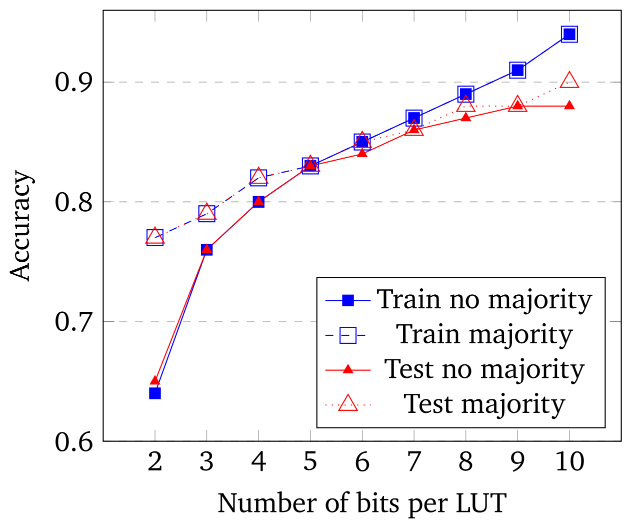


Figure 4.1.: Training and testing accuracies with and without using a majority vote mechanism for a LUT network with five hidden layers and 1024 LUTs per hidden layer. The task is Binary-MNIST. For low arity, using a majority vote significantly enhances performance while the effect lessens with higher arity, but still enhances the testing accuracy a little. Interestingly, all accuracies coincide at five LUTs per hidden layer.

4.2. Maximizing layer-wise mean accuracy while training

As mentioned earlier, every LUT is trained with the same label vector $\boldsymbol{y}$ and is able to give a prediction. With increasing hidden layer, the LUTs get increasingly better. Figure 4.2 visualizes this using histograms. The underlying data is the training accuracy per LUT per hidden layer for an 8-LUT network with five hidden layers and 1024 LUTs per hidden layer applied on the Binary-MNIST dataset. The training accuracy of this LUT network is 0.89 and the testing accuracy is 0.87, the same accuracies as in the first experiment. The x-axis represents the training accuracy. For each hidden layer, we divide the data in 10 equally-sized bins (in terms of the accuracy) from the minimum to maximum value. Then for each bin, we erect a bar, where the height represents the number of data points that fall into this bin.

We can see that on average, the accuracy increases with hidden layer number and the spread of accuracies reduces dramatically. For hidden layer 1, the accuracies are worst and the spread is maximal. The worst LUTs on hidden layer 1 are only slightly better than chance. That motivates us to conduct an experiment in which we try to maximize the accuracy of the LUTs while training.

We conceptualize the following algorithm: after training a layer, we discard the n worst LUTs in that layer and establish n new LUTs that are hopefully better. We said "hopefully" because the learning algorithm takes a random subset of columns of the previous layer, leaving it to chance if a new LUT will perform better in terms of accuracy. After obtaining those n new LUTs, we score every LUT and again discard the n worst LUTs until the mean accuracy does not change anymore. We set a patience p, i.e., a number of iterations after no change in mean accuracy we will move onto the next layer. The implementation is described in Algorithm 3.

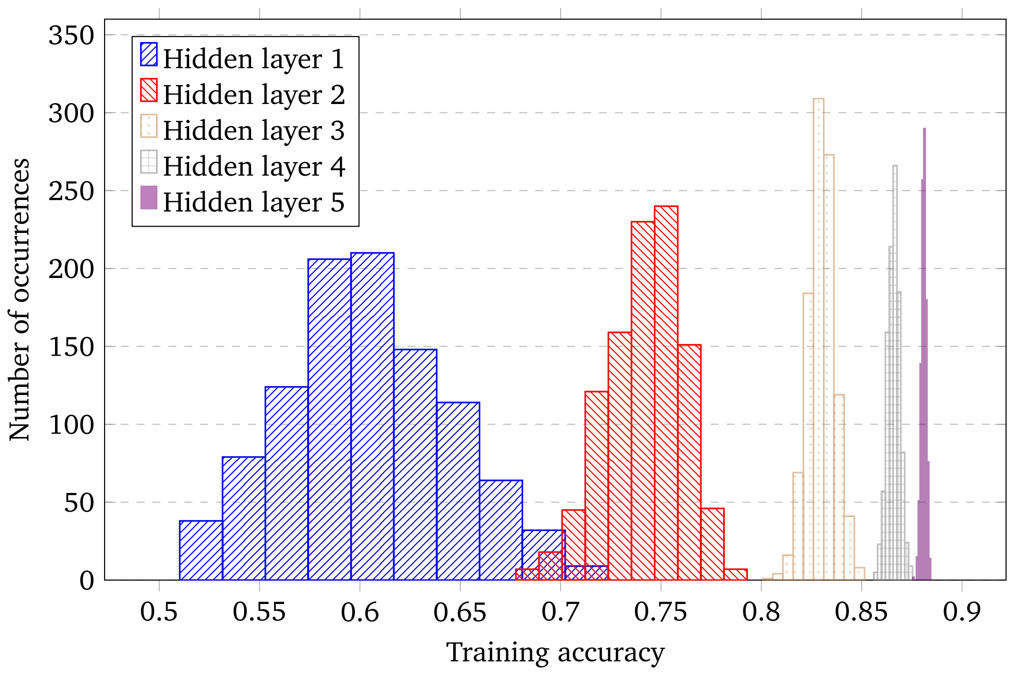


Figure 4.2.: Histograms of training accuracies per hidden layer for an 8-LUT network with five hidden layers and 1024 LUTs per hidden layer. The task is Binary-MNIST. For each hidden layer, the data is divided into 10 equally sized bins from minimum to maximum value. For each range, we erect a bar with the height equal to the number of data points that fall within that range. The overall accuracy of this network is 0.89 on the training set and 0.87 on the testing set.

Algorithm 3 Maximizing layer-wise mean accuracy while training

```
Given binary features $X$ with $N$ rows and binary label vector $y$, construct a LUT network
with $L$ hidden layers where the training accuracy in each hidden layer is maximized. This
algorithm modifies Algorithm 1.

Choose additional hyperparameters:
    Number of LUTs $n$ to discard after each iteration
    Patience $p \in \mathbb{N}$
for $i = 1, \dots, L$ do
    Train layer $L$ as in Algorithm 1
    no_change $\in \mathbb{N} \leftarrow 0$
    best $\in \mathbb{R} \leftarrow 0$
    while no_change $< p$ do
        Discard $n$ worst performing LUTs on $(X, y)$
        Append $n$ new LUTs trained on $(X, y)$ to the current layer
        accs $\in \mathbb{R}^N \leftarrow$ accuracy of each LUT on $(X, y)$
        curr $\in \mathbb{R} \leftarrow$ arithmetic mean(accs)
        if curr $>$ best then
            best $\leftarrow$ curr
            no_change $\leftarrow 0$
        else
            no_change $+= 1$
        end if
    end while
end for
The single LUT after the last hidden layer is created as usual
```

Starting another experiment, we again use an 8-LUT network with five hidden layers and 1024 LUTs per hidden layer and the Binary-MNIST dataset. We set $p = 10$ and $n = 50$. After running Algorithm 3, we obtain a training accuracy of 0.90 and a testing accuracy of 0.88, slightly better than before. In Figure 4.3 we can see the accuracies per hidden layer visualized using a histogram. Contrary to Figure 4.2, the distributions are much more narrow now. After training, we also tried prediction with a majority vote, but the accuracies did not change.

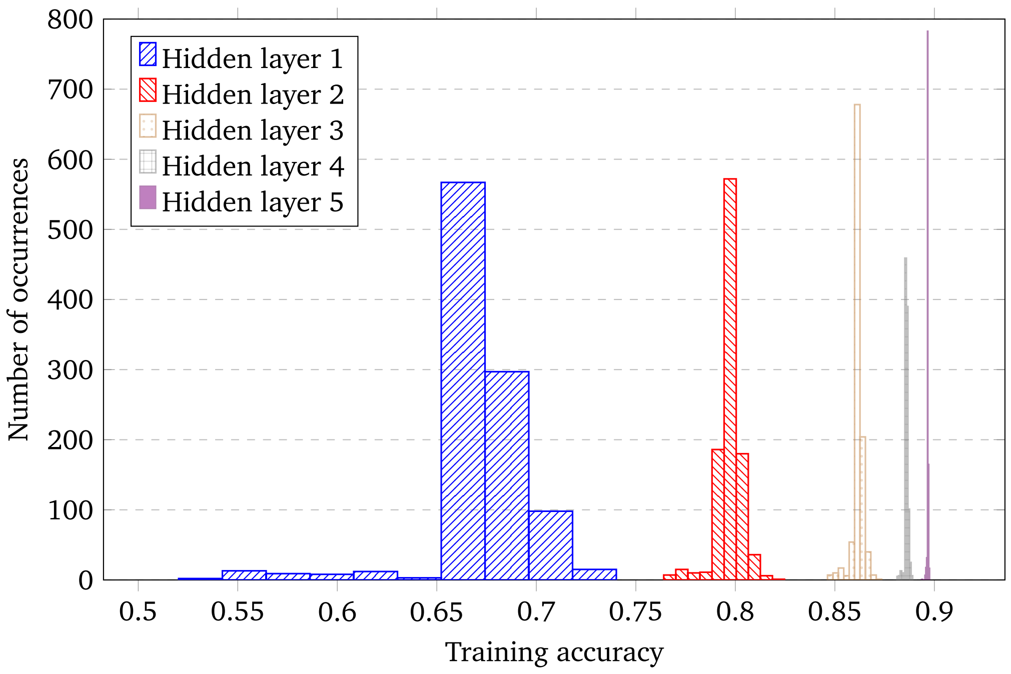


Figure 4.3.: Histograms of training accuracies per hidden layer for an 8-LUT network with five hidden layers and 1024 LUTs per hidden layer, applied on Binary-MNIST. For this network, we applied Algorithm 3 to try to improve performance, yielding an overall accuracy of 0.90 on the training set and 0.88 on the testing set. The results of this network are slightly better (increase of 1% training and testing accuracy) compared to the same network without optimization. Contrary to Figure 4.2, the distributions are much narrower.

