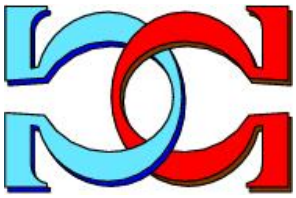

CDMTCS
Research
Report
Series

What Perceptron 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; revision August 2022

Centre for Discrete Mathematics and
Theoretical Computer Science

What Perceptron 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 2, 2022

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.

Many papers show that PNNs can approximate a wide variety of functions, but comparatively, very few discuss their limitations and 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). 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 various 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 [20]. The distinction between data-based and model-based knowledge is essential in systems engineering, where NNs are integrated into critical systems whose failures can harm 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 [7, 22]. 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 [12]: fast non-conscious thinking relies on a data-based empirical learning process. In contrast, conscious slow thinking is the result of explainable procedural reasoning. The human mind combines admirably 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 suitable 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 matters 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 more adequate for classifying robust massive information: minor input modifications will not drastically affect the classification result. This typically happens for NNs dealing with sensory information and implementing perception functions like 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) by analysing the source code of programs? The answer is probably no because software correctness is very sensitive to small changes in the source code. Moreover, the relationship between software syntax and its meaning defined by the operational semantics of the programming language can be profound and intricate. Similar issues can arise when we may try to use NNs as monitors for detecting failures of software systems. How much confidence can we have 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 the previous example again, 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 a 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 the convexity of data sets is essential for learning a given property, what is the complexity of 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). 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 the robust Boolean functions and introduces three infinite classes of Boolean functions, $PARITY_n, R^1_n, R^2_n$, used as benchmarks. We move on to limitations of single-

layer and multi-layer PNNs in computing the strongly sensitive functions $PARITY_n$, which are the most difficult to compute. In contrast, we prove that the robust functions like R_n^1, R_n^2 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 $PARITY_4$ 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

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

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 (DNF)* [6, p. 123]. The number of true points of f will be denoted by $\#T(f)$; $\#F(f) = 2^n - \#T(f)$. 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. *Consider the function f with $T(f) = \{111, 100, 001\}$, i.e. $f(111) = f(100) = f(001) = 1$. Then, $d(111, 100) = d(111, 001) = 2$; in fact, $d(\mathbf{x}, \mathbf{y}) = 2$ for all distinct $\mathbf{x}, \mathbf{y} \in T(f)$.*

Example 2. *For the Boolean functions $PARITY_3$, R_3^1, and R_3^2 we have $T(PARITY_3) = \{000, 011, 101, 110\}$, $T(R_3^1) = \{001, 010, 011, 100, 101, 110, 111\}$, $T(R_3^2) = \{100, 101, 110, 111\}$. A simple computation shows that $PARITY_3$, $d(\mathbf{x}, \mathbf{y}) = 2$ for all distinct $\mathbf{x}, \mathbf{y} \in T(PARITY_3)$; R_3^1 and R_3^2 do not have this property.*

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* [19, p. 7] computes 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 as follows:

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

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

Example 3. *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 $R_n^2 = P_{\frac{1}{2},(n,-1,\ldots,-1)}$. Indeed, with reference to (2) for R_n^1 we set $\mathbf{w} = (w_1, w_2, \ldots, w_n) = (1, 1, \ldots, 1)$ and $\theta = \frac{1}{2}$,

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

and for R_n^2 we set $\mathbf{w} = (w_1, w_2, \ldots, w_n) = (n, -1, -1, \ldots, -1)$ and $\theta = \frac{1}{2}$,

$$P_{\theta,\mathbf{w}}(\mathbf{x}) = \begin{cases} 1, & \text{if } nx_1 - \sum_{i=2}^n x_i \geq \frac{1}{2}, \\ 0, & \text{otherwise}, \end{cases} = R_n^2(\mathbf{x}).$$

Table 1 shows that R_3^1, R_3^2 are threshold functions.

x_1	x_2	x_3	$x_1+x_2+x_3$	$P_{\frac{1}{2},(1,1,1)}(\mathbf{x})$	R_3^1	$3x_1-x_2-x_3$	$P_{\frac{1}{2},(3,-1,-1)}(\mathbf{x})$	R_3^2
0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	-1	0	0
0	1	0	1	1	1	-1	0	0
0	1	1	2	1	1	-2	0	0
1	0	0	1	1	1	3	1	1
1	0	1	2	1	1	2	1	1
1	1	0	2	1	1	2	1	1
1	1	1	3	1	1	1	1	1

