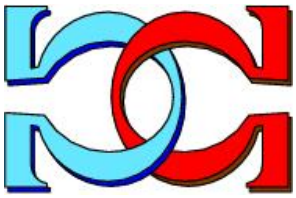

CDMTCS
Research
Report
Series

What Neural Networks Are (Not) Good For?

Cristian S. Calude[1], Shahrokh Heidari[1], Joseph Sifakis[2]

[1]School of Computer Science, University of Auckland, New Zealand
[2]Verimag Laboratory, University Grenoble Alpes, France

CDMTCS-556
July 2021

Centre for Discrete Mathematics and
Theoretical Computer Science

What Neural Networks Are (Not) Good For?

Cristian S. Calude[1], Shahrokh Heidari[1], Joseph Sifakis[2]

[1]School of Computer Science, University of Auckland, New Zealand
[2]Verimag Laboratory, University Grenoble Alpes, France

August 10, 2021

Abstract

Neural Networks (NNs) are essential components of intelligent systems because they produce efficient solutions to problems of overwhelming complexity for conventional computing methods. There are lots of papers showing that NNs can approximate a wide variety of functions, but comparatively very few discuss their limitations. To illustrate the role played by information coding in NN computations we define and study sensitive and robust Boolean functions. We also prove that an exponential large set of functions in the first group are exponentially difficult to compute by multiple-layer perceptrons (hence incomputable by single-layer perceptrons) and a comparatively large set of functions in the second one, but not all, are computable by single-layer perceptrons. This suggests that the success of NNs, or lack of it, are in part determined by properties of the learned data sets. Finally we use polynomial threshold perceptrons to compute all Boolean functions with quantum annealing and present in detail a QUBO computation on the D-Wave Advantage.

1 Introduction

NN is a computing paradigm that radically differs from conventional computing. NNs can learn how to solve a problem through training with big data representing an I/O relation: they produce empirical data-based knowledge different from model-based knowledge generated from the execution of fully crafted and understood algorithms. Model-based approaches allow explainability and are amenable to rigorous analysis while data-based techniques are hard to interpret and understand. NNs work amazingly well in a variety of areas, like automatic speech recognition, image recognition, natural language processing, drug discovery and toxicology, automatic game playing, etc. They also have an increasing number of applications in software and systems engineering. Despite their empirical successes, very little is understood about how machine learning (ML) models accomplish their tasks. Explainable-AI is an active research topic focusing on the generation of models explicating the behaviour of AI-enabled systems [1]. The distinction between data-based and model-based knowledge is essential in systems engineering, where NNs are integrated in critical systems whose failures can be harmful for their environment. For instance, using end-to-end ML-enabled solutions in autonomous systems, such as self-driving cars, has been the object of hot debates as it is practically impossible to estimate their trustworthiness [2, 3]. The trend moves toward intelligent systems that adequately combine data-based and model-based components and take the best from each approach by determining trade-offs between performance and trustworthiness. This trend is also boosted by the striking similarity between these two computing paradigms and the two types of human thinking [4]: fast non-conscious thinking relies on a data-based empirical learning process while conscious slow thinking is the result of a procedural explainable reasoning. Human mind combines in an admirable way the two types of thinking to

produce knowledge and solve problems. Hence, it is natural to investigate how the two complementary computing paradigms can be combined in the best possible manner to address the machine intelligence challenge.

It is well understood that data-based empirical learning should be robust to data variations and guaranteeing this property is a non trivial problem. Intuitively, robustness can be conceived as a metric property of the learned data sets such that the meanings of very "close" representations do not significantly differ. While there is some understanding of robustness, there is a lack of comprehension of its invariances and determinants.

If we recognise that NNs work well in some cases, but not in all, then, natural questions arise: What NNs are good for? When should we use them and when not? How significant are the cases when NN's do not work? Are they practically irrelevant or just esoteric cases interesting only from a theoretical point of view? Is it possible to distinguish between problems where the application of NNs is obvious and problems where model-based solutions are more adequate? How can we combine these two types of solutions?

Our work is motivated by the observation that NNs seem to be more adequate for the classification of robust massive information: small modification of the input will not drastically affect the classification result. This typically happens for NNs dealing with sensory information and implementing perception functions like in medical image analysis or face recognition. However, there are applications where using NNs hardly makes sense. For example, is it possible to train a NN to check a given property (even a syntactic one) from the analysis of the source code of programs? The answer is probably no because software correctness is very sensitive to small changes of the source code. Moreover, the relationship between software syntax and its meaning defined by the operational semantics of the programming language can be deep and intricate. Similar issues can arise when we may try to use NNs as monitors for detecting failures of software systems. How much confident can we be in their verdicts obtained after a sufficiently long training with testing data sets that distinguish between accepted and non-accepted test sequences? Our confidence in such NN oracles would decrease as their sensitivity to input change increases.

Another question that naturally rises is how the coding of information may impact the complexity of the learning process. Considering again the previous example, the same program can admit a large variety of semantically equivalent representations at different abstraction levels e.g. source code, object code or even in the form of a transition system if the program is finite state. How the adopted type of representation affects the complexity of the learning process? Conversely, consider data sets with properties that are easy to learn. Can transformations of their representation by "weird" scrambling functions affect robustness and thus increase the learning complexity? For instance, if convexity of data sets is important for learning a given property, what is the complexity for representations obtained using codes that jeopardise their convexity?

In this paper we study only NN's computations of Boolean functions. To illustrate the role played by information coding in NN computations we define sensitive and robust Boolean functions and prove that an exponential large set of functions in the first group are exponentially difficult to compute by multiple-layer perceptrons (hence incomputable by single-layer perceptrons) and a comparatively large set of functions in the second one, but not all, are computable by single-layer perceptrons. This suggests that the success of NNs, or lack of it, are in part determined by properties of the learned data sets.

The paper starts with a minimal amount of notation, defines the notions of sensitive and robust Boolean functions and introduces three infinite classes of Boolean functions $PARITY_n, R^1_n, R^2_n$ used as case studies. We move to limitations of single-layer and multi-layer perceptrons in computing the sensitive functions $PARITY_n$, which are the most difficult to compute. In contrast we prove that the robust functions like R^1_n, R^2_n are computable by single-layer perceptrons. Then we give more general results, including

the fact that the set of sensitive functions which are computed by multi-layer perceptrons (MLPs) with a single hidden layer and an exponential number of hidden units is exponentially larger than the set of threshold functions.

Finally, we use polynomial threshold perceptrons to compute all Boolean functions with quantum annealing and present in detail a QUBO computation of $PARITY_4$ on the D-Wave Advantage. We end with a few conclusions and two open questions.