For a more complete picture, we vary the arity from two to 10 while applying Algorithm 3. There are again five hidden layers with 1024 LUTs per hidden layer. The result can be seen in Figure 4.4. We can see that pushing up the layer-wise accuracy does indeed help increase the overall accuracy, albeit only significantly for low arity. For high arity, the effect is diminished.

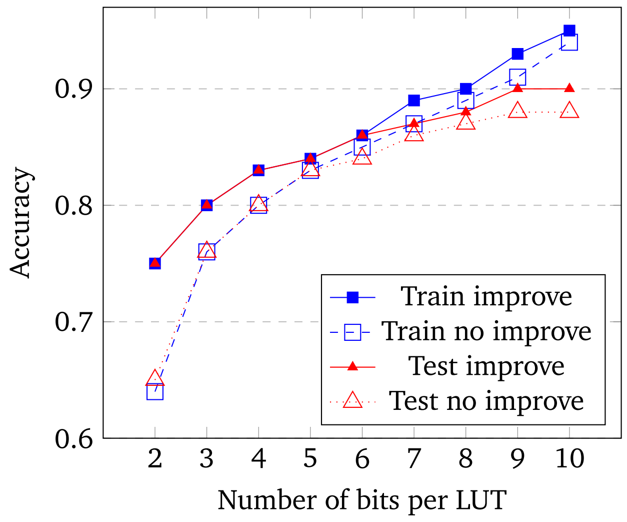


Figure 4.4.: Training and testing accuracies for LUT networks of varying arity with five layers and 1024 LUTs per hidden layer applied on Binary-MNIST. Solid lines and markers denote accuracies of networks where we applied Algorithm 3. For low arity, improving the layer-wise accuracies has a significant effect on the overall performance, while for higher arity the effect is diminished.

4.3. Feature engineering

So far we have not done any more sophisticated feature engineering except making a binary version out of the original MNIST dataset. Since feature engineering is an integral part of machine learning in general, we try it too and see if there is an improvement of the model's performance. We enhance the data by emphasizing the edges. Concretely, we create a copy of the original image and replace each pixel with black if the adjacent pixel has the same value and white if the adjacent pixel has a different value. We do this process both in x- and y-direction and add the results to the original image. Figure 4.5 illustrates this technique for one example.

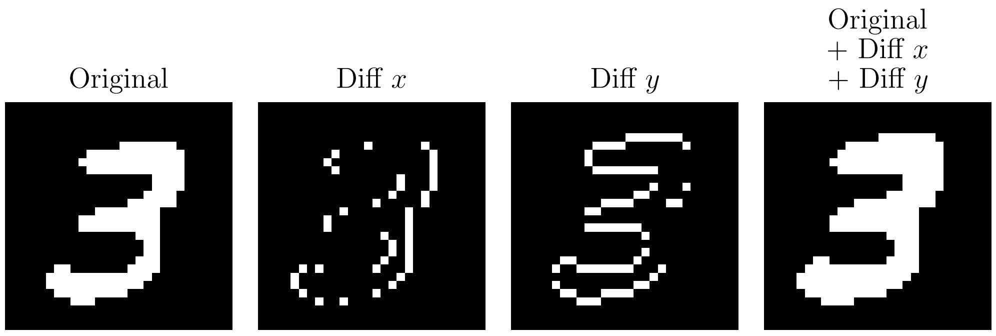


Figure 4.5.: Single example from the training data where we applied feature engineering. We create a copy of the original image and replace each pixel with black if the adjacent pixel has the same value and white if the adjacent pixel has a different value. We perform this process both in x- and y-direction and add the results to the original image. Looking at the feature-engineered image on the right, the number appears thicker.

Using the dataset enhanced with feature engineering, we train a LUT network with five hidden layers and 1024 LUTs per hidden layer, varying the arity from two to 10. The result can be seen in Figure 4.6. We can see that the accuracies are consistently higher for models that were trained on images where feature engineering had been applied. The technique described in this section was inspired by the Sobel operator [14] which calculates the gradients in an image in either x- or y-direction. Using this operator as feature engineering technique as opposed to our own produces similar results. We presented our own as main technique for the sake of a simpler explanation and brevity.

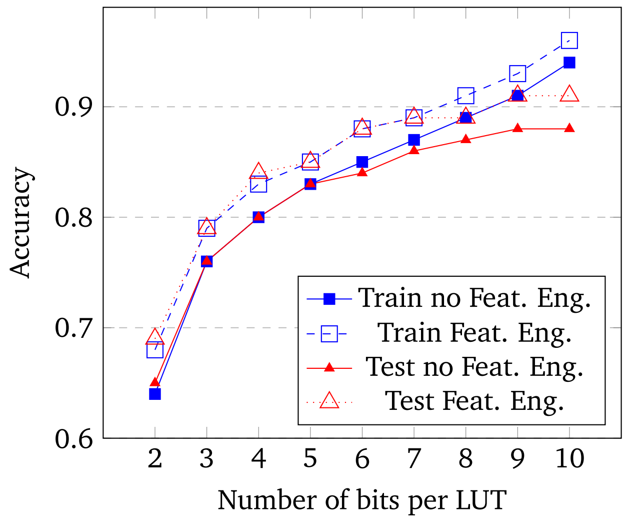


Figure 4.6.: Train and test accuracies for models trained on unmodified Binary-MNIST data and Binary-MNIST data where feature engineering had been applied. We can see that using feature engineering boosts the performance.

4.4. Plain LUT ensembling

In Section 4.1, as opposed to having one final LUT, we counted the number of output labels in the last hidden layer and took that label as final prediction whichever was occurring the most. We could view that as a *wisdom of the crowd* technique. Another approach is to bunch together n separate LUT networks that do not use a majority vote and combine their prediction into one. We simply use that label as final prediction which was predicted the most out of the n LUT networks. In the event that both labels occur the same number of times, we make a random choice. The scheme is described in detail in Algorithm 4.

We conduct an ensembling experiment where we use 2-LUTs as base classifiers. Keeping the arity low allows us to increase the number of individual LUT networks to a much higher degree. Training and inference for 8-LUT networks would take too long, at least with our current code. Figure 4.7 visualizes the result of the experiment. We use LUT network ensembles with $2^{1,\dots,10}$ individual LUT classifiers. At just two LUT networks, the training accuracy is fairly low at 0.68, as it would be with a single 2-LUT network. Increasing the number of LUT networks increases the training accuracy and a peak at 0.78 with 64 LUTs is reached. Increasing the number of LUTs even more has no effect on the accuracy. Interestingly, the testing accuracy is consistently higher than the training accuracy.

Algorithm 4 LUT ensembling

```
Given trained $\text{LUT}_{1,\ldots,n}$, perform ensembling to predict on yet unseen $\boldsymbol{x}$.

Initialize empty list $\alpha = [\,]$
for $i = 1, \ldots, n$ do
    Append prediction of $\text{LUT}_i$ on $\boldsymbol{x}$ to $\alpha$ (either 0 or 1)
end for
if $\sum_{\alpha_i=0} > \sum_{\alpha_i=1}$ then
    Return 0
else if $\sum_{\alpha_i=0} < \sum_{\alpha_i=1}$ then
    Return 1
else
    Return random choice of $\{0, 1\}$
end if
```

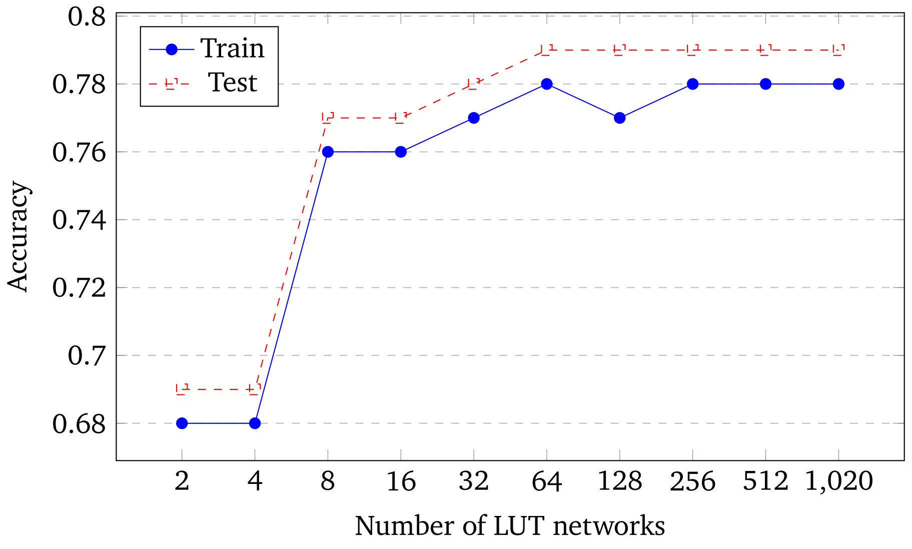


Figure 4.7.: Train and test accuracies for LUT network ensembles according to Algorithm 4 on Binary-MNIST. The base classifiers are 2-LUT networks with five hidden layers and 1024 LUTs per hidden layer. Training and testing accuracies reach a peak at 64 LUT networks, increasing the LUT number even more has no effect on the accuracy. Interestingly, the testing accuracy is consistently higher than the training accuracy.