Table 1: Truth tables for the threshold functions R_3^1, R_3^2

□

Theorem 1. [1, 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 4. *It is known that XOR and XNOR are not threshold functions [16]. More generally, the functions $PARITY_n$ cannot be computed by single-layer PNNs.*

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 single-layer PNNs, but $PARITY_n$ is not. What is the reason for these results? What makes some functions, but not all, computable by PNNs or even computable by some single-layer PNNs?

As pointed out in [14], this phenomenon is not surprising; we 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^{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 $PARITY_n$, 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 $R_n^1(\mathbf{x}) = R_n^1(\mathbf{y})$, 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 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.

A sensitivity measure has been studied as the complexity of Boolean functions [11] and is the subject of the Sensitivity Theorem [8].

Definition 1. *[11, p. 57] Let f be a function of $n > 1$ variables. The sensitivity of f at $\mathbf{x} \in \{0,1\}^n$ is the number $s(f, \mathbf{x})$ of $\mathbf{y} \in \{0,1\}^n$ that differ from $\mathbf{x}$ in exactly one bit and satisfy $f(\mathbf{x}) \neq f(\mathbf{y})$. The sensitivity $s(f)$ of f is the maximum of $s(f, \mathbf{x})$ taken on all $\mathbf{x} \in \{0,1\}^n$. We say that f is* fully sensitive *if $s(f) = n$, i.e. $s(f)$ is maximum.*

Comment 1. Note that there exist fully sensitive Boolean functions f that are constant (robust) on as many inputs as we wish up to $2^n - n - 1$. Indeed, every function f such that $f(0^n) = 1$, $f(10^{n-1}) = f(010^{n-2}) = \cdots = f(0^{n-1}1) = 0$ is fully sensitive because $s(f, 0^n) = n$; however, we can assign $f(\mathbf{x}) = 0$ to as many of the unused $2^n - n - 1$ inputs $\mathbf{x}$, which shows that full sensitivity is "local". The minimal full sensitivity is achieved when $f(0^n) = 1$ and $f(\mathbf{x}) = 0$, for $\mathbf{x} \neq 0^n$. The same argument works for every $\mathbf{x} \in \{0,1\}^n$ and 0, 1 instead of 0^n and 1.

The properties of "sensitivity" necessary in this paper require some "uniformity" for all points concerning their "neighbourhoods," not just one "isolated" sensitive point, as in the case of full sensitivity. We define two (stronger) forms of sensitivity satisfying this requirement in the following.

Definition 2. *A function f of $n > 1$ variables is*

(a) sensitive *if it is not constant 0 and for all $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$ with $d(\mathbf{x}, \mathbf{y}) = 1$ and $f(\mathbf{x}) = 1$, we have $f(\mathbf{y}) = 0$.*

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

Proposition 1. *Strong sensitivity implies sensitivity which implies full sensitivity. The converse implications are false.*

Proof. It is clear that strong sensitivity implies sensitivity, 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 $f(01) = f(11) = 0, d(01, 11) = 1$. Sensitivity implies full sensitivity because by hypothesis there exists $\mathbf{x}$ such that $f(\mathbf{x}) = 1$, so for every $\mathbf{y} \neq \mathbf{x}$ with $d(\mathbf{x}, \mathbf{y}) = 1$ we have $f(\mathbf{y}) = 0$ showing that f is fully sensitive. Finally, R_2^1 is fully sensitive but not sensitive because $R_2^1(01) = R_2^1(11) = 1, d(01, 11) = 1$. □

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

Proposition 2. *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 computed by a single-layer* PNN.