2 Classes of Boolean functions

The set of binary strings of length n is denoted by $\{0,1\}^n$. Bits will be denoted by x, y and bit-strings by $\mathbf{x}, \mathbf{y}$: depending on the context we will write $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ or $\mathbf{x} = x_1 x_2 \ldots x_n$. The Boolean operations will be denoted by $^-$ (negation), $\vee$ (disjunction) and omitted $\cdot$ (conjunction). The set of reals is denoted by $\mathbb{R}$; a vector of n real-valued components is denoted by $\mathbf{w}$.

In this paper we study Boolean functions of $n > 1$ variables $f : \{0,1\}^n \to \{0,1\}$, shortly, *functions*. The true/false points of a function f are denoted by $T(f) = \{\mathbf{x} \in \{0,1\}^n \mid f(\mathbf{x}) = 1\}$ and $F(f) = \{\mathbf{x} \in \{0,1\}^n \mid f(\mathbf{x}) = 0\}$, respectively.

Every function $f : \{0,1\}^2 \to \{0,1\}$ can be naturally extended to $n > 2$ variables in the following way: $f_2 = f$ and $f_n : \{0,1\}^n \to \{0,1\}$, $f_n(x_1, x_2, \ldots, x_n) = f_2(f_{n-1}(x_1, \ldots, x_{n-1}), x_n)$. In this way we get the functions of n variables OR_n, XOR_n but not $XNOR_n$. We denote by R_n^1, $R_n^2 : \{0,1\}^n \to \{0,1\}$ defined by $R_n^1(\mathbf{x}) = OR_n(\mathbf{x})$ and $R_n^2(\mathbf{x}) = x_1$ and by $PARITY_n : \{0,1\}^n \to \{0,1\}$ the function

$$PARITY_n(\mathbf{x}) = \begin{cases} 1, & \text{if the number of 0's in } \mathbf{x} \text{ is odd,} \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Lemma 1. *For every $n > 1$, $PARITY_n = XOR_n$ for even n and $PARITY_n = \overline{XOR}_n$ for odd n.*

Proof. It is seen that $PARITY_2 = XOR$ and $PARITY_{n+1}(x_1, x_2, \ldots, x_{n+1}) = \overline{XOR}(PARITY_n(x_1, x_2, \ldots, x_n), x_{n+1})$. □

In what follows a function f will be represented in *full disjunctive normal form* [5, p. 123]. The number of true points of f will be denoted by $\#T(f)$. By d we denote the *Hamming distance* between strings of length n: $d(\mathbf{x}, \mathbf{y})$ is the number of positions i such that $x_i \neq y_i$.

Example 1. *If $T(f) = \{111, 100, 001\}$, then $d(\mathbf{x}, \mathbf{y}) = 2$ for all distinct $\mathbf{x}, \mathbf{y} \in T(f)$. The Boolean function $PARITY_3$ has this property, but R_3^1 and R_3^2 do not have it.*

3 Computing with single-layer perceptrons

A binary classifier is a function which decides whether or not an input belongs to a specific set. A *linear threshold computing unit* or *(single-layer) perceptron* [6, p. 7] is a function $P_{\theta,\mathbf{w}} : \mathbb{R}^n \to \{0,1\}$, depending on two parameters, a threshold $\theta \in \mathbb{R}$ and a vector of n weights $\mathbf{w} = (w_1, w_2, \ldots, w_n) \in \mathbb{R}^n$, defined by

$$P_{\theta,\mathbf{w}}(\mathbf{x}) = \begin{cases} 1, & \textit{if } \sum_{i=1}^n w_i x_i \geq \theta, \\ 0, & \textit{otherwise}. \end{cases} \tag{2}$$

Example 2. *For every $n > 1$, the functions R_n^1, R_n^2 are threshold functions.*

Proof. We have: $R_n^1 = P_{\frac{1}{2},(1,1,\ldots,1)}$ and $R_n^2 = P_{\frac{1}{2},(n,-1,\ldots,-1)}$. □

Theorem 1. [7, Theorem 3.7] *A function f of $n > 1$ variables is a threshold function if and only if for every positive integer k, for every sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k \in T(f))$ and every sequence $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_k \in F(f)$ we have $\sum_{i=1}^k \mathbf{x}_i \neq \sum_{i=1}^k \mathbf{y}_i$.*

Example 3. *It is known that XOR and XNOR are not threshold functions [8]. The functions $PARITY_n$ cannot be computed by perceptrons too.*

Proof. Consider the following four vectors in $\{0,1\}^n$: $\mathbf{x}_1 = 0^{n-2}01$, $\mathbf{x}_2 = 0^{n-2}10$, $\mathbf{y}_1 = 0^{n-2}00$, $\mathbf{y}_2 = 0^{n-2}11$. For every n we have $PARITY_n(\mathbf{x}_1) = PARITY_n(\mathbf{x}_2)$, $PARITY_n(\mathbf{y}_1) = PARITY_n(\mathbf{y}_2)$, $PARITY_n(\mathbf{x}_1) \neq PARITY_n(\mathbf{y}_1)$ and $\mathbf{x}_1 + \mathbf{x}_2 = \mathbf{y}_1 + \mathbf{y}_2$, so the conclusion follows from Theorem 1. □

4 Sensitive vs. robust functions

In the previous section we have proved that R_n^1 and R_n^2 are computable by perceptrons, but $PARITY_n$ is not. What is the reason for these different results? What make some functions, but not all, computable by perceptrons, and more generally, easy computable by NNs?

As it was pointed out in [9], this phenomenon is not surprising, we just need to ask the question differently. First, how can NNs approximate functions well in practice, when the set of possible functions is exponentially larger than the set of practically possible NNs? Indeed, there are 2^{2^n} different functions of n variables, so an NN implementing a generic function in this class requires at least 2^n bits to describe, that is, more bits than there are atoms in our universe if $n > 260$ (not a large number of variables for practical applications). A similar analysis points to an exponential difference between the number of different functions of a fixed number of variables (double exponential) and the number of polynomial NNs.

To provide an answer to the question above, let us first note the difference between the functions $PARITY_n$, on one side, and the functions R_n^1 and R_n^2, on the other side. For $PARITY_n$, a single bit in the input $\mathbf{x}$ can flop the value of $PARITY_n(\mathbf{x})$ from 0 to 1 or vice-versa. In contrast, R_n^1 and R_n^2 are more robust for variations of their inputs. Indeed, for R_n^1, if two inputs $\mathbf{x}, \mathbf{y}$ contain each an 1, say $x_i = 1$ and $y_j = 1$, then $R_n^1(\mathbf{x}) = R^1(\mathbf{y})$, and only $R_n^1(0, \ldots 0) = 0$; for R_n^2 we have $R_n^2(x_1, x_2, \ldots, x_n) = R_n^2(x_1, y_2, \ldots, y_n)$, for all $x_1, x_2, \ldots, x_n, y_2, \ldots, y_n \in \{0,1\}$.

This suggests that sensitivity vs. robustness can cause differences in NNs computability or incomputability of functions, so we will propose a model for them. Informally, a function is sensitive if a "small variation" in the input will determine a jump in the values of the function from 0 to 1 or vice-versa; a function which is not sensitive will be called robust.

Definition 1. *Fix a function f of $n > 1$ variables. We say that f is*

(a) strongly sensitive *if for all $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$ with $d(\mathbf{x}, y) = 1$, we have $f(\mathbf{x}) = \overline{f(\mathbf{y})}$,*

(b) sensitive *if for all $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$ with $d(\mathbf{x}, \mathbf{y}) = 1$ and $f(\mathbf{x}) = 1$, we have $f(\mathbf{y}) = 0$.*

It is clear that strong sensitiveness implies sensitiveness, but the converse implication is false. Indeed, the function $f(00) = 1, f(\mathbf{x}) = 0$, for $\mathbf{x} \neq 00$ is sensitive as $d(00, 01) = d(00, 10) = 1, f(00) = 1 \neq f(01) = f(10)$, but not strongly sensitive as $f(01) = f(11) = 0, d(01, 11) = 1$.

Next we show that Example 1 is in fact more general:

Proposition 1. *Assume that f is a function with $n > 1$ variables. Then the following two conditions are equivalent:*

(a) The function f is sensitive.

(b) For every distinct $\mathbf{x}, \mathbf{y} \in T(f)$, $d(\mathbf{x}, \mathbf{y}) \geq 2$.

Proof. For the direct implication we assume by absurdity the existence of $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$ with $d(\mathbf{x}, \mathbf{y}) = 1$ and $f(\mathbf{x}) = f(\mathbf{y}) = 1$. Then, by (a) $f(\mathbf{x}) = \overline{f(\mathbf{y})}$, a contradiction. Conversely, if we assume by absurdity that there exist $\mathbf{x} \neq \mathbf{y}, \mathbf{x}, \mathbf{y} \in T(f)$ such that $d(\mathbf{x}, \mathbf{y}) = 1$, then by (b), $d(\mathbf{x}, \mathbf{y}) \geq 2$, a contradiction. □

Corollary 1. *No (strongly) sensitive function is a threshold function.*

Proof. Consider a sensitive function f of $n > 1$ variables. Let us take $\mathbf{x}', \mathbf{x}'' \in T(f)$, hence by Proposition 1, $d(\mathbf{x}', \mathbf{x}'') \geq 2$. Then, there exist $1 \leq i < j \leq n$, $\mathbf{u}, \mathbf{v}, \mathbf{z}$ such that $\mathbf{x}' = \mathbf{u}x_i\mathbf{v}x_j\mathbf{z}$, $\mathbf{x}'' = \mathbf{u}\bar{x}_i\mathbf{v}\bar{x}_j\mathbf{z}$. We now choose $\mathbf{y}' = \mathbf{u}\bar{x}_i\mathbf{v}x_j\mathbf{z}$, $\mathbf{y}'' = \mathbf{u}x_i\mathbf{v}\bar{x}_j\mathbf{z}$. We note that $d(\mathbf{x}', \mathbf{y}') = d(\mathbf{x}'', \mathbf{y}'') = 1$ and $\mathbf{x}' + \mathbf{x}'' = \mathbf{y}' + \mathbf{y}''$, $\mathbf{x}', \mathbf{x}'' \in T(f)$, $\mathbf{y}', \mathbf{y}'' \in F(f)$. The conclusion follows from Theorem 1. □

MLPs have more computational power than perceptrons. An MLP consists of an input layer, intermediate (hidden) layers and an output layer [10], see Figure 1. An MLP $P : \mathbb{R}^{n_1} \to \mathbb{R}^{n_L}$ is defined by L mappings acting on a sequence of spaces $(\mathbb{R}^{n_1}, \mathbb{R}^{n_2}, \ldots, \mathbb{R}^{n_L})$ [11], $P^1 : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}, P^2 : \mathbb{R}^{n_2} \to \mathbb{R}^{n_3}, \ldots, P^{L-1} : \mathbb{R}^{n_{L-1}} \to \mathbb{R}^{n_L}$, where each P^i $(1 \leq i < L)$ consists of j perceptrons defined as:

$$P^{i,j}_{\theta^i_j, \mathbf{w}^i_j}(\mathbf{a}^{i-1}) = \begin{cases} 1, & \text{if } \sum_{k=1}^{n_i} w^i_{jk} a^i_k \geq \theta^i_j, \\ 0, & \text{otherwise}, \end{cases}$$

such that a) $\mathbf{a}^0 = \mathbf{x} \in \mathbb{R}^{n_1}$, $\mathbf{a}^1 \in \mathbb{R}^{n_2}, \ldots$, $\mathbf{a}^{L-1} \in \mathbb{R}^{n_L}$, b) $\mathbf{w}^i \in \mathbb{R}^{(n_{i+1} \times n_i)}$ denotes the weight matrix connecting i^{th} layer to $(i+1)^{th}$ layer; $\mathbf{w}^i_j$ is the j^{th} row of matrix $\mathbf{w}^i$, c) $\theta^i \in \mathbb{R}^{(n_{i+1} \times 1)}$ is the threshold vector and θ^i_j is the j^{th} row of this vector, d) n_i is the number of units in the ith layer $(1 \leq i < L)$ and n_L is the number of units in the output layer. The mappings above feed the input patterns into the hidden layers to categorise different classes in the output layer. When we have just one unit in the output layer, the MLP is called binary classifier.

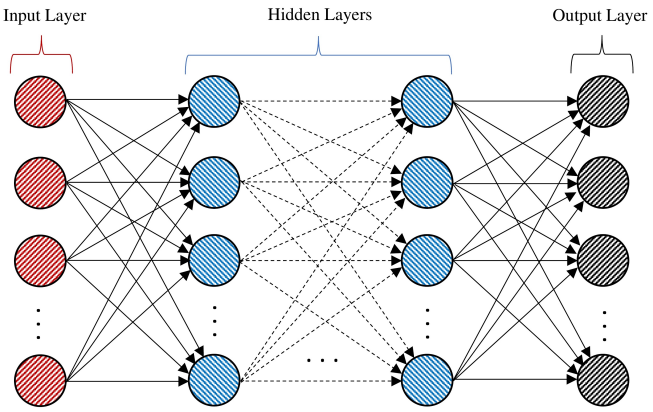

Figure 1: MLP with an input layer, two hidden layers and an output layer

Example 4. *Figure 2 exemplifies a three-layer MLP with one hidden layer.*

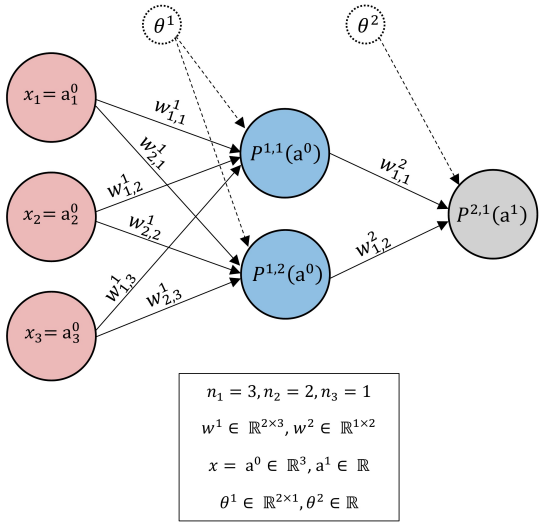

Figure 2: A simple example of an MLP with a hidden layer

Theorem 2. [Universality Theorem 7.1 [7, p. 74-83]] *Every function of $n > 1$ variables can be computed by an MLP with a single hidden layer.*

Theorem 3. *Every sensitive function of $n > 1$ variables with $k = \#(T(f))$ is computed by an MLP with a single hidden layer and exactly k hidden units.*

Proof. By Theorem 2, every function can be computed by an MLP with a hidden layer by mapping each $\mathbf{x} \in T(f)$ to a unit in the hidden layer [12, p. 3]; as $k = \#(T(f))$, an MLP with a single hidden layer can compute f with k hidden units.

By sensitivity and Proposition 1, for all $\mathbf{x}, \mathbf{y} \in T(f)$ we have $d(\mathbf{x}, \mathbf{y}) \geq 2$, so points in $(T(f)$ cannot be combined to reduce the disjunctive normal form [13, p. 162]), hence the number of hidden units cannot be reduced. As each true point is a node in the hidden layer, the hidden layer cannot have less than k units [12]. □

Corollary 2. *Every function $PARITY_n$ with $n > 1$ is computed by an MLP with a single hidden layer and exactly 2^{n-1} hidden units.*

Comment 1. In [9] a similar result was proved for a more complicated function: n variables cannot be multiplied using fewer than 2^n perceptrons in an MLP with a single hidden layer.

Theorem 4. *For every $n > 2$ there exist $2^{2^{n-2}} - 2$ sensitive, not strongly sensitive functions which are computed by MLPs with a single hidden layer and an exponential number of hidden units.*

Proof. Take a strongly sensitive function f (for which $\#T(f) = 2^{n-1}$) and remove from $T(f)$ a subset of P containing 2^{n-2} points. Then for every non-empty subset $S \subset P$ we consider the function f_S whose set of true points is $T(f_S) = (T(f) \setminus P) \cup S$. Every function f_S is sensitive by Proposition 1, not strongly sensitive by Corollary 2, and as $\#(T(f_S)) > 2^{n-2}$, every MLP with a single hidden layer that computes it has an exponential number of hidden units by Theorem 3. Furthermore, the number of all functions f_S is at least $2^{2^{n-2}} - 2$. □

Corollary 3. *The set of sensitive functions of $n > 2$ variables which are computed by MLPs with a single hidden layer and an exponential number of hidden units is exponentially larger than the set of threshold functions.*

Proof. The set of threshold functions of $n > 1$ variables has less than 2^{n^2} functions, [7, Theorem 4.3], while by Theorem 4, the set of functions which are computed by MLPs with a single hidden layer and an exponential number of hidden units has at least $2^{2^{n-2}} - 2$ functions.

□

Which functions are strongly sensitive? We first prove some invariants of strongly sensitive functions. The following results follows directly by the definition of strong sensitivity.

Lemma 2. *If f is a strongly sensitive function of $n > 1$ variables, then the functions $\bar{f}$, f_π (where π is a permutation of the set $\{1, 2, \ldots, n\}$) defined by $\bar{f}(\mathbf{x}) = \overline{f(\mathbf{x})}$, $f_\pi(x_1 x_2 \ldots x_n) = f(x_{\pi(1)} x_{\pi(2)} \ldots x_{\pi(n)})$ are also strongly sensitive.*

Proposition 2. *Let f be a function of $n > 1$ variables. The following statements are equivalent:*

(a) f is strongly sensitive.

(b) The function $g_f(\mathbf{x}, x_{n+1}) = XNOR(f(\mathbf{x}), x_{n+1})$ is strongly sensitive.

(c) The function $h_f(\mathbf{x}, x_{n+1}) = XOR(f(\mathbf{x}), x_{n+1})$ is strongly sensitive.

Proof. Assume first that (a) is true and take $\mathbf{x}x_{n+1}, \mathbf{y}y_{n+1} \in \{0,1\}^{n+1}$ such that $d(\mathbf{x}x_{n+1}, \mathbf{y}y_{n+1}) = 1$. Permuting the variables and using Lemma 2 we can assume that $x_{n+1} = y_{n+1}$ and $d(\mathbf{x}, \mathbf{y}) = 1$. From the sensitivity of f it follows that $f(\mathbf{x}) = \overline{f(\mathbf{y})}$, hence we have:

$$g_f(\mathbf{x}, x_{n+1}) = f(\mathbf{x})\ x_{n+1} \vee \overline{f(\mathbf{x})}\ \overline{x_{n+1}} = \overline{f(\mathbf{y})}\ x_{n+1} \vee f(\mathbf{y})\ \overline{x_{n+1}} = \overline{g_f(\mathbf{y}, x_{n+1})},$$

so g_f is strongly sensitive.

Next assume that (b) is true and take $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$ such that $d(\mathbf{x}, \mathbf{y}) = 1$. By (b) $g_f(\mathbf{x}, x_{n+1}) = \overline{g_f(\mathbf{y}, x_{n+1})}$ because $d(\mathbf{x}x_{n+1}, \mathbf{y}x_{n+1}) = 1$. Indeed, if by absurdity $f(\mathbf{x}) \neq \overline{f(\mathbf{y})}$, then $g_f(\mathbf{x}, x_{n+1}) = f(\mathbf{x})\ x_{n+1} \vee \overline{f(\mathbf{x})}\ \overline{x_{n+1}} = f(\mathbf{y})\ x_{n+1} \vee \overline{f(\mathbf{y})}\ \overline{x_{n+1}} = g_f(\mathbf{y}, x_{n+1}) \neq \overline{g_f(\mathbf{y}, x_{n+1})}$, a contradiction.

Finally, by Lemma 2, g_f is strongly sensitive if and only if $h_f = \overline{g_f}$ is strongly sensitive, that is, (b) is equivalent to (c). □

The following equalities are easy to verify:

Lemma 3. *The following relations are true for all $x, y, z \in \{0,1\}$:*

1. $XOR(x, XOR(y, z)) = \overline{XOR}(x, \overline{XOR}(y, z))$,
2. $XOR(x, \overline{XOR}(y, z)) = \overline{XOR}(x, XOR(y, z))$.

Theorem 5. *The functions $PARITY_n$ and $\overline{PARITY_n}, n > 1$ are the only strongly sensitive functions.*

Proof. Clearly, $PARITY_n$ is strongly sensitive; by Lemma 2, $\overline{PARITY_n}$ is also strongly sensitive.

If f_n is a strongly sensitive function of $n > 2$ variables, then $f_n(x_1, \ldots, x_{n-1}, x_n) = XOR(f_{n-1}, x_n)$, where $f_{n-1}(x_1, \ldots, x_{n-1}) = f_n(x_1, \ldots, x_{n-1}, 0)$. By Proposition 2, f_{n-1} is also strongly sensitive. In this way we get the sequence of strongly sensitive functions $f_{n-1}, f_{n-2}, \ldots, f_2$ satisfying the relations

$$f_i(x_1, \ldots, x_{i-1}, x_i) = XOR(f_{i-1}, x_i). \tag{3}$$

Out of all 16 functions of 2 variables only two, $PARITY_2, \overline{PARITY_2}$, are strongly sensitive. Going backwards via (3) and using Proposition 2 we infer that every strongly sensitive function of $n > 2$ variables can be obtained by $n-1$ compositions of XOR and $\overline{XOR}$. From Lemma 3 we deduce that in the set of 2^{n-1} functions obtained from all compositions of XOR and $\overline{XOR}$ there are only two distinct functions, $PARITY_n$ and $\overline{PARITY_n}$. □

Comment 2. Every strongly sensitive function f of $n > 1$ variables has $\#(F(f)) = 2^{n-1}$.

From Corollary 1 we deduce that every threshold function is not sensitive, that is, "there exist $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$ such that $d(\mathbf{x}, y) = 1$ we have $f(\mathbf{x}) = f(\mathbf{y})$"; this property seems to be a weak form of robustness. The condition "for every $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$ such that $d(\mathbf{x}, y) = 1$ we have $f(\mathbf{x}) = f(\mathbf{y})$" is too strong, as it is satisfied only by the constant functions. A better definition is:

Definition 2. *The function f is* robust *if for every $\mathbf{x} \in \{0,1\}^n$ there exists $\mathbf{y} \in \{0,1\}^n$ such that $d(\mathbf{x}, y) = 1$ we have $f(\mathbf{x}) = f(\mathbf{y})$.*

Example 5. *Every function f with $T(f) = \{\mathbf{x}, \mathbf{y}\}$ and $d(\mathbf{x}, \mathbf{y}) = 1$ is robust and threshold.*

Proposition 3. *The functions R_n^1 and R_n^2 are robust and threshold.*

Proof. If $\mathbf{x} \in \{0,1\}^n$ with $R_n^1(\mathbf{x}) = 1$ we can find an $\mathbf{y} \in \{0,1\}^n$ such that $d(\mathbf{x}, \mathbf{y}) = 1$ and $R_n^1(\mathbf{y}) = 1$. If $\mathbf{x}$ contains only one 1, then $\mathbf{y}$ can be obtained by from $\mathbf{x}$ by replacing a single bit 0 with 1; otherwise $\mathbf{y}$ can be obtained from $\mathbf{x}$ by replacing a single bit 1 with 0. If $R_n^1(\mathbf{x}) = 0$, then $\mathbf{x} = 0^n$, so we take $y = 10^{n-1}$: $d(\mathbf{x}, \mathbf{y}) = 1$ and $R_n^1(\mathbf{y}) = 1$. If $\mathbf{x} \in \{0,1\}^n$ with $R_n^2(\mathbf{x}) = x_1 = 1$, then every $\mathbf{y} \in \{0,1\}^n$ such that $y_1 = 1$ and $d(\mathbf{x}, \mathbf{y}) = 1$ satisfies $R_n^2(\mathbf{y}) = 1$; the case $R_n^2(\mathbf{x}) = 0$ is similar. By Example 2, R_n^1 and R_n^2 are threshold functions. □

A function is *monotone* in case for every $\mathbf{x} \leq \mathbf{y}$ (that is, for every $1 \leq i \leq n$ we have $x_i \leq y_i$) we have $f(\mathbf{x}) \leq f(\mathbf{y})$.

Example 6. *Monotone non-constant functions are robust, but not all of them are computable by perceptrons.*

Proof. The set of threshold functions of $n > 1$ variables has less than 2^{n^2} functions, [7, Theorem 4.3], which is a smaller subset of the set of monotone functions whose cardinality is the Dedekind number $D_n \geq 2^{\binom{n}{\lfloor n/2 \rfloor}}$, [14], hence.

□

5 Quantum annealing computation of polynomial threshold perceptrons

A polynomial threshold unit is a generalisation of a perceptron in which the linear threshold is replaced by a polynomial threshold, see [15, p. 5]. In detail, for a positive integer n we define the set $[n] = \{1, 2, \ldots, n\}$ and the multi-set $[n]^m$ containing all possible selections with repetitions of at most m objects from $[n]$.

Example 7. *For $n = 3$ and $m = 2$ we have* $[3]^2 = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,1\}, \{2,2\}, \{3,3\}, \{1,2\}, \{1,3\}, \{2,3\}\}$.

Consider a multi-set $S \in [n]^2$ and an input vector $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$. By x_S we denote the product of x_i for $i \in S$. For instance, we have $x_\emptyset = 1$, $x_{\{2,3\}} = x_{\{3,2\}} = x_2x_3$, $x_{\{1,1\}} = x_1^2$.

A polynomial threshold perceptron is defined by a vector parameter $\mathbf{w}_S$, with $S \in [n]^m$. The function $P^m_{\mathbf{w}_S} \colon \mathbf{R}^n \to \{0,1\}$ computed by a polynomial threshold perceptron with parameter $\mathbf{w}_S$ is

$$P^m_{\mathbf{w}_S}(\mathbf{x}) = \begin{cases} 1, & \text{if } \sum_{T \in S} w_T x_T \geq 0, \\ 0, & \text{otherwise}. \end{cases} \tag{4}$$

Example 8. *For $n = 3$, $m = 2$ and $\mathbf{x} = (x_1, x_2, x_3) \in \mathbb{R}^n$ the weighted sum of inputs for the polynomial threshold perceptron has the form:*

$$w_\emptyset + w_1x_1 + w_2x_2 + w_3x_3 + w_{1,1}x_1^2 + w_{2,2}x_2^2 + w_{3,3}x_3^2 + w_{1,2}x_1x_2 + w_{1,3}x_1x_3 + w_{2,3}x_2x_3.$$

If the input vector $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \{0,1\}^n$, then $x_i^r = x_i$ for all $r > 1$ and $i = 1, 2, \ldots, n$. For example, in this case $[3]^2 = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}\}$.

Example 9. *For $n, m = 2$ and $\mathbf{x} = (x_1, x_2) \in \{0,1\}^2$ we have $[n]^2 = \{\emptyset, \{1\}, \{2\}, \{1,1\}, \{2,2\}, \{1,2\}\} = \{\emptyset, \{1\}, \{2\}, \{1,2\}\}$. The weighted sum of inputs z, is $z = w_\emptyset + w_{\{1\}}x_1 + w_{\{2\}}x_2 + w_{\{1,2\}}x_1x_2$. If we take $S = [2]^2$, $w_\emptyset = -\frac{1}{2}$,$w_{\{1\}} = w_{\{2\}} = 1$ and $w_{\{1,2\}} = -2$, $z = x_1 + x_2 - 2x_1x_2 - \frac{1}{2}$, then the polynomial threshold perceptron computes XOR, see Table 1. If we take $S = [2]^2$, $w_\emptyset = \frac{1}{2}$,$w_{\{1\}} = w_{\{2\}} = -1$ and $w_{\{1,2\}} = 2$, then $S = [2]^2$, $z = -x_1 - x_2 + 2x_1x_2 + \frac{1}{2}$, then the polynomial threshold perceptron computes $XNOR$.*

Table 1: Polynomial threshold perceptron for XOR of two variables.

x_1	x_2	z	$P^2_{\mathbf{w}_S}(x_1, x_2)$	$x_1 \oplus x_2$
0	0	$-\frac{1}{2}$	0	0
0	1	$\frac{1}{2}$	1	1
1	0	$\frac{1}{2}$	1	1
1	1	$-\frac{1}{2}$	0	0

Theorem 6. [Universality Theorem [16, p. 53]] *Every function of n variables is computable by a polynomial threshold perceptron with degree of n.*

Corollary 4. *Every function $PARITY_n$ is computable by a polynomial threshold perceptron of degree n.*

A Quantum Unconstrained Binary Optimisation (QUBO) problem is an **NP**-hard mathematical problem consisting in the minimisation of a quadratic objective function

$$q(\mathbf{x}) = \mathbf{x}^T Q \mathbf{x},$$

where $\mathbf{x} \in \{0,1\}^n$ and $Q = (Q_{i,j})$ is an $n \times n$ matrix:

$$x^* = \min_{\mathbf{x} \in \{0,1\}^n} \sum_{n \geq i \geq j \geq 1} x_i Q_{i,j} x_j. \tag{5}$$

The matrix Q can be chosen to be upper-diagonal so can write

$$q(\mathbf{x}) = \sum_i Q_{i,i}x_i + \sum_{i<j} Q_{i,j}x_ix_j.$$

The diagonal terms $Q_{i,i}$ are the linear coefficients and the non-zero off-diagonal terms $Q_{i,j}, i < j$ are the quadratic coefficients. The quantum annealing computer D-Wave solves natively QUBO problems [17, 18].

To compute a polynomial threshold perceptron using quantum annealing computation we need to turn the polynomial in (4) into an equivalent quadratic one. To this aim we use the Reduction by Substitution method [19, p. 1237] implemented by the *make_ quadratic* function in [20]. Suppose that $x_1x_2x_3 \in \{0,1\}^3$. The product of x_1x_2 is replaced by a new variable x_4, $x_1x_2x_3 = x_3x_4$, where $x_4 = x_1x_2$; to enforce the last equality a penalty function is added to x_3x_4. Accordingly,

$$x_1x_2x_3 = \min_{x_4}\{x_3x_4 + MP(x_1, x_2; x_4)\},$$

where, M is the penalty and

$$P(x_1, x_2; x_4) = x_1x_2 - 2(x_1 + x_2)x_3 + 3x_4.$$

Similarly, a polynomial term involving more than three variables can be reduced to a sum of quadratic ones by sequentially reducing the degree of the terms by one.

Corollary 5. *Every function with any number of variables can be computed by a quantum annealing program (on D-Wave).*

Proof. By Theorem 6 and the Reduction by Substitution method, a QUBO objective function can be obtained which is computable on D-Wave. □

Corollary 6. *Every function $PARITY_n$ is computable by a quantum annealing program (on D-Wave).*

Example 10. *For $n = 4$ we have*

$$\begin{aligned} P^4_{\mathbf{w}_S}(x_1, x_2, x_3, x_4) = &-x_0 - x_1 - x_2 - x_3 \\ &+ 2x_0x_1 + 2x_0x_2 + 2x_0x_3 + 2x_1x_2 + 2x_1x_3 + 2x_2x_3 \\ &- 4x_0x_1x_2 - 4x_0x_1x_3 - 4x_0x_2x_3 - 4x_1x_2x_3 \\ &+ 8x_0x_1x_2x_3. \end{aligned}$$

To convert the five non-quadratic terms to quadratic ones in $P^4_{\mathbf{w}_S}$ we use the D-Wave make_quadratic *function [20]. To this aim we define two ancillary variables $x_4 = x_0x_1$ and $x_5 = x_2x_3$. Next, we reformulate $P^4_{\mathbf{w}_S}$ based on x_4 and x_5:*

$$\begin{aligned} p_2(\mathbf{x}) = &-x_0 - x_1 - x_2 - x_3 \\ &+ 2x_4 + 2x_0x_2 + 2x_0x_3 + 2x_1x_2 + 2x_1x_3 + 2x_5 \\ &- 4x_2x_4 - 4x_3x_4 - 4x_0x_5 - 4x_1x_5 \\ &+ 8x_4x_5 + M(P1 + P2), \end{aligned}$$

where, $P1$ and $P2$ are the penalty functions and M is the penalty weight [21]:

$$P1(x_0, x_1; x_4) = x_0x_1 - 2x_0x_4 - 2x_1x_4 + 3x_4, P2(x_2, x_3; x_5) = x_2x_3 - 2x_2x_5 - 2x_3x_5 + 3x_5.$$

Accordingly, for $M = 5$ we have

$$
\begin{aligned}
p_2(\mathbf{x}) = & -x_0 - x_1 - x_2 - x_3 + 2x_4 + 2x_5 \\
& + 2x_0x_2 + 2x_0x_3 + 2x_1x_2 + 2x_1x_3 \\
& - 4x_2x_4 - 4x_3x_4 - 4x_0x_5 - 4x_1x_5 \\
& + 8x_4x_5 \\
& + 5x_0x_1 - 10x_0x_4 - 10x_1x_4 + 15x_4 \\
& + 5x_2x_3 - 10x_2x_5 - 10x_3x_5 + 15x_5
\end{aligned}
$$

Last we simplify the above equation and get

$$
\begin{aligned}
p_2(\mathbf{x}) = & -x_0 - x_1 - x_2 - x_3 + 17x_4 + 17x_5 \\
& + 5x_0x_1 + 2x_0x_2 + 2x_0x_3 + 2x_1x_2 + 2x_1x_3 + 5x_2x_3 \\
& - 10x_0x_4 - 10x_1x_4 - 4x_2x_4 - 4x_3x_4 \\
& - 4x_0x_5 - 4x_1x_5 - 10x_2x_5 - 10x_3x_5 \\
& + 8x_4x_5
\end{aligned}
$$

The Appendix contains the computation details. The visualisation of Q on D-Wave Advantage using D-Wave Inspector is presented in Figure 4 and Figure 3: as expected, there is no broken chain. It is easy to check the correctness of the QUBO formulation $p_2(\mathbf{x})$, i.e. for all $\mathbf{x} \in \{0,1\}^6$, $PARITY_4(\mathbf{x}) = 1$ if and only if $\mathbf{x}$ is a solution of the QUBO problem $p_2(\mathbf{x})$.

6 Conclusions

The paper contributes to the investigation of a not yet fully explored problem of computational limitations of NNs. Solutions to that problem are especially important as NNs are broadly used in intelligent systems despite the lack of a theory for understanding and providing guarantees for their behaviour. Our study is limited to NNs implementing Boolean functions, but we believe that the results can be transposed to other domains too. Starting from the observation that NNs are good enough for the classification of massive data that exhibit some kind of robustness we provide a framework for formalising this property and provide related computability results. We initially introduce three characteristic and simple classes of functions and show how the complexity of NNs computing them is correlated with this property. Then we show how these results can be generalised to classes of functions depending on the robustness, or its opposite, sensitivity, of their representations.

Our study raises a whole host of problems about the computability of classes of functions and in particular the following open questions: Does there exist $k > 0$ such that every robust function is computable by an MLP with a single hidden layer and at most k hidden units? How can one combine classical and quantum computing with NN computing to obtain better results?

Acknowledgment

We thank V. Mitrana for comments which improved the paper and J. M. Gottlieb and T. Mittal for insight into D-Wave Advantage.

Appendix: QUBO for $PARITY_4$

We have used the function *make_quadratic* from Dimond Library of the Ocean SDK, there is a library named Dimod [20] to generate the QUBO for $P^4_{\mathbf{w}_S}$ in Example 10.

```
#Computing f_4 with D-Wave
import dimod
from dwave.system import DWaveSampler
from dwave.system import DWaveSampler, EmbeddingComposite
poly = {(0,): -1, (1,): -1, (2,): -1, (3,):-1, (0, 1): 2, (0,2): 2,
        (0,3):2, (1,2):2, (1,3):2, (2,3):2,(0, 1, 2): -4, (0, 1, 3): -4,
        (0, 2, 3): -4, (1, 2, 3): -4, (0,1,2,3): 8}

bqm = dimod.make_quadratic(poly, 5, dimod.BINARY)
a=list(bqm.to_qubo())
print(a)
```

make_quadratic.py

```
[{(0, 2): 2, (0, 3): 2, (0, 1): 5.0, (0, '0*1'): -10.0, (0, '2*3'): -4,
 (1, 2): 2, (1, 3): 2, (1, '0*1'): -10.0, (1, '2*3'): -4, (2, '0*1'): -4,
 (2, 3): 5.0, (2, '2*3'): -10.0, (3, '0*1'): -4, (3, '2*3'): -10.0,
 ('0*1', '2*3'): 8, (0, 0): -1.0, (1, 1): -1.0, (2, 2): -1.0,
 (3, 3): -1.0, ('0*1', '0*1'): 17.0, ('2*3', '2*3'): 17.0}, 0.0]
```

print(a).txt

The coloured terms in the output ($0 * 1$ and $2 * 3$) are the auxiliary variables x_4 and x_5. Accordingly, the QUBO Q was created and used on D-Wave Advantage for minimisation. Bellow the process and the results are shown. The minimum energies (-1) correspond exactly to the values of $\mathbf{x}$ such that $PARITY_4(\mathbf{x}) = 1$.

```
Q={('x0','x2'):2, ('x0','x3'):2, ('x0','x1'):5, ('x0','x4'):-10, ('x0', 'x5'): -4,
   ('x1','x2'): 2, ('x1', 'x3'):2,('x1','x4'):-10, ('x1','x5'):-4, ('x2', 'x4'): -4,
   ('x2','x3'):5, ('x2','x5'):-10, ('x3', 'x4'): -4, ('x3', 'x5'): -10,('x4','x5'):8,
   ('x0', 'x0'): -1, ('x1', 'x1'): -1, ('x2', 'x2'): -1, ('x3', 'x3'): -1, ('x4', 'x4'): 17,
   ('x5', 'x5'): 17}

sampler_auto = EmbeddingComposite(DWaveSampler(solver={'topology__type__eq':'pegasus'}))

sampleset = sampler_auto.sample_qubo(Q, num_reads=1000, answer_mode='histogram',
                                     chain_strength=10)

print(sampleset)
```

E4_minimisation.py

x0	x1	x2	x3	x4	x5	energy	num_oc.	chain_b.
0	0	0	1	0	0	−1.0	71	0.0
1	0	1	1	1	0	−1.0	113	0.0
2	1	0	1	1	0	−1.0	108	0.0
3	1	1	1	0	1	−1.0	31	0.0
4	1	0	0	0	0	−1.0	37	0.0
5	1	1	0	1	1	0	−1.0	46	0.0
6	0	1	0	0	0	0	−1.0	70	0.0
7	0	0	0	1	0	0	−1.0	65	0.0
8	1	1	0	0	1	0	0.0	27	0.0
9	0	1	1	0	0	0	0.0	39	0.0
10	1	1	1	1	1	1	0.0	45	0.0
11	1	0	1	0	0	0	0.0	24	0.0
12	0	0	1	1	0	1	0.0	61	0.0
13	0	0	0	0	0	0	0.0	28	0.0
14	1	0	0	1	0	0	0.0	31	0.0
15	0	1	0	1	0	0	0.0	42	0.0
16	1	1	1	1	0	1	3.0	9	0.0
17	0	1	1	0	1	0	3.0	2	0.0
18	1	0	0	1	0	1	3.0	14	0.0
19	0	1	0	1	0	1	3.0	9	0.0
20	1	0	1	0	1	0	3.0	6	0.0
21	1	1	0	0	0	0	3.0	4	0.0
22	0	1	0	1	1	0	3.0	5	0.0
23	1	0	0	1	1	0	3.0	7	0.0
24	0	1	1	0	0	1	3.0	36	0.0
25	1	0	1	0	0	1	3.0	16	0.0
26	1	1	1	1	1	0	3.0	7	0.0
27	0	0	1	1	0	0	3.0	13	0.0
41	1	0	1	0	0	1	3.0	1	0.166667
28	1	1	0	1	0	1	5.0	1	0.0
29	0	1	1	1	1	0	5.0	2	0.0
30	1	1	1	0	0	1	5.0	9	0.0
31	1	0	1	1	1	0	5.0	2	0.0
32	0	0	0	1	0	1	6.0	1	0.0
33	1	0	1	1	1	1	6.0	2	0.0
34	1	1	0	1	1	1	6.0	3	0.0
35	0	1	1	1	1	1	6.0	2	0.0
36	0	1	0	0	1	0	6.0	1	0.0
37	1	1	1	0	1	1	6.0	2	0.0
38	1	0	1	1	0	0	6.0	2	0.0

```
39  0  0  1  0  0  1     6.0        2        0.0
40  0  1  1  1  0  0     6.0        4        0.0
['BINARY', 42 rows, 1000 samples, 6 variables]
```

D–Wave output.txt

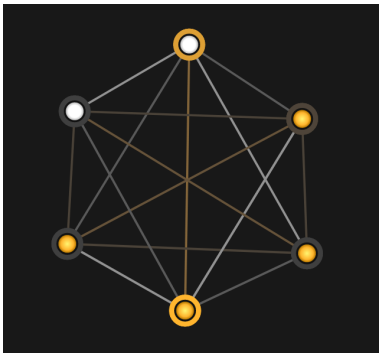

Figure 3: Q graph

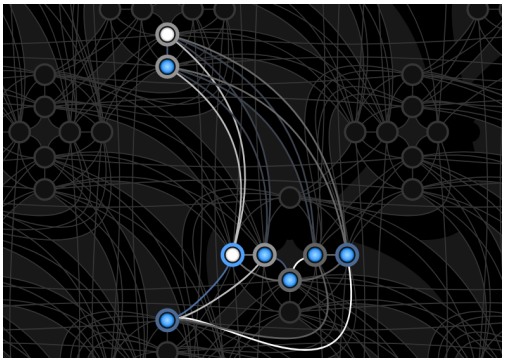

Figure 4: Graph Q in Pegasus graph

References

[1] R. Roscher, B. Bohn, M. F. Duarte, and J. Garcke, "Explainable machine learning for scientific insights and discoveries," *IEEE Access*, vol. 8, pp. 42200–42216, 2020.

[2] D. Harel, A. Marron, and J. Sifakis, "Autonomics: In search of a foundation for next-generation autonomous systems," *Proceedings of the National Academy of Sciences*, vol. 117, no. 30, pp. 17491–17498, 2020.

[3] J. Sifakis, "Can we trust autonomous systems? boundaries and risks," in *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings* (Y. Chen, C. Cheng, and J. Esparza, eds.), vol. 11781 of *Lecture Notes in Computer Science*, pp. 65–78, Springer, 2019.

[4] D. Kahneman, *Thinking, Fast and Slow*. New York: Farrar, Straus and Giroux, 2011.

[5] Y. Crama and P. L. Hammer, *Boolean Functions Theory, Algorithms, and Applications*. Cambridge, England, UK: Cambridge University Press, 2011.

[6] R. Rojas, *Neural Networks*. Berlin: Springer, 1996.

[7] M. Anthony, *Discrete Mathematics of Neural Networks: Selected Topics*. Philadelphia, PA: SIAM, 2001.

[8] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press, 1969.

[9] H. W. Lin, M. Tegmark, and D. Rolnick, "Why does deep and cheap learning work so well?," *Journal of Statistical Physics*, vol. 168, no. 6, pp. 1223–1247, 2017.

[10] N. L. W. Keijsers, "Neural Networks," in *Encyclopedia of Movement Disorders* (K. Kompoliti and L. V. Metman, eds.), pp. 257–261, Oxford: Academic Press, Jan. 2010.

[11] Z. Peng, "Multilayer Perceptron Algebra," Jan. 2017, http://arxiv.org/abs/1701.04968.

[12] B. Steinbach and R. Kohut, "Neural networks – a model of Boolean functions," in *Boolean Problems, Proceedings of the 5th International Workshop on Boolean Problems*, pp. 223–240, 2002.

[13] N. Pippenger, "The shortest disjunctive normal form of a random boolean function," *Random Structures & Algorithms*, vol. 22, no. 2, pp. 161–186, 2003.

[14] T. Stephen and T. Yusun, "Counting inequivalent monotone boolean functions," *Discrete Applied Mathematics*, vol. 167, pp. 15–24, 2014.

[15] M. Anthony, "Boolean functions and artificial neural networks." CDAM research report series (LSE-CDAM-2003-01), http://www.cdam.lse.ac.uk/Reports/Files/cdam-2003-01.pdf, 2003.

[16] C. Wang and A. Williams, "The threshold order of a Boolean function," *Discrete Applied Mathematics*, vol. 31, no. 1, pp. 51–69, 1991.

[17] C. McGeoch, *Adiabatic Quantum Computation and Quantum Annealing. Theory and Practice*. Morgan & Claypool Publishers, 2014.

[18] C. S. Calude, E. Calude, and M. J. Dinneen, "Adiabatic quantum computing challenges," *ACM SIGACT News*, vol. 46, pp. 40–61, March 2015.

[19] H. Ishikawa, "Transformation of general binary MRF minimization to the first-order case," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 6, pp. 1234–1249, 2010.

[20] "D-Wave Systems, Dimod. https://docs.ocean.dwavesys.com/en/stable/docs_dimod/reference/generated/dimod.higherorder.utils.make_quadratic.html," 2021.

[21] "D-Wave. Problem-Solving Handbook. https://docs.dwavesys.com/docs/latest/handbook_reformulating.html?highlight=higher%20degree#polynomial-reduction-by-substitution," 2021.