4.5. Advanced LUT ensembling

Now we try another ensembling method which is very well established in literature and practice, the AdaBoost.M1 algorithm [15] which can be seen in Algorithm 5. As the scheme in the previous Section 4.4, AdaBoost.M1 combines predictions of n separate classifiers $f_{1,\dots,n}$ into one. Every classifier is trained individually with the addition of weights $\boldsymbol{w}$, where $w_k > 0$. Every sample k in the training set is assigned weight $w_k = 1/N$ in the beginning, where N is the number of samples. The weights represent how much attention during training a single example should get, the higher the weight, the more attention the sample gets. Since all weights are equal in the beginning, every sample gets the same amount of attention in the first iteration.

Once iteration i is finished, a number $\alpha_i \in \mathbb{R}$ is computed. If $\alpha_i > 0$, then the classifier is better than random and if $\alpha_i < 0$, then the classifier is worse than random. Using α_i, we update each weight of only those samples which were misclassified. If the classifier is better than random, then we can do some fine-tuning and focus more on the misclassifications which corresponds to assigning higher weights to misclassified samples. If the classifier is worse than random, then we are not yet ready for fine-tuning, and we should focus more on the most important samples which corresponds to assigning lower weights to misclassified samples. After n iterations, training is finished, and we can predict on yet unseen sample $\boldsymbol{x}$ using $\text{sign}(\sum_{i=1}^{n} \alpha_i f_i(\boldsymbol{x}))$.

Algorithm 5 AdaBoost.M1 algorithm according to [15]

```
1:  Given dataset $(\boldsymbol{X}, \boldsymbol{y})$, where $\boldsymbol{X} = \boldsymbol{x}_1, \dots, \boldsymbol{x}_N$ and $\boldsymbol{y} = y_1, \dots, y_N$, $y_i \in \{-1, 1\}$ and model
    class, construct a classifier that is made up of $n$ individual classifiers $f_1, \dots, f_n$.
2:  Initialize weight vector $\boldsymbol{w} \in \mathbb{R}^N$ with $w_{1,\dots,N} = 1/N$
3:  for $i = 1, \dots, n$ do
4:      Train classifier $f_i$ on $(\boldsymbol{X}, \boldsymbol{y})$ and weights $\boldsymbol{w}$
5:      $\text{err}_i \leftarrow \left( \sum_{k=1}^{N} w_k \mathbb{1}(f_i(\boldsymbol{x}_k) \neq y_k) \right) / \sum_{k=1}^{N} w_k$
6:      $\alpha_i = \log((1 - \text{err}_i)/\text{err}_i)$
7:      for $k = 1, \dots, N$ do
8:          $w_k \leftarrow w_k \exp(\alpha_i \mathbb{1}(f_i(\boldsymbol{x}_k) \neq y_k))$
9:      end for
10: end for
11: Prediction on yet unseen $\boldsymbol{x}$: $\text{sign}\left( \sum_{i=1}^{n} \alpha_i f_i(\boldsymbol{x}) \right)$
```

Using AdaBoost.M1 with LUT networks only works if we modify the algorithm slightly. Looking at Algorithm 5 on line 4, we notice that there is no way to incorporate weights into the LUT learning algorithm (see Algorithm 1). Therefore, instead of training on all samples $\boldsymbol{X}$ with weights at each iteration, we train on a subset of samples $\boldsymbol{X}' \subset \boldsymbol{X}$ with fixed size $\rho \in \{1, \ldots, N\}$ without weights. By choosing those samples with the highest weights to be included in $\boldsymbol{X}'$, we are able to give more attention to those samples.

The modified AdaBoost.M1 for LUT networks can be seen in Algorithm 6. The modifications take place from line 5 to line 9. On the first iteration, we train on the entire dataset. On all other iterations, we train on ρ samples with the highest weights. The command argsort($\boldsymbol{w}$) returns indices that would sort $\boldsymbol{w}$. We are interested in the highest weights, not the lowest ones, so we reverse the sorted indices: reversed(argsort($\boldsymbol{w}$)). We then take a subset, the first ρ entries: reversed(argsort($\boldsymbol{w}$))[: ρ]. Now we have the indices of ρ samples with the highest weights and take a subset of the dataset: $\boldsymbol{X}$[idxs] and $\boldsymbol{y}$[idxs].

Algorithm 6 LUT ensembling inspired by AdaBoost.M1

```
1: Given dataset $(\boldsymbol{X}, \boldsymbol{y})$, where $\boldsymbol{X} = \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ and $\boldsymbol{y} = y_1, \ldots, y_N$, $y_i \in \{-1, 1\}$, construct a
   classifier that is made up of $n$ individual classifiers $\text{LUT}_{1,\ldots,n}$.
2: Initialize weight vector $\boldsymbol{w} \in \mathbb{R}^N$ with $w_{1,\ldots,N} = 1/N$
3: Choose $\rho$ out of $\{1, \ldots, N\}$
4: Train $\text{LUT}_1$ on $(\boldsymbol{X}, \boldsymbol{y})$
5: for $i = 2, \ldots, n$ do
6:     idxs = reversed(argsort($\boldsymbol{w}$))[: $\rho$]
7:     $\boldsymbol{X}' \subset \boldsymbol{X} \leftarrow \boldsymbol{X}[\text{idxs}]$
8:     $\boldsymbol{y}' \subset \boldsymbol{y} \leftarrow \boldsymbol{y}[\text{idxs}]$
9:     Train $\text{LUT}_i$ on $(\boldsymbol{X}', \boldsymbol{y}')$
10:    $\text{err}_i \leftarrow \left( \sum_{k=1}^N w_k \mathbb{1}(\text{LUT}_i(\boldsymbol{x}_k) \neq y_k) \right) / \sum_{k=1}^N w_k$
11:    $\alpha_i = \log((1 - \text{err}_i)/\text{err}_i)$
12:    for $k = 1, \ldots, N$ do
13:        $w_k \leftarrow w_k \exp(\alpha_i \mathbb{1}(\text{LUT}_i(\boldsymbol{x}_k) \neq y_k))$
14:    end for
15: end for
16: Prediction on yet unseen $\boldsymbol{x}$: $\text{sign}\left( \sum_{i=1}^n \alpha_i \text{LUT}_i(\boldsymbol{x}) \right)$
```

We apply Algorithm 6 and use 2-LUT networks with five hidden layers and 1024 LUTs per hidden layer and choose $\rho = N/5$. We vary the number of LUT networks from 2 to 1024. The motivation for choosing 2-LUT networks is computation time and disk size. Since we want to keep open the possibility for many LUT networks in the ensembled classifier, we have to restrict ourselves with the arity, otherwise it would take too long to train and predict. Our overall goal also includes a small disk size. A LUT network ensemble of 8-LUTs with five hidden layers and 1024 LUTS per hidden layer would take up 168 MB of space, way out of proportions.

The results can be seen in Figure 4.8. We can see that with an increasing number of LUT networks, we can indeed improve the accuracy. At 1024 LUT networks, we achieve a training accuracy of 0.94 and testing accuracy of 0.93. When it comes to the testing accuracy, this is the best result so far. Interesting is the lack of a significant difference between training and testing accuracies. Usually there is a point where overfitting starts to take place and the testing accuracy decreases compared to the training accuracy. Perhaps overfitting will take place at more LUT networks than 1024, but we do not go above that because it would take too much time to train for us.

However, a LUT network ensemble of 2-LUTs with five hidden layers and 1024 LUTs per hidden layer takes up 2.62 MB of disk space, larger than the CNN we used before which was 1.16 MB in size. We therefore evaluate training and testing performance on a 2-LUT ensemble with five hidden layers and only 64 LUTs per hidden layer. This architecture takes up only 168 KB of disk space. We obtain a training and testing accuracy of 0.94.

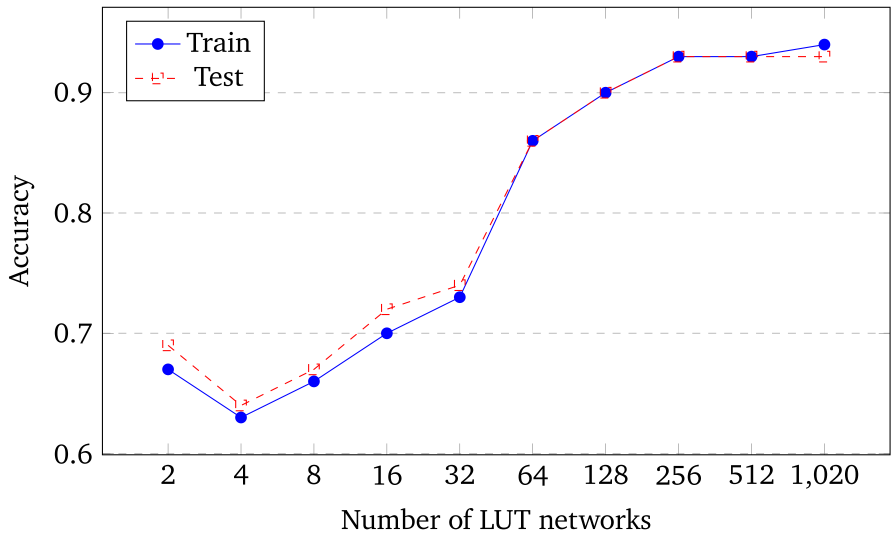


Figure 4.8.: Training and testing accuracies for ensembles of 2-LUT networks with five hidden layers and 1024 LUTs per hidden layer according to Algorithm 6. At 1024 LUT networks, we obtain training and testing accuracies of 0.94 and 0.93, respectively, best so far when it comes to testing accuracy. Notice the lack of significant difference between training and testing accuracy as it would be when overfitting. Perhaps overfitting takes place at more individual LUTs.

As opposed to LUT architectures in previous sections, the AdaBoost.M1 inspired ensemble requires floating-point computations to make predictions (α_i in Algorithm 6). LUTs and LUT networks can be implemented fast on a binary level, since for inference, we just have to look up entries in tables. The floating-point computation here further complexes inference, however, we will not investigate to what degree speed is influenced.

4.6. Combining different methods

We combine different methods presented in previous sections to try to push the accuracies even further. We utilize an AdaBoost.M1 inspired LUT classifier from Section 4.5 with 1024 individual 2-LUT networks that each have 5 hidden layers and 64 LUTs per hidden layer. Additionally, we apply Algorithm 3 to each LUT network while training and also use the feature-engineered dataset from Section 4.3. For the hyperparameters, we set $\rho = N/5$, the number of LUTs to discard after each iteration $n = 10$ and patience $p = 10$. We obtain a training accuracy of 0.75 and testing accuracy of 0.78, worse than the results from Section 4.5. The results make us question Algorithm 3. It is interesting though that pushing up the accuracies of individual LUTs in LUT networks leads to an overall inferior performance when constructing an ensembled classifier.

4.7. Summary

Starting with a LUT-network architecture from [3], we tried to make improvements in terms of accuracy and size which were successful. A majority vote and maximizing the layer-wise mean accuracy while training help improve performance of single LUT networks of low arity, for higher arity, however, the change does not surpass 1-2%. Feature engineering gave small improvements in accuracy. Ensembling techniques are promising, especially an AdaBoost.M1 ensemble achieved a testing accuracy of 0.94, better than any previous single LUT network.

With a good choice of architecture, we were able to reduce the size of the ensemble significantly while retaining accuracy. However, the advance ensembling scheme requires floating-point computations again, albeit few. We also obtained results that suggest combining ensembling and maximizing the layer-wise mean accuracy is not a good idea. What we have not discussed is pruning of LUT networks, i.e. discarding individual LUTs that do not contribute to the final prediction. We have also not discussed multi-class classification which would be possible with LUTs using a one-vs-one or one-vs-all approach similar to SVMs [16].

Chapter 5

And-Inverter Graphs (AIGs)

We will now consider another scheme of building a circuit. First we will introduce the concept of an AIG and how it can be used to predict something. Then, we will devise an initialization scheme and local search algorithm that we try out by running experiments. We will find that the local search algorithm works, and we are able to iteratively improve the accuracy. The resulting AIGs are very small (way below 1 Kilobyte), so their implementation on devices with little computing power is promising. What remains to be done is to try out better search algorithms that are faster and that find AIGs with higher accuracies.

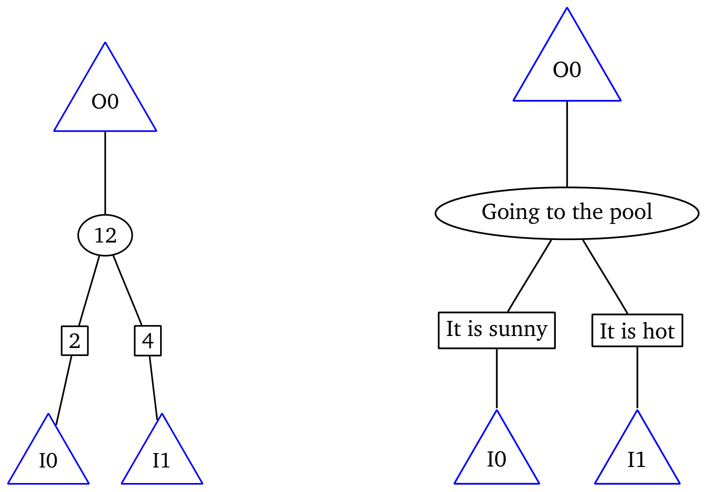
5.1. Introduction

Let us start by introducing another predictive system that looks like a network and works with binary values. Consider Boolean variable 2. For consistency, which will become apparent later on, we use numeric variables. It can either be true or false. Introducing another variable 4, we can perform a logical operation on 2 and 4, similar to how we can perform a mathematical operation with two real numbers (e.g., adding, subtracting, dividing them, etc.). Logical operators include AND ($\wedge$), OR ($\vee$), IMPLICATION ($\Rightarrow$), XOR ($\otimes$) and so on. If we compute

$$12 = 2 \wedge 4, \tag{5.1}$$

where 12 is another Boolean variable, we can visualize this using a graph, visible in Figure 5.1. First let us consider Figure 5.1b. We interpret variables 2 and 4 as *inputs* and thus draw them using boxes and label them additionally using triangle nodes. Variable 6 in Equation 5.1 becomes and *and-node*, i.e., it is the result of performing AND on 2 and 4, and we denote it using a circular node. Since variable 12 is already the output of Equation 5.1, we mark it using a triangle node.

We could already imagine Equation 5.1 being a predictive system. Suppose we want to build a predictive system that forecasts whether teenage boy "Hans" will refresh himself at the family's swimming pool. We are his neighbors and have good information about his daily life, albeit limited. We observe that on a hot and sunny day, he is almost always at the pool. Thus, we assign 2 to true if it is sunny and false otherwise and 4 to true if it is hot and false otherwise. Variable 12 now represents our prediction if Hans will go to the pool. Figure 5.1b visualizes this predictive system.



(a) Plain tree. **(b)** Tree with interpretation of variables.

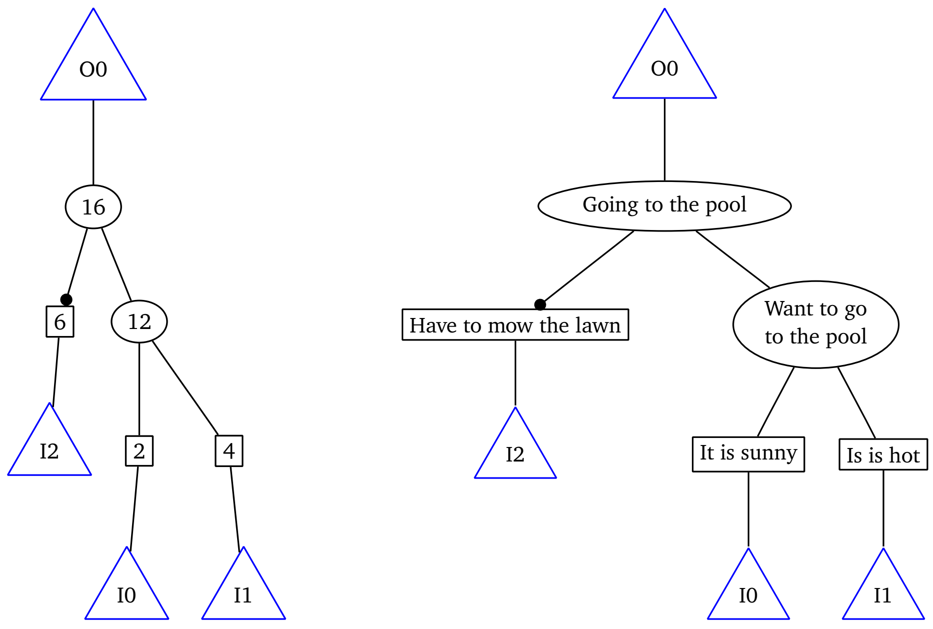
Figure 5.1.: Equation 5.1 visualized using a graph in (a) with a possible meaning to the variables in (b). Inputs are drawn using a box node and further labelled using triangle nodes. Circular nodes perform AND on two other nodes which can be either input nodes or other AND nodes. Outputs are denoted using triangle nodes.

Looking at Figure 5.1b, we notice that it looks like an upside-down tree that has two branches. Consequently, given any node, nodes that are connected and above it, we call "parents". For example, variable 12 is the parent of variable 2 and 4 because it is further up in the tree. Nodes that are below a node and connected to it, we call "children", so variables 2 and 4 are children of variable 12.

We will now consider more complexity in our predictive system. We observe that on some days, Hans is mowing the lawn instead of relaxing at the pool, surely because his parents told him to do so. If binary variable 6 represents whether Hans has to mow the lawn, we can update our predictive system:

$$16 = \neg 6 \wedge (2 \wedge 4), \tag{5.2}$$

where the symbol ¬ denotes negation and variable 16 is the new output. The visualization of Equation 5.2 using a graph can be seen in Figure 5.2. There is a new input (I2) with a black dot at the connection to the next node. This black dot denotes negation.



(a) Plain tree. **(b)** Tree with interpretation of variables.

Figure 5.2.: Predictive system from Figure 5.1 with complexity added. There is now another binary variable "Have to mow the lawn" which is an input. The black dot denotes negation. What was previously the output of the system becomes an intermediate result: "Want to go to the pool".

Lets us now consider even more complexity. We know that Hans has a sister, and sometimes we would find her alone at the pool on a hot and sunny day, but we never observe the two being together there. On one occasion, we see that Hans chases her away from the pool with his friend. Hans' friend is not always visiting, and we have in vague memory that he is an avid baseball fan.

We thus come up with following reasoning: Hans does not like being at the pool with his sister together, probably because he thinks she is annoying. But if his friend is over, he has the courage to chase her away. If a baseball game is on TV, Hans' friend will prefer to stay at home and watch it. The resulting predictive system is visualized in Figure 5.3 with numeric variable names and in Figure 5.4 with an interpretation given to the variables. We invite the reader to take some time studying it. The uppermost part can be understood easily by considering that for boolean variables A and B: $A \lor B \Leftrightarrow \neg(\neg A \land \neg B)$.

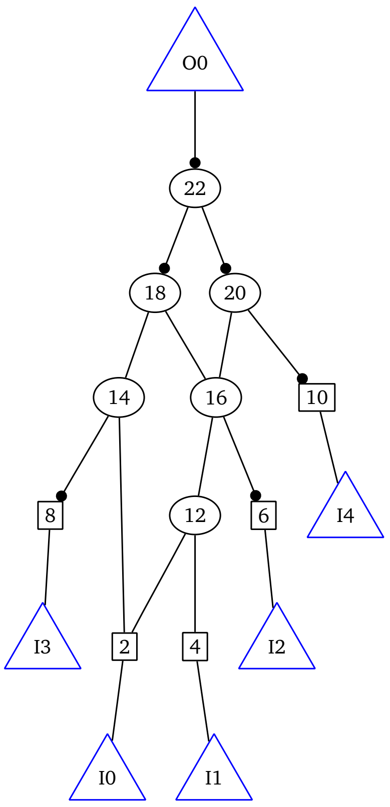


Figure 5.3.: Predictive system if teenage boy Hans will go relax himself at the family's swimming pool, with more complexity added.

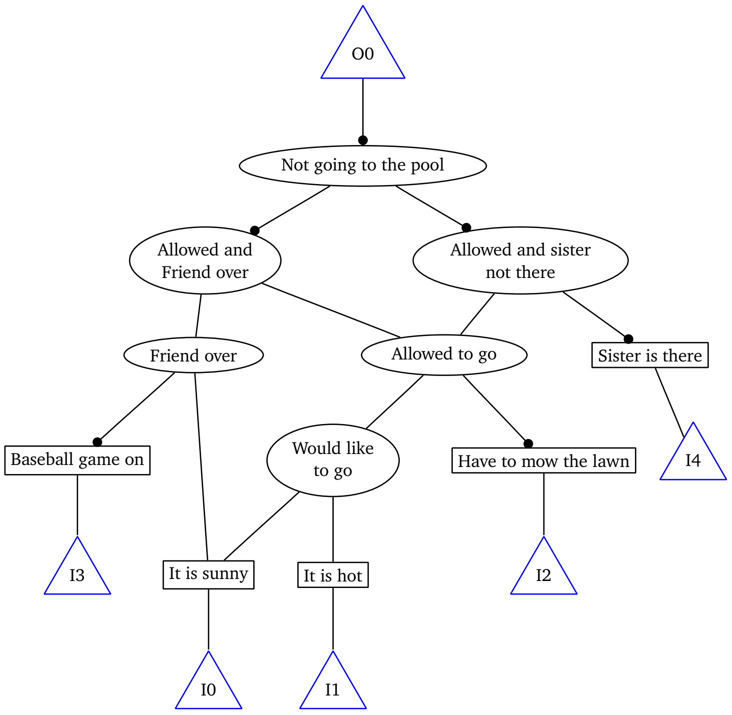


Figure 5.4.: Predictive system if teenage boy Hans will go relax himself at the family's swimming pool, with more complexity added. Here we wrote an interpretation to each node. The uppermost part can be understood easily by considering that for boolean variables A and B: $A \lor B \iff \neg(\neg A \land \neg B)$.

An interpretation, such as in Figure 5.4 is usually not available to us. For example, in the case of the Binary-MNIST dataset, there are 784 possible inputs and a local search algorithm (that we will later introduce) searches for an AIG structure. Even if the meaning of each input is apparent to us, how these inputs are processed further down in the tree is hard to interpret. In the following sections, we will thus not try to come up with an interpretation of the trees.

Looking at Figure 5.4, it looks like we are dealing with facts and implication, but really this is a *predictive system*, so its output is not guaranteed to be true. After all, we constructed this system with our own limited information. There are many more factors to consider if Hans will go to the swimming pool, such as homework, the pool being under maintenance, Hans preferring to play computer games indoor and so on.

As with LUTs, a predictive system like this cannot handle uncertainty, i.e., what is the *probability* of Hans going to the pool. The output is either 0 or 1. One solution would be to discretize the probability range, e.g., 0-25%, 26-50%, 51-75%, 76-100% and have a 2-bit output. In this thesis, however, we do not consider a probabilistic output.

This new type of predictive system introduced in this section is called an “and-inverter graph” (AIG) [17]. AIGs are used to represent logic functions, and they can be easily implemented as circuits. We will use them in a machine learning context, i.e., given a training set with features and labels we search for an architecture that has the highest possible accuracy.

5.2. Initialization of a random AIG

First we must initialize a random AIG that we can later pass to the local search algorithm. We need a systematic way to build an AIG structure, where cyclicity is forbidden.

We use an approach inspired from neural networks. A NN has a certain number of hidden layers and each hidden layer takes inputs only from the previous one. Thus, there is never cyclicity. We take a similar approach for AIGs by specifying the number of hidden layers, the number of nodes per hidden layer and how many outputs there are. Each node takes two inputs (i.e., children c_1 and c_2) randomly from the previous hidden layer and the inputs to the node are randomly negated.

Important to note is that almost all architectures initialized this way have “inactive nodes”, i.e. they do not contribute to the final prediction. Inactive nodes are still part of the architecture, though, and may become active during the search process. An AIG with all inactive nodes removed, we call a “pruned AIG”. Figure 5.5 visualizes a randomly initialized AIG, where the inactive nodes are dotted. Algorithm 7 describes the initialization process more formally.

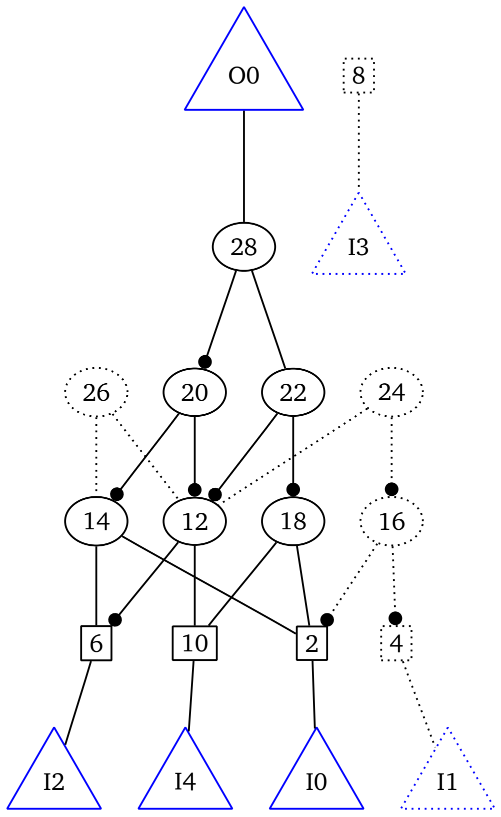


Figure 5.5.: Randomly initialized AIG with inactive nodes (dotted). Inactive nodes do not contribute to the prediction, but may become active during the search process. Input nodes I1 and I3 also do not contribute to the prediction.

Algorithm 7 Random AIG initialization

```
Given number of inputs $I$, number of hidden layers $L$, number of nodes per hidden
layer $l_1, \dots, l_L$ and number of outputs $O$, construct a random AIG.

for hidden layer $i = 1, \dots, L$ do
    for node $l_1, \dots, l_L$ do
        Choose $c_1$ randomly from any nodes in the previous layer
        Negate $c_1$ with a 50% chance
        Choose $c_2$ randomly from any nodes in the previous layer, where $c_2 \neq c_1$
        Negate $c_2$ with a 50% change
    end for
end for
for output node $i = 1, \dots, O$ do
    Choose $c_1$ randomly from any nodes in the last hidden layer
    Negate $c_1$ with a 50% chance
    Choose $c_2$ randomly from any nodes in the hast hidden layer, where $c_2 \neq c_1$
    Negate $c_2$ with a 50% change
end for
```

5.3. Greedy local search

Given our dataset and randomly initialized AIG, we would like an architecture that has the highest possible performance in terms of accuracy, meaning we need a search algorithm. We choose a greedy local search [18] because it is fairly simple and easy to implement, thus a great way to start.

The basic idea is to consider a set of successor architectures. Such a set arises when we consider each active node in the graph and change a small thing, i.e., change a child to another node or negating a child. We score each architecture and take the one that has the highest training accuracy, thus proceeding to the next iteration. If the training accuracy does not improve for a certain number of iterations, we finish. The greedy local search is described in Algorithm 8.

Algorithm 8 AIG greedy local search

```
Given binary dataset $(\boldsymbol{X}, \boldsymbol{y})$ and randomly initialized AIG with architecture $\sigma_0$, perform a
greedy local search.

Choose patience $p$
no_change $\in \mathbb{N} \leftarrow 0$
$s \in \mathbb{R} \leftarrow 0$
$\sigma \leftarrow \sigma_0$
while no_change $< p$ do
    $\Sigma = [\,]$
    $S = [\,]$
    for all active nodes in $\sigma$ do
        $\sigma' \leftarrow$ change $c_1$ to any node in the previous layer, where $c_1 \neq c_2$
        $\Sigma$.append($\sigma'$), $S$.append(accuracy_score($\sigma'$, $(\boldsymbol{X}, \boldsymbol{y})$))
        $\sigma' \leftarrow$ change $c_2$ to any node in the previous layer, where $c_2 \neq c_1$
        $\Sigma$.append($\sigma'$), $S$.append(accuracy_score($\sigma'$, $(\boldsymbol{X}, \boldsymbol{y})$))
        $\sigma' \leftarrow$ change polarity of $c_1$
        $\Sigma$.append($\sigma'$), $S$.append(accuracy_score($\sigma'$, $(\boldsymbol{X}, \boldsymbol{y})$))
        $\sigma' \leftarrow$ change polarity of $c_2$
        $\Sigma$.append($\sigma'$), $S$.append(accuracy_score($\sigma'$, $(\boldsymbol{X}, \boldsymbol{y})$))
    end for
    $s' \leftarrow \max(S)$
    if s' > s then
        $s \leftarrow s'$
        no_change $\leftarrow 0$
        $\sigma \leftarrow \Sigma[\text{argmax}(S)]$
    else
        no_change $+= 1$
    end if
end while
```

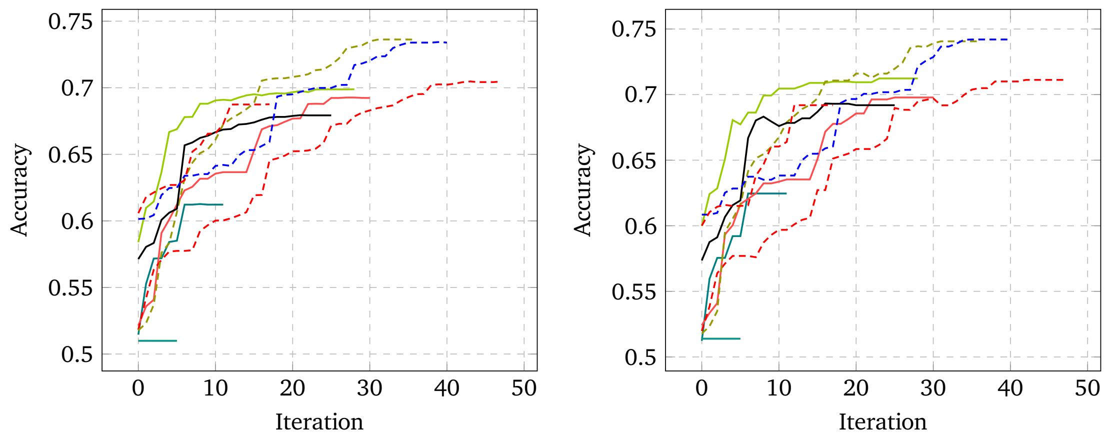
5.4. Experiments

Consider again the Binary-MNIST dataset for our experiment. We use the AIG format and library "AIGER" [19] to specify and run our AIGs. We have written the network structure and local search algorithm from scratch in C++ in approximately 960 lines of code. Along with random initialization and the local search, the code features functionality to export into AIGER format and to export visualizations. In addition, we have written a Python script that exports a binary NumPy array into the same format as the MNIST dataset. Preprocessing of the data can thus be performed in Python. The code is available on GitHub [20].

Utilizing initialization from Algorithm 7 and training from Algorithm 8, we run 42 experiments. There are 784 inputs, and we choose 3 hidden layers with 64 AND nodes per hidden layer, meaning any resulting AIG will maximally have 192 AND-gates. The very low number of AND-gates that we permit is due to computational constraints during search. The output layer has one AND node and takes inputs from the last hidden layer. In Figure 5.6, 10 out of the 42 runs are visualized.

The mean final training and testing accuracies are 0.65 and 0.66, respectively. The maximum final training and testing accuracies are both 0.75. The initialization accuracies vary greatly, from 0.51 to 0.66. A summary can be seen in Table 5.1. A higher initial training accuracy does not necessarily mean a higher final accuracy. A visualization of the pruned best AIG found in terms of testing accuracy can be seen in Figure 5.7.

Due to a bug in our code, the search algorithm found an architecture which is usually not allowed, namely with an AND-node taking two inputs that are the same, thus just copying another AND-node. Nevertheless, this was the best AIG found and its very small size is interesting. The best AIG found with a valid architecture can be seen in Appendix A.2.



(a) Training. **(b)** Testing.

Figure 5.6.: Training and testing accuracies of a greedy local search according to Algorithm 8 on AIGs, with Binary-MNIST as the dataset. We performed 42 experiments, 10 of which are visualized here. The initial accuracies vary from 0.51 to 0.66. A higher initial accuracy does not necessarily mean a higher final accuracy. The best AIG out of all runs has a testing accuracy of 0.75.

	Train	Test
Mean initial accuracy	0.54	0.55
Min initial accuracy	0.51	0.51
Max initial accuracy	0.64	0.66
Mean final accuracy	0.65	0.66
Min final accuracy	0.51	0.51
Max final accuracy	0.75	0.75

Table 5.1.: Statistics over initial and final training and testing accuracies out of the 42 runs in which we performed an AIG local search. The mean final accuracies are quite low, indicating that we get stuck at suboptimal levels and never recover. A "lucky" initialization with high accuracy could give a head start in the search. However, with the greedy local search we performed this was not the case.

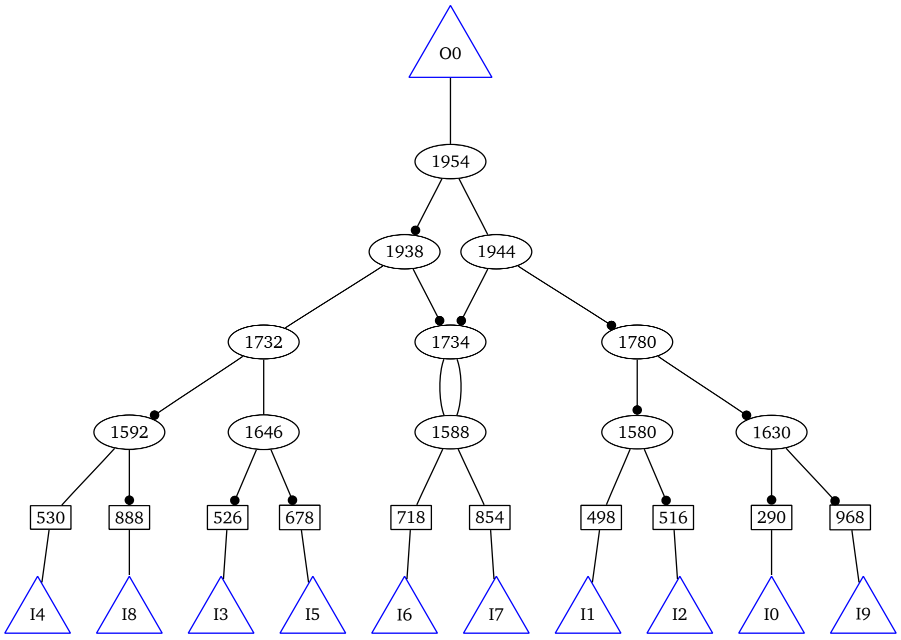


Figure 5.7.: Best AIG found using a greedy local search. Due to a bug in our code, node 1734 takes two identical inputs which is normally not permitted. Nevertheless, it has a testing accuracy of 0.745, just a little over the second best AIG with a testing accuracy of 0.742. It is interesting that an AIG this small still generalizes, albeit with low accuracy.

We can see that we are able to improve the accuracy iteratively, confirming the possibility of using a search algorithm to find AIGs that act as binary classifiers. However, we can also see that the search algorithm has to be improved upon. A "lucky" initialization with high accuracy could give a head start in the search. In Table 5.1, we can see that in at least one run, we started with a testing accuracy of 0.66. Thus, a cheap way to start well could be to try out many random initializations and choose the best to start the search.

With the greedy local search we performed, a higher initial accuracy was of little advantage since the algorithm often got stuck anyway, but with an improved algorithm, trying to start with a higher accuracy already could be one heuristic to try. The greedy local search algorithm is too eager, we quickly get stuck at areas from which we never recover. More advanced search algorithms are thus required. Algorithm 8 does not include reversing changes in the architecture, i.e., going back to previous iterations. Reversing the architecture to how it was just before the variable no_change in Algorithm 8 gets incremented could be a heuristic to save ourselves from getting stuck and thus having to finish the search.

The tiny size of the AIGs is promising. With the architecture chosen for our experiments, a single unpruned AIG only has 68 Bytes of disk size. We arrived at this number by converting an ASCII-encoded AIG file to binary using AIGER and looking at the file size. A small size, together with the lack of need for floating-point matrix computations, make these architectures extremely fast and thus suitable for electronic devices with very little computing power. Furthermore, AIGs are easily translatable to circuits that are implemented on hardware, so the architecture could be much bigger than what we allowed it to be.

The AIGs that the search algorithm found are surprisingly small, especially considering they generalize on a testing set, albeit with relatively low accuracy. It seems the search algorithm has found a way of extreme compression, classification comes down do a handful of pixels. Compression, however, is not our primary goal, and we would rather have a much bigger architecture that has a higher accuracy. The main problem remains the sheer size of the search space.

5.5. Summary

We introduced AIGs and investigated whether they can be used as predictive systems, where we have found that this is possible. We conceptualized our own initialization scheme and chose a greedy local search as search algorithm. The code for our experiments in C++ we wrote from scratch, making use of the AIGER library.

Choosing an architecture that has maximally just 192 AND-gates, the best local search run returned an AIG that has an accuracy of 0.75 which is significantly above chance, meaning learning had taken place. The tiny size, together with the prospect of implementation on hardware make AIGs very promising. Future work should consist of trying to find better search algorithms that are able to find better architectures in less time.

Chapter 6

Comparison of Models

To get an overview in terms of LUT-based models, AIGs and other common ML algorithms, we evaluate training and testing accuracies and model size. The convolutional neural network has the highest accuracy, but is bigger and requires floating-point matrix operations. Given the right choice of architecture and algorithm, LUT-based models are performing quite well, considering reduced size and possible bit-level implementation. The best AIG that we found lacks good accuracy compared to most other models, but is nevertheless very promising with its ultra-small file size of just 68 Bytes.

6.1. Accuracy and size

As in [3], we train various machine learning algorithms on the Binary-MNIST dataset and compare their performance. The training and testing accuracies can be seen in Table 6.1 and testing accuracies are visualized in Figure 6.1. For the LUT-based architectures, we use the format

```
(<num hidden layers>@<LUTs per hidden layer>)
    x<num LUT networks> <arity>LUT.
```

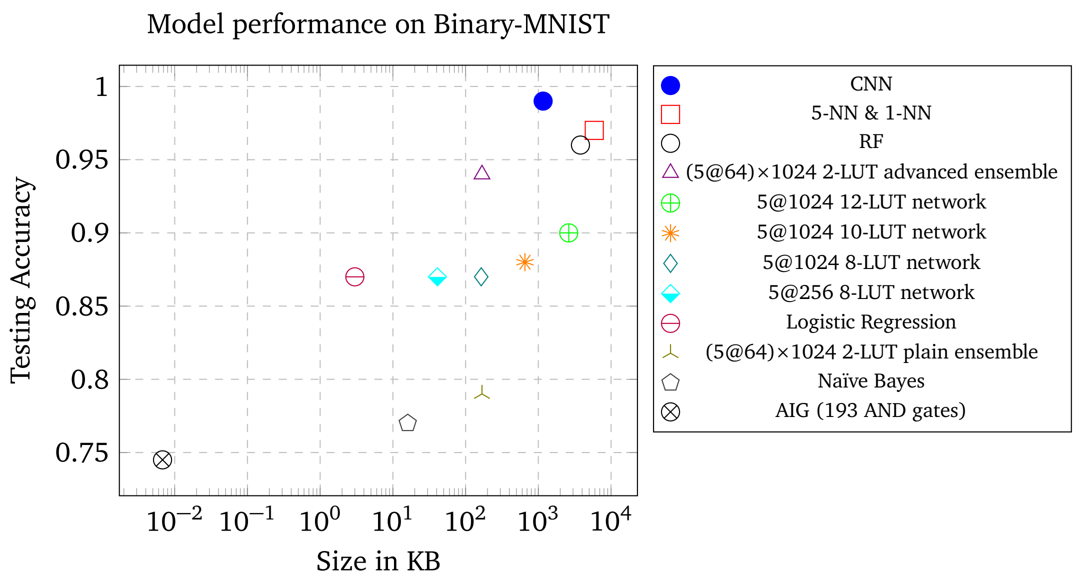


Figure 6.1.: Testing accuracy on the dataset "Binary-MNIST" and size for various models. The *x*-axis that represents the size has a logarithmic scale. The underlying data that was used to generate this plot can be seen in Table 6.1. Our main contributions are a (5@64)×1024 2-LUT advanced ensemble, a binary model with improved accuracy and reduced sized compared to existing approach 5@1024 12-LUT network and an AIG as classifier, a model that is tiny and fast, albeit with lower accuracy.

For the reader wishing to refresh their knowledge about convolutional neural networks, [21] provides a good overview. We implemented the CNN using the Python package PyTorch [22]. The CNN has two convolutional layers with 64 and 32 filters and 2 fully connected layers with 256 and 128 neurons. After the convolutions, we perform max pooling, and we use ReLU as activation function between the layers. The output has a sigmoid activation function. We use the Adam optimizer and go through the dataset only once, i.e., train for one epoch. For all other learning algorithms, we use the Python package Scikit-learn [23]. Except the parameters that we explicitly state, we use the default parameters from Scikit-learn.

We can see that a 12-LUT network beats Logistic Regression and Naïve Bayes. The best performing model is the convolutional neural network. Our results are the same as in [3], take or leave 1%.

Another interesting aspect to consider is how much disk space these models take up. In Table 6.1 the last column shows the size of each model in Kilobytes, the derivation of which can be seen in Appendix A.1. We should note that the sizes presented here are only an approximation and the actual implementation on hardware will always be bigger. In terms of accuracy, the convolutional neural network performs best. It is fairly big, though, and could be made smaller by a technique called "pruning" [24]. An advanced ensemble of 1024 2-LUTs with 5 layers and 64 LUTs per hidden layer has a nice trade-off between performance and size. The AIG is one of the models with the worst accuracies, however the accuracy is still significantly above 0.50 meaning learning had taken place. With just 68 Bytes, it is by far the smallest learned model.

Algorithm	Training accuracy	Testing accuracy	Size
Convolutional Neural Network	0.99	0.99	1164 KB
5-Nearest Neighbors	0.99	0.97	5881 KB
1-Nearest Neighbors	1.00	0.97	5881 KB
Random Forests (10 trees)	1.00	0.96	3797 KB
(5@64)×1024 2-LUT advanced ensemble	0.94	0.94	168 KB
5@1024 12-LUT network	0.99	0.90	2622 KB
5@1024 10-LUT network	0.94	0.88	655 KB
5@1024 8-LUT network	0.89	0.87	164 KB
5@256 8-LUT network	0.88	0.87	41 KB
Logistic Regression	0.87	0.87	3 KB
(5@64)×1024 2-LUT plain ensemble	0.78	0.79	168 KB
AIG (193 AND-gates)	0.75	0.75	0.0068 KB
Naïve Bayes	0.74	0.74	16 KB
Random Guess	0.50	0.50	0.0004 KB

Table 6.1.: Performance of various ML algorithms on the Binary-MNIST dataset. How the sizes of the parameters were computed can be seen in Appendix A.1.

Table 6.1 does not fully do justice to the binary models, i.e., LUT networks and AIGs because binary models have the potential to be implemented without floating-point computation and are thus much faster. An exception would be the advanced LUT ensembling from Section 4.5 which utilizes parameters that are floats. However, apart from these parameters which are not many to begin with (as much as there are individual LUT networks), everything is in binary and thus can be implemented fast. AIGs are excellent in terms of size and speed; they are tiny and can be easily implemented natively on hardware. However, finding a good AIG is hard, i.e., there is need for a much better search algorithm.

6.2. Summary

We evaluated training and testing accuracies for LUT-based models and AIGs, along with other popular machine learning models. The best performing model is the convolutional neural network, however, it is fairly big and requires floating-point computation. The advanced LUT ensemble has an accuracy that is fairly good, along with a much smaller model size compared to the CNN. We saw that the choice of architecture, especially the arity, for LUT-based models is crucial; a bad choice will result in a big size with questionable or no improvement in accuracy.

The AIG has a relatively low accuracy compared to all other models, however, it is extremely small and thus very suitable for implementation on small electronic devices with little compute. It is important to make ourselves clear that Table 6.1 does not discuss inference speed, LUT-based models and AIGs work in binary and have the potential to work much faster than other machine learning models.

Chapter 7

Conclusion

In this thesis, we took two existing circuit designs and investigated the possibility and practicality of using them as binary predictive systems, mostly by experimental approaches, where we used a binarized version of MNIST as training and testing data.

The first circuit design was a network of LUTs. We started with the definition of a single LUT, and then we considered LUT networks that are made of many individual LUTs. We recreated one experiment from the paper that introduced LUTs. Then we took the idea of LUT networks further and devised different schemes to improve the accuracy, where we took three main approaches, i.e., modifying either existing LUT networks or the LUT learning algorithm, enhancing the dataset using feature-engineering and constructing ensembles of many LUT networks. Using LUT ensembles, we could improve the accuracy, together with obtaining a reduced model size, if the architecture is chosen appropriately.

The second circuit design was an AIG. We devised an initialization scheme from scratch and performed a local greedy search to find a suitable architecture. The search algorithm worked, albeit the resulting AIGs had a much lower accuracy than the LUT-based models. However, given the tiny model size of the AIGs, they are very promising to be implemented for small electronic devices or even implementation on hardware. To get an overview, we summarized model accuracies and size for LUT-based models, AIGs and other ML algorithms in a Table.

7.1. Findings and limitations

To run experiments with LUT-based models, we wrote our own code in Python, where the most important external package was NumPy. The code is available on GitHub [13]. We recreated one experiment from the original LUT paper [3] and our results were the same. We found that the number of LUTs per hidden layer can sometimes be reduced, resulting in a smaller size with almost no performance drop. A majority vote and maximizing the layer-wise mean accuracy while training help improving the performance of single LUT networks of low arity.

For higher arity, however, the change does not surpass 1-2%. Feature engineering gave small improvements in accuracy. Ensembling techniques are promising, especially an AdaBoost.M1 ensemble achieved a testing accuracy higher than any previous single LUT network. With a good choice of architecture, we were able to reduce the size of the ensemble significantly while retaining accuracy. We also obtained results that suggest combining ensembling and maximizing the layer-wise mean accuracy is not a good idea.

We wrote the code for our AIG experiments in C++ from scratch, making use of the AIGER library. The code is available on GitHub [20]. We found that the AIG local search algorithm works, and we were able to iteratively improve the accuracy. Because of computational time constraints, we restricted the search space to architectures with a maximum of 192 AND-gates. The best local search run returned an AIG that has an accuracy significantly above chance.

Evaluating training and testing accuracies for LUT-based models, AIGs and other machine learning models, the convolutional neural network had the highest accuracy out of all models. A CNN, however, is bigger and requires floating-point matrix operations. Given the right choice of architecture and algorithm, LUT-based models performed quite well, considering reduced size and possibility of bit-level implementation. The choice of architecture (especially the arity) for LUT-based models is crucial; a bad choice will result in a big size with questionable improvement in accuracy. The best AIG that we found lacked good accuracy compared to most other models, but is nevertheless very promising with its very small file size, thus possibly suitable for implementation on small electronic devices with little computing power.

7.2. Future work

Improving accuracy and reducing size of LUT-based models is key for making them viable for popular use. We have not discussed which percentage of a LUT network is inactive and can thus be pruned. As of now, it is unclear how the initialization of AIGs influences the search process. Better search algorithms have to be investigated to find AIGs faster and with higher accuracy. We have also not discussed multi-class classification which would be possible with both LUT-based models and AIGs using a one-vs-one or one-vs-all approach similar to SVMs [16].

Bibliography

[1] Roy Schwartz et al. “Green ai”. In: *Communications of the ACM* 63.12 (2020), pp. 54–63 (cit. on p. 1).

[2] Di Liu et al. “Bringing AI to edge: From deep learning’s perspective”. In: *Neurocomputing* (2021) (cit. on p. 1).

[3] Satrajit Chatterjee. “Learning and memorization”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 755–763 (cit. on pp. 2, 8, 10, 14, 15, 17, 18, 19, 20, 35, 50, 51, 54).

[4] Stefan Leijnen and Fjodor van Veen. “The neural network zoo”. In: *Multidisciplinary Digital Publishing Institute Proceedings*. Vol. 47. 1. 2020, p. 9 (cit. on p. 6).

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016 (cit. on p. 6).

[6] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. “Neural architecture search: A survey”. In: *The Journal of Machine Learning Research* 20.1 (2019), pp. 1997–2017 (cit. on p. 6).

[7] Günter Klambauer and Sepp Hochreiter. “Deep Learning and Neural Networks I”. Unpublished lecture notes. 2019 (cit. on p. 6).

[8] Zoubin Ghahramani. “Unsupervised learning”. In: *Summer school on machine learning*. Springer. 2003, pp. 72–112 (cit. on p. 7).

[9] Seppo Linnainmaa. “Taylor expansion of the accumulated rounding error”. In: *BIT Numerical Mathematics* 16.2 (1976), pp. 146–160 (cit. on p. 7).

[10] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv preprint arXiv:1609.04747* (2016) (cit. on p. 7).

[11] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cit. on pp. 15, 16).

[12] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2 (cit. on p. 17).

[13] Bernhard Gstrein. *LUT Github Repo*. https://github.com/texmex76/master. 2022 (cit. on pp. 17, 18, 54).

[14] Irwin Sobel, R Duda, and P Hart. “Sobel-Feldman Operator”. In: *researchgate.net* (2014) (cit. on p. 28).

[15] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. “Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors)”. In: *The annals of statistics* 28.2 (2000), pp. 337–407 (cit. on p. 31).

[16] Christopher M Bishop and Nasser M Nasrabadi. “Pattern recognition and machine learning”. In: vol. 4. 4. Springer, 2006, pp. 338–339 (cit. on pp. 35, 55).

[17] The Free Encyclopedia Wikipedia. *And-inverter graph*. https://en.wikipedia.org/wiki/And-inverter_graph. Accessed 2022-05-03. 2022 (cit. on p. 41).

[18] Daphne Koller and Nir Friedman. “Probabilistic graphical models: principles and techniques”. In: MIT press, 2009, p. 1155 (cit. on p. 44).

[19] Armin Biere. *The AIGER And-Inverter Graph (AIG) Format Version 20071012*. Tech. rep. 07/1. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, 2007 (cit. on p. 46).

[20] Bernhard Gstrein. *AIG GitHub repo*. https://github.com/texmex76/aig-classifier. 2022 (cit. on pp. 46, 55).

[21] Rikiya Yamashita et al. “Convolutional neural networks: an overview and application in radiology”. In: *Insights into imaging* 9.4 (2018), pp. 611–629 (cit. on p. 51).

[22] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (cit. on p. 51).

[23] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 51).

[24] Pavlo Molchanov et al. “Pruning convolutional neural networks for resource efficient inference”. In: *arXiv preprint arXiv:1611.06440* (2016) (cit. on p. 52).

Appendix

A.1. Derivation of parameter sizes in Section 6.1

The actual size on disk of a machine learning model heavily depends on its implementation, so the sizes given in Table 6.1 are merely an approximation. Mostly we state the size of all parameters, if available. In those cases, the actual model size will always be bigger because machine instructions are also required that specify what to do with the parameters.

For nearest neighbors, the whole dataset is required to be available for inference, so we just state the size of Binary-MNIST. Given binary features $\boldsymbol{X}$ with 60000 rows and 784 columns and binary labels $\boldsymbol{y}$ with 10000 rows and one column, we obtain 5881 KB.

The size of the LUT networks is computed using Equation3.4.

For a convolutional neural network, we consider the total number of all trainable parameters (i.e., all weight entries) which we obtain from the PyTorch model in one line of code. We assume 32-bit floats.

We determine the size of all parameters of logistic regression and naïve Bayes by going through all methods the respective Scikit-learn object has and adding everything up.

Since Random Forests are made up of multiple decision trees, we go through each of them. In Scikit-learn a decision tree has the method `tree_`, and we consider what it contains for the size.

For random guess, we just take into account the random seed. If we use a 32-bit integer as random seed, then the parameter size becomes 4 Bytes.

The size of the AIG we computed by passing a text-based aag-file to the program "aigtoaig" from the software AIGER. The output file is an AIG in binary, and we read off its size. We have to keep in mind that the binary AIG file we obtain might even be too big and could be implemented smaller.

A.2. Best valid AIG found

The best AIG found is visualized in Figure 5.7, but due to a bug in our code, one AND node takes two identical inputs, which is normally not allowed, hence we visualize the best valid AIG in Figure A.1. The testing accuracy of this AIG is 74.2% on Binary-MNIST.

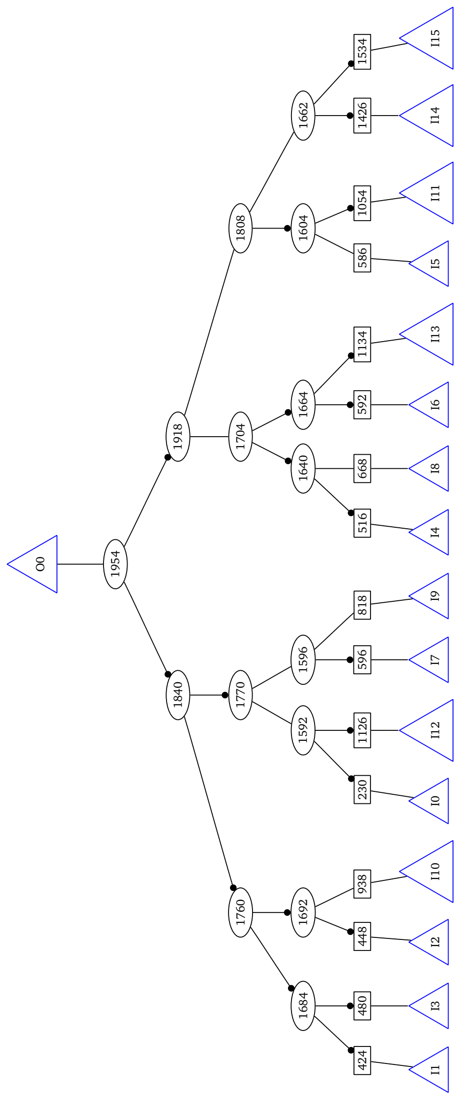


Figure A.1.: Best AIG with a valid architecture that was found using greedy local search with 74.2% testing accuracy on Binary-MNIST.