Proof. Consider a sensitive function f of $n > 1$ variables. Let us take $\mathbf{x}', \mathbf{x}'' \in T(f)$, hence by Proposition 2, $d(\mathbf{x}', \mathbf{x}'') \geq 2$. 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. □

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 [13], see Figure 1. A multi-layer PNN $P : \mathbb{R}^{n_1} \to \mathbb{R}^{n_L}$ is defined by $L-1$ mappings acting on a sequence of spaces $(\mathbb{R}^{n_1}, \mathbb{R}^{n_2}, \ldots, \mathbb{R}^{n_L})$ [17], $P^1 : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}, P^2 : \mathbb{R}^{n_2} \to \mathbb{R}^{n_3}, \ldots, P^{L-1} : \mathbb{R}^{n_{L-1}} \to \mathbb{R}^{n_L}$, where each P^i $(1 \leq i < L)$ consists of j PNNs defined by:

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

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 multi-layer PNN is called a binary classifier.

Example 5. *A three-layer* PNN *with one hidden layer is presented in Figure 2.*

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

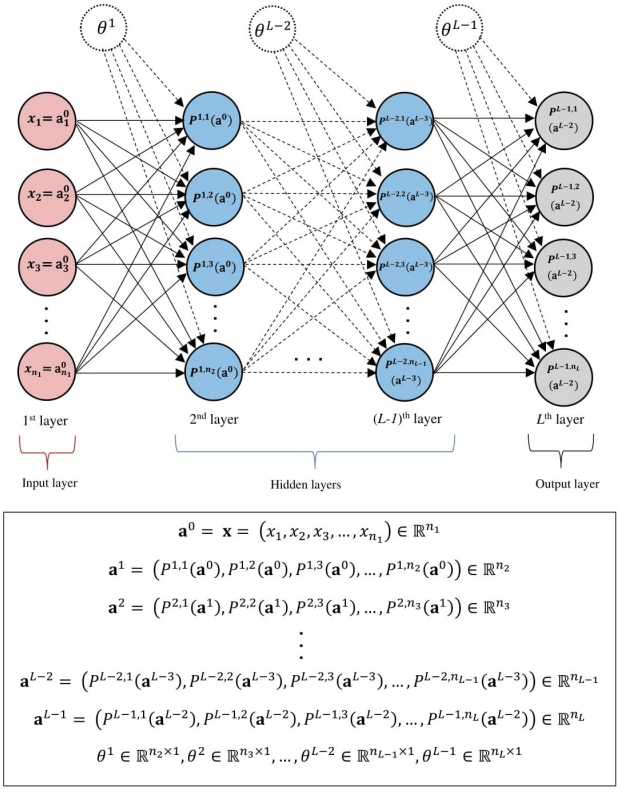

Figure 1: Multi-layer PNN with an input layer, $L-2$ hidden layers and an output layer

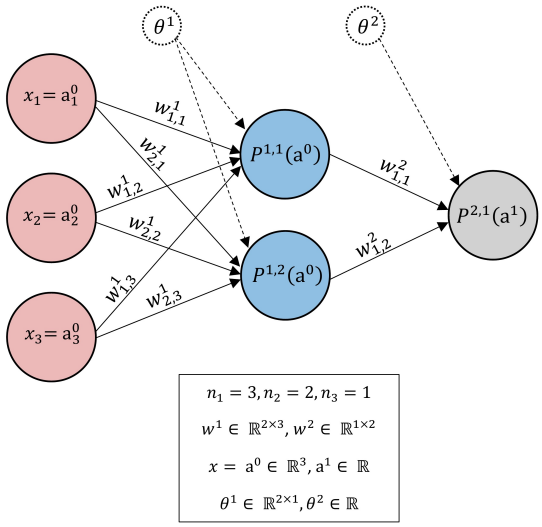

Figure 2: A simple example of a multi-layer PNN with a hidden layer

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. Recall that a DNF formula consists of a disjunction of conjunctions:

$$\bigvee_{1 \le i \le k} \left(x_1^i \wedge x_2^i \cdots \wedge x_n^i\right), \text{ and } x_j^i = x_j \text{ if } x_j^i = 1; \text{ otherwise } x_j^i = \bar{x}_j,$$

where $f(x_1^i, x_2^i, \dots, x_n^i) = 1$ for $1 \le i \le k$. Fix $k = \#(T(f))$. Based on the DNF

formula, f can be computed by a multi-layer PNN with a single hidden layer where each $(x_1^i \wedge x_2^i \cdots \wedge x_n^i)$ is mapped to a unit in the hidden layer for computation [24, p. 3]. Therefore, by Theorem 2, a multi-layer PNN with a single hidden layer can compute any function with k hidden units.

By sensitivity and Proposition 2, for all $\mathbf{x}, \mathbf{y} \in T(f)$ we have $d(\mathbf{x}, \mathbf{y}) \geq 2$, so the points in $T(f)$ cannot be combined to reduce the DNF formula [18, p. 162], hence the number of hidden units cannot be reduced. As each true point is a unit in the hidden layer, the hidden layer cannot have less than k units [24]. □

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

Comment 2. In [14] 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.

Example 6. *Figure 3 shows a multi-layer* PNN *with a single hidden layer and exactly $2^{2-1} = 2$ hidden units that computes $PARITY_2$.*

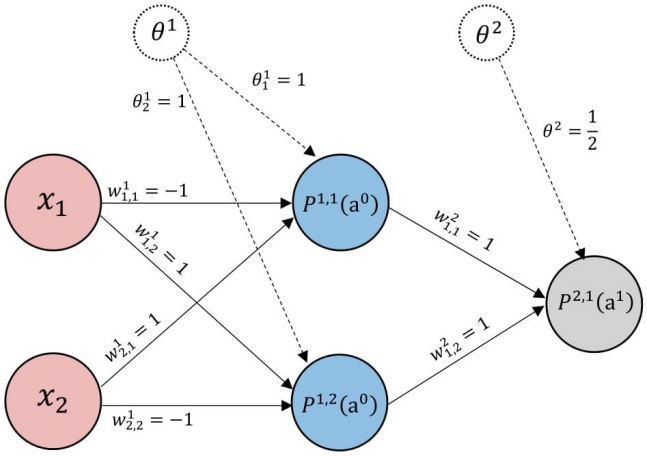

Figure 3: A multi-layer PNN with a single hidden layer computing $PARITY_2$

To justify the claim, we consider the weight matrices $\mathbf{w}^1$, $\mathbf{w}^2$ defined as

$$\mathbf{w}^1 = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}, \text{ and, } \mathbf{w}^2 = (1, 1),$$

and $\theta^1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and $\theta^2 = \frac{1}{2}$. Threfore, we have three perceptrons: $P^{1,1}, P^{1,2}$, and $P^{2,1}$ defined as:

$$P^{1,1}(\mathbf{a}^0) = \begin{cases} 1, & \text{if } -\mathbf{a}_1^0 + \mathbf{a}_2^0 \geq 1, \\ 0, & \text{otherwise}, \end{cases}$$

$$P^{1,2}(\mathbf{a}^0) = \begin{cases} 1, & \text{if } \mathbf{a}_1^0 - \mathbf{a}_2^0 \geq 1, \\ 0, & \text{otherwise}, \end{cases}$$

$$P^{2,1}(\mathbf{a}^1) = \begin{cases} 1, & \text{if } \mathbf{a}_1^1 + \mathbf{a}_2^1 \geq \frac{1}{2}, \\ 0, & \text{otherwise}, \end{cases}$$

where $a^0 = (x_1, x_2)$ and $a^1 = \big(P^{1,1}(\mathbf{a}^0), P^{1,2}(\mathbf{a}^0)\big)$. Table 2 shows how $PARITY_2$ can be computed with the multi-layer PNN *in* Figure 3*:*

$x_1 = \mathbf{a}_1^0$	$x_2 = \mathbf{a}_2^0$	$-\mathbf{a}_1^0 + \mathbf{a}_2^0$	$P^{1,1}(\mathbf{a}^0) = \mathbf{a}_1^1$	$\mathbf{a}_1^0 - \mathbf{a}_2^0$	$P^{1,2}(\mathbf{a}^0) = \mathbf{a}_2^1$	$\mathbf{a}_1^1 + \mathbf{a}_2^1$	$P^{2,1}(\mathbf{a}^1)$
0	0	0	**0**	0	**0**	0	**0**
0	1	1	**1**	-1	**0**	1	**1**
1	0	-1	**0**	1	**1**	1	**1**
1	1	0	**0**	0	**0**	0	**0**

Table 2: Truth table for the multi-layer PNN in Figure 3

Theorem 4. *For every $n > 2$ there exist $2^{2^{n-2}} - 2$ sensitive, not strