What Neural Networks Are (Not) Good For?

Cristian S. Calude\textsuperscript{1}, Shahrokh Heidari\textsuperscript{1}, Joseph Sifakis\textsuperscript{2}

\textsuperscript{1}School of Computer Science, University of Auckland, New Zealand
\textsuperscript{2}Verimag Laboratory, University Grenoble Alpes, France

CDMTCS-556
July 2021; revision December 2021

Centre for Discrete Mathematics and Theoretical Computer Science
What Perceptron Neural Networks Are (Not) Good For?

Cristian S. Calude¹, Shahrokh Heidari¹, Joseph Sifakis²

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

December 17, 2021

Abstract

Perceptron Neural Networks (PNNs) 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 PNNs can approximate a wide variety of functions, but comparatively very few discuss their limitations, the scope of this paper. To this aim we define two classes of Boolean functions – sensitive and robust –, and prove that an exponentially large set of sensitive functions are exponentially difficult to compute by multi-layer PNNs (hence incomputable by single-layer PNNs) and a comparatively large set of functions in the second one, but not all, are computable by single-layer PNNs.

Finally we used polynomial threshold PNNs to compute all Boolean functions with quantum annealing and present in detail a QUBO computation on the D-Wave Advantage.

These results confirm that the successes of PNNs, or lack of them, are in part determined by properties of the learned data sets and suggest that sensitive functions may not be (efficiently) computed by PNNs.

1 Introduction

Neural Networks (NNs), a computing paradigm that radically differs from conventional computing, 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 NNs 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 an 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 arises 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 some computational limits of Boolean functions with Perceptron Neural Nets (PNNs). Depending on the number of layers, PNNs can be single-layer or multi-layer. To compare the computational power of various PNNs we define two classes of Boolean functions – sensitive and robust –, and prove that an exponentially large set of functions in the first class are exponentially difficult to compute by multi-layer PNNs (hence incomputable by single-layer PNNs) and a comparatively large set of functions in the second one, but not all, are computable by single-layer PNNs.

The paper starts with a minimal amount of notation, defines two notions of sensitive and one of robust Boolean functions and introduces three infinite classes of Boolean functions, $\text{PARITY}_n, P^1_n, P^2_n$, used as benchmarks. We move on to limitations of single-layer and multi-layer PNNs in computing the strongly sensitive functions $\text{PARITY}_n$, 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 PNNs. Then we give more general results, including the fact that the set of sensitive functions which are computed by multi-layer PNNs with a single hidden layer and an exponential number of hidden units is exponentially larger than the set of functions computable by single-layer PNNs.

Finally, we use polynomial threshold PNNs to compute all Boolean functions with quantum annealing and present in detail a QUBO computation of $\text{PARITY}_4$ 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 $x, y$: depending on the context we will write $x = (x_1, x_2, \ldots, x_n)$ or $x = x_1 x_2 \ldots x_n$. The Boolean operations will be denoted by $\neg$ (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 $w$.

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 $T(f) = \{x \in \{0, 1\}^n \mid f(x) = 1\}$ and $F(f) = \{x \in \{0, 1\}^n \mid f(x) = 0\}$, 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 $R_1^n, R_2^n : \{0, 1\}^n \to \{0, 1\}$ defined by $R_1^n(x) = OR_n(x)$ and $R_2^n(x) = x_1$ and by $\text{PARITY}_n : \{0, 1\}^n \to \{0, 1\}$ the function

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

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

**Proof.** It is seen that $\text{PARITY}_2 = XOR$ and $\text{PARITY}_{n+1}(x_1, x_2, \ldots, x_{n+1}) = \overline{\overline{\text{XOR}}}(\text{PARITY}_n(x_1, x_2, \ldots, x_n), x_{n+1})$. \qed

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 $n$: $d(x, y)$ is the number of positions $i$ such that $x_i \neq y_i$.

**Example 1.** If $T(f) = \{111, 100, 001\}$, then $d(x, y) = 2$ for all distinct $x, y \in T(f)$. The Boolean function $\text{PARITY}_3$ has this property, but $R_1^3$ and $R_2^3$ do not have it.

3 Computing with single-layer PNNs

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 PNN [6, p. 7] computes a function $P_{\theta, w} : \mathbb{R}^n \to \{0, 1\}$, depending on two parameters, a threshold $\theta \in \mathbb{R}$ and a vector of $n$ weights $w = (w_1, w_2, \ldots, w_n) \in \mathbb{R}^n$, defined as follows:

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

3
The functions of the form (2) are also called threshold.

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

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

**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 $x_1, x_2, \ldots, x_k \in T(f)$ and every sequence $y_1, y_2, \ldots, y_k \in T(f)$ we have $\sum_{i=1}^k x_i \neq \sum_{i=1}^k y_i$.

**Example 3.** It is known that XOR and XNOR are not threshold functions [8]. More generally, the functions $\text{PARITY}_n$ cannot be computed by single-layer PNNs.

**Proof.** Consider the following four vectors in $\{0,1\}^n$: $x_1 = 0^{n-2}01$, $x_2 = 0^{n-2}10$, $y_1 = 0^{n-2}00$, $y_2 = 0^{n-2}11$. For every $n$ we have $\text{PARITY}_n(x_1) = \text{PARITY}_n(x_2)$, $\text{PARITY}_n(y_1) = \text{PARITY}_n(y_2)$, $\text{PARITY}_n(x_1) \neq \text{PARITY}_n(y_1)$ and $x_1 + x_2 = y_1 + y_2$, so the conclusion follows from Theorem 1. □

## 4 Sensitive vs. robust functions

In the previous section we have proved that $R^1_n$ and $R^2_n$ are computable by single-layer PNNs, but $\text{PARITY}_n$ is not. What is the reason for these results? What make some functions, but not all, computable by PNNs or even computable by some single-layer PNNs?

As it was pointed out in [9], this phenomenon is not surprising, we just need to ask the question differently. First, how can PNNs approximate functions well in practice, when the set of possible functions is exponentially larger than the set of practically possible PNNs? Indeed, there are $2^n$ different functions of $n$ variables, so a PNN 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 threshold PNNs.

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

This suggests that the two properties, robustness and sensitivity, could determine respectively, PNN’s computability or incomputability. To test this hypothesis we will propose definitions for these properties. 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 is robust. Quantifying the “small variation" will refine the definition.

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

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

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

It is clear that strong sensitiveness implies sensitiveness, but the converse implication is false. Indeed, the function $f(00) = 1$, $f(01) = 0$, $f(10) = 0$, $f(11) = 1$ is sensitive.
$d(00,10)=1$, $f(00)=1 \neq f(01)=f(10)$, 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 $x,y \in T(f)$, $d(x,y) \geq 2$.

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

**Corollary 1.** No (strongly) sensitive function is computed by a single-layer PNN.

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

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

$$P^i_{j,k}(a_i) = \begin{cases} 1, & \text{if } \sum_{k=1}^{n_i} w_{jk} a_k \geq \theta^i_j, \\ 0, & \text{otherwise}, \end{cases}$$

such that a) $a^0 = x \in \mathbb{R}^{n_1}$, $a^1 \in \mathbb{R}^{n_2}$, $\ldots$, $a^{L-1} \in \mathbb{R}^{n_L}$, b) $w^i \in \mathbb{R}^{(n_{i+1} \times n_i)}$ denotes the weight matrix connecting $i^{th}$ layer to $(i+1)^{th}$ layer; $w^i_j$ is the $j^{th}$ row of matrix $w^i$, c) $\theta^i \in \mathbb{R}^{(n_{i+1} \times 1)}$ is the threshold vector and $\theta^i_j$ is the $j^{th}$ row of this vector, d) $n_i$ is the number of units in the $i$th 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 multi-layer PNN is called binary classifier.

**Example 4.** A three-layer PNN with one hidden layer is presented in Figure 2.

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

**Theorem 3.** Every sensitive function of $n > 1$ variables is computed by a multi-layer PNN with a single hidden layer and $#(T(f))$ hidden units.

**Proof.** By Theorem 2, every function can be computed by a multi-layer PNN with a hidden layer by mapping each $x \in T(f)$ to a unit in the hidden layer [12, p. 3]; if $k = #(T(f))$, a multi-layer PNN with a single hidden layer can compute $f$ with $k$ hidden units.

By sensitivity and Proposition 1, for all $x, y \in T(f)$ we have $d(x,y) \geq 2$, so the 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]. $\square$
Corollary 2. Every function PARITY\textsubscript{n} with \( n > 1 \) is computed by a multi-layer PNN 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 a multi-layer PNN with a single hidden layer.

Theorem 4. For every \( n > 2 \) there exist \( 2^{2n-2} - 2 \) sensitive, not strongly sensitive functions which are computed by multi-layer PNNs 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 \( S \subset P \) 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 \( \#(T(f_S)) > 2^{n-2} \), by Theorem 3, every multi-layer PNN with a single hidden layer that computes it has an exponential number of hidden units. Furthermore, the number of all functions \( f_S \) is at least \( 2^{2n-2} - 2 \). \( \square \)
**Corollary 3.** The set of sensitive functions of \( n > 2 \) variables which are computed by multi-layer PNNs 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 multi-layer PNNs 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 result follows directly from the definition of strong sensitivity.

**Lemma 2.** If \( f \) is a strongly sensitive function of \( n > 1 \) variables, then the functions \( \tilde{f} \), \( f_{\pi} \) (where \( \pi \) is a permutation of the set \( \{1, 2, \ldots, n\} \)) defined by \( \tilde{f}(x) = f(x), f_{\pi}(x_1, x_2 \ldots x_n) = f(x_{\pi(1)} x_{\pi(2)} \ldots x_{\pi(n)}) \) 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 \( g_f(x, x_{n+1}) = \text{XNOR}(f(x), x_{n+1}) \) is strongly sensitive.

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

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

\[
g_f(x, x_{n+1}) = f(x) x_{n+1} \lor \tilde{f}(y) x_{n+1} = f(y) x_{n+1} \lor f(y) x_{n+1} = g_f(y, x_{n+1}),
\]

so \( g_f \) is strongly sensitive.

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

Finally, by Lemma 2, \( g_f \) is strongly sensitive if and only if \( h_f = g_f \) 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 \( x, y, z \in \{0,1\}^n \):

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

**Theorem 5.** The functions \( \overline{\text{PARITY}_n} \) and \( \overline{\text{PARITY}_n}, n > 1 \) are the only strongly sensitive functions.

**Proof.** Clearly, \( \overline{\text{PARITY}_n} \) is strongly sensitive; by Lemma 2, \( \overline{\text{PARITY}_n} \) is also strongly sensitive.

If \( f_n \) is a strongly sensitive function of \( n > 2 \) variables, then \( f_n(x_1, \ldots, x_{n-1}, x_n) = \text{XOR}(f_{n-1}(x_n), x_{n-1}) = f_n(x_1, \ldots, x_{n-1}, 0) \). 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

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

(3)

Out of all 16 functions of 2 variables only two, $\text{PARITY}_2, \overline{\text{PARITY}}_2$, 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 $\text{XOR}$ and $\overline{\text{XOR}}$. From Lemma 3 we deduce that in the set of $2^{n-1}$ functions obtained from all compositions of $\text{XOR}$ and $\overline{\text{XOR}}$ there are only two distinct functions, $\text{PARITY}_n$ and $\overline{\text{PARITY}}_n$.

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 $x, y \in \{0, 1\}^n$ such that $d(x, y) = 1$ we have $f(x) = f(y)$”; this property seems to be a weak form of robustness. The condition “for every $x, y \in \{0, 1\}^n$ such that $d(x, y) = 1$ we have $f(x) = f(y)$” 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 $x \in \{0, 1\}^n$ there exists $y \in \{0, 1\}^n$ such that $d(x, y) = 1$ we have $f(x) = f(y)$.

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

Proposition 3. The functions $R_1^n$ and $R_2^n$ are robust and threshold.

Proof. If $x \in \{0, 1\}^n$ with $R_1^n(x) = 1$ we can find an $y \in \{0, 1\}^n$ such that $d(x, y) = 1$ and $R_1^n(y) = 1$. If $x$ contains only one 1, then $y$ can be obtained by from $x$ by replacing a single bit 0 with 1; otherwise $y$ can be obtained from $x$ by replacing a single bit 1 with 0. If $R_1^n(x) = 0$, then $x = 0^n$, so we take $y = 10^{n-1}$; $d(x, y) = 1$ and $R_1^n(y) = 1$. If $x \in \{0, 1\}^n$ with $R_2^n(x) = x_1 = 1$, then every $y \in \{0, 1\}^n$ such that $y_1 = 1$ and $d(x, y) = 1$ satisfies $R_2^n(y) = 1$; the case $R_2^n(x) = 0$ is similar. By Example 2, $R_1^n$ and $R_2^n$ are threshold functions.

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

Example 6. Monotone non-constant functions are robust, but not all of them are computable by single-layer PNNs.

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 $D_n \geq 2^{\left(\binom{n}{2}\right)}$, [14], hence the result.

5 Quantum annealing computation of polynomial threshold single-layer PNNs

A polynomial threshold unit is a generalisation of a PNN 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 $[n]$. 

8
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 \( x_\emptyset = 1 \), \( x_{\{2,3\}} = x_{\{3,2\}} = x_2 x_3 \), \( x_{\{1,1\}} = x_1^2 \).

A polynomial threshold single-layer PNN is defined by a vector parameter \( \mathbf{w}_S \), with \( S \in [n]^m \). The function \( P_{\mathbf{w}_S}^n : \mathbb{R}^n \to \{0, 1\} \) computed by a polynomial threshold single-layer PNN with parameter \( \mathbf{w}_S \) is

\[
P_{\mathbf{w}_S}^n(x) = \begin{cases} 1, & \text{if } \sum_{T \in S} w_T x_T \geq 0, \\ 0, & \text{otherwise}. \end{cases}
\]

The degree of the polynomial is the degree of the PNN.

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 single-layer PNN has the form:

\[
w_\emptyset + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_{1,1} x_1^2 + w_{2,2} x_2^2 + w_{3,3} x_3^2 + w_{1,2} x_1 x_2 + w_{1,3} x_1 x_3 + w_{2,3} x_2 x_3.
\]

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 \([n]^2 = \emptyset, \{1\}, \{2\}, \{1, 1\}, \{2, 2\}, \{1, 2\}\). The weighted sum of inputs \( z = w_\emptyset + w_{\{1\}} x_1 + w_{\{2\}} x_2 + w_{\{1,2\}} x_1 x_2 \). If we take \( S = [2]^2 \), \( w_\emptyset = -\frac{1}{2}, w_{\{1\}} = w_{\{2\}} = 1 \) and \( w_{\{1,2\}} = -2, z = x_1 + x_2 - 2 x_1 x_2 - \frac{1}{2} \), then the polynomial threshold single-layer PNN computes \( \text{XOR} \), 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 \( S = [2]^2 \), \( z = -x_1 - x_2 + 2 x_1 x_2 + \frac{1}{2} \), then the polynomial threshold single-layer PNN computes \( \text{XNOR} \).

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( z )</th>
<th>( P_{\mathbf{w}_S}^2(x_1, x_2) )</th>
<th>( x_1 \oplus x_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>(-\frac{1}{2})</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>(\frac{1}{2})</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>(\frac{1}{2})</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(-\frac{1}{2})</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: Polynomial threshold single-layer PNN for \( \text{XOR} \) of two variables.

Theorem 6. [Universality Theorem [16, p. 53]] Every function of \( n \) variables is computable by a degree \( n \) polynomial threshold single-layer PNN.

Corollary 4. Every function \( \text{PARITY}_n \) is computable by a degree \( n \) polynomial threshold single-layer PNN.

A Quantum Unconstrained Binary Optimisation (QUBO) problem is an \( \text{NP} \)-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:

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

The matrix \( Q \) can be chosen to be upper-diagonal so can write
\[ q(x) = \sum_i Q_{i,i}x_i + \sum_{i<j} Q_{i,j}x_ix_j. \]

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 single-layer PNN 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]. As an example, 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 polynomials 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. \(\square\)

**Corollary 6.** Every function \(\text{PARITY}_n\) is computable by a quantum annealing program (on D-Wave).

**Example 10.** For \(n = 4\) we have

\[ P^4_{\text{ws}}(x_1, x_2, x_3, x_4) = -x_0 - x_1 - x_2 - x_3 + 2x_0x_1 + 2x_0x_2 + 2x_0x_3 + 2x_1x_2 + 2x_1x_3 + 2x_2x_3 - 4x_0x_1x_2 - 4x_0x_1x_3 - 4x_0x_2x_3 - 4x_1x_2x_3 + 8x_0x_1x_2x_3. \]

To convert the five non-quadratic terms to quadratic ones in \(P^4_{\text{ws}}\) 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 \(P^4_{\text{ws}}\) based on \(x_4\) and \(x_5\):

\[ p_2(x) = -x_0 - x_1 - x_2 - x_3 + 2x_4 + 2x_0x_2 + 2x_0x_3 + 2x_1x_2 + 2x_1x_3 + 2x_5 - 4x_2x_4 - 4x_3x_4 - 4x_0x_5 - 4x_1x_5 + 8x_4x_5 + M(P_1 + P_2), \]

where, \(P_1\) and \(P_2\) are the penalty functions and \(M\) is the penalty weight [21]:

\[ P_1(x_0, x_1; x_4) = x_0x_1 - 2x_0x_4 - 2x_1x_4 + 3x_4, P_2(x_2, x_3; x_5) = x_2x_3 - 2x_2x_5 - 2x_3x_5 + 3x_5. \]

Accordingly, for \(M = 5\) we have
\[ p_2(x) = -x_0 - x_1 - x_2 - x_3 + 2x_4 + 2x_5 \\
+ 2x_0x_2 + 2x_0x_3 + 2x_1x_2 + 2x_1x_3 \\
- 4x_2x_4 - 4x_3x_4 - 4x_0x_5 - 4x_1x_5 \\
+ 8x_4x_5 \\
+ 5x_0x_1 - 10x_0x_4 - 10x_1x_4 + 15x_4 \\
+ 5x_2x_3 - 10x_2x_5 - 10x_3x_5 + 15x_5 \]

Last we simplify the above equation and get

\[ p_2(x) = -x_0 - x_1 - x_2 - x_3 + 17x_4 + 17x_5 \\
+ 5x_0x_1 + 2x_0x_2 + 2x_0x_3 + 2x_1x_2 + 2x_1x_3 + 5x_2x_3 \\
- 10x_0x_4 - 10x_1x_4 - 4x_2x_4 - 4x_3x_4 \\
- 4x_0x_5 - 4x_1x_5 - 10x_2x_5 - 10x_3x_5 \\
+ 8x_4x_5 \]

The Appendix contains the computation details. The visualisation of \( Q \) on D-Wave Advantage using D-Wave Inspector is presented in Figure 3 and Figure 4: as expected, there is no broken chain. It is easy to check the correctness of the QUBO formulation \( p_2(x) \), i.e. for all \( x \in \{0, 1\}^5 \), \( \text{PARITY}_4(x) = 1 \) if and only if \( x \) is a solution of the QUBO problem \( p_2(x) \).

6 Conclusions

The paper contributes to the investigation of a not yet fully explored problem of computational limitations of PNNs, and, more generally, NNs. Solutions to this problem are especially important as PNNs are broadly used in intelligent systems despite the lack of a theory for understanding and providing guarantees for their behaviour.

The starting point was the observation that PNNs are good enough for the classification of massive data that exhibit some kind of “robustness”. To test this conjecture we defined two classes of Boolean functions – sensitive and robust –, and proved that an exponentially large set of functions in the first class are exponentially difficult to compute by multi-layer PNNs (hence incomputable by single-layer PNNs) and a comparatively large set of functions in the second one, but not all, are computable by single-layer PNNs. The difference in PNNs computability between sensitive and robust functions is sharp. Sensitive functions are difficult to compute or incomputable by PNNs, a property which could depend on data coding. The results presented in this paper suggest that successes of PNNs, or lack of them, are in part determined by properties of the learned data sets; in particular, data robustness seems essential for computing with PNNs.

Finally we used polynomial threshold single-layer PNNs to compute all Boolean functions with quantum annealing and present in detail a QUBO computation on the D-Wave Advantage.

Our study is limited to PNNs implementing Boolean functions, but we conjecture that the results obtained here can be transposed to other types of NNs. In favour of this conjecture we mention similar results for Recurrent Neural Networks (RNNs). Indeed, as every Turing machine can be simulated by a RNN [22], hence the abstract (Blum) complexity theory [23] applies to RNNs. As a consequence, there exists a topologically large class of arbitrarily sparse \( \{0, 1\} \)-valued computable functions such that any finite variant of the constructed function is arbitrarily complex [24, 23].

11
This study raises a host of problems about the NN computability of classes of functions, in particular the following questions: How can one combine model-based computing/quantum computing with PNN computing to obtain better results?

Acknowledgment

We thank V. Mitran and the anonymous referees 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_quadric from Dimond Library of the Ocean SDK, there is a library named Dimod [20] to generate the QUBO for $P_{wS}^4$ in Example 10.

breakwhite-space

```python
import dimod
dw = DWaveSampler()
import dimod
from dwave.system import 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, (0, 1, 2, 3, 4): 8}
bqm = dimod.make_quadratic(poly, 5, dimod.BINARY)
a = list(bqm.to_ising())
print(a)

# printed a

1 0 1 1 0 1 -1.0 113 0.0
1 0 1 1 1 0 1 -1.0 108 0.0
1 1 1 0 1 1 0 -1.0 31 0.0
1 0 0 0 0 0 0 -1.0 37 0.0
1 0 1 1 0 1 -1.0 46 0.0
0 1 0 0 0 0 -1.0 70 0.0
0 0 0 0 1 0 1 -1.0 65 0.0
1 0 0 0 1 0 0 0.0 27 0.0
0 0 1 1 0 0 0 0.0 39 0.0
1 0 1 1 1 1 0 0.0 45 0.0
1 1 1 0 0 0 0 0.0 24 0.0
1 0 0 1 1 0 1 0.0 61 0.0
1 0 0 0 0 0 0 0.0 28 0.0
1 0 0 1 0 0 0 0.0 31 0.0
1 0 1 0 0 0 0 0.0 42 0.0
1 1 1 1 0 1 0 3.0 9 0.0
1 0 1 1 0 1 0 3.0 7 0.0
1 1 1 0 0 0 0 3.0 14 0.0
1 0 0 1 0 1 0 3.0 9 0.0
0 1 0 1 0 0 0 3.0 6 0.0
0 0 1 1 1 0 0 3.0 4 0.0
0 0 0 0 0 0 3.0 5 0.0
0 0 0 0 0 0 3.0 7 0.0
0 1 1 0 0 0 1 3.0 36 0.0
0 0 1 0 0 0 1 3.0 16 0.0
1 0 1 1 0 0 0 3.0 7 0.0
0 0 1 1 0 0 0 3.0 13 0.0
```

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 $x$ such that $PARITY_4(x) = 1$.

breakwhite-space

```python
Q=((x0', x2'), 2, (x0', x3'), 2, (x0', x1'), 5, (x0', x4'), 20, (x0', x5'), 4, (x2', x3'), 2, (x2', x4'), 2, (x2', x5'), 10, (x3', x4'), 8, (x3', x5'), 10, (x4', x5'), 8, (x0', x3'), 1, (x1', x3'), 5, (x2', x2'), 1, (x3', x3'), 1, (x4', x4'), 17, (x5', x5'), 17)
sampler = EmbeddingComposite(DWaveSampler(solver=['topology_type': 'pegasus']))
sampleset = sampler.sample_qubo(Q, num_reads=1000, answer_mode='histogram', chain_strength=10)
print(sampleset)

# printed sampleset

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
<th>energy</th>
<th>num_oc</th>
<th>chain b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 0 0 0</td>
<td>-1.0</td>
<td>171</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 1 0 1 1</td>
<td>-1.0</td>
<td>113</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 0 1 1 1 0 1</td>
<td>-1.0</td>
<td>108</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 1 1 0 1 0 1</td>
<td>-1.0</td>
<td>31</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 1 0 0 0 0 0</td>
<td>-1.0</td>
<td>37</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 1 0 1 1 0 1</td>
<td>-1.0</td>
<td>46</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 0 1 0 0 0 0</td>
<td>-1.0</td>
<td>70</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 0 0 0 0 1 0</td>
<td>-1.0</td>
<td>65</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 0 0 0 0 0 0</td>
<td>-1.0</td>
<td>27</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 0 0 1 0 0 0</td>
<td>0.0</td>
<td>39</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 1 1 1 1 1 0</td>
<td>0.0</td>
<td>45</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1 0 1 0 0 0</td>
<td>0.0</td>
<td>24</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 0 0 1 1 0 0</td>
<td>0.0</td>
<td>61</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 0 0 0 0 0 0</td>
<td>0.0</td>
<td>28</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14 1 0 0 0 0 0</td>
<td>0.0</td>
<td>31</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 0 0 0 0 0 0</td>
<td>0.0</td>
<td>42</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 1 1 1 0 1 1</td>
<td>3.0</td>
<td>9</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17 0 1 1 0 1 0</td>
<td>3.0</td>
<td>2</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18 1 1 0 0 0 1</td>
<td>3.0</td>
<td>14</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19 0 1 0 1 0 1</td>
<td>3.0</td>
<td>9</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 1 0 1 0 1 0</td>
<td>3.0</td>
<td>6</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21 1 1 0 0 0 0</td>
<td>3.0</td>
<td>4</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22 0 1 0 1 1 0</td>
<td>3.0</td>
<td>5</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23 0 1 1 0 0 0</td>
<td>3.0</td>
<td>7</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24 0 1 1 0 0 1</td>
<td>3.0</td>
<td>36</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25 1 0 1 0 0 1</td>
<td>3.0</td>
<td>16</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26 1 1 1 0 0 0</td>
<td>3.0</td>
<td>7</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27 0 0 1 1 0 0</td>
<td>3.0</td>
<td>13</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

12
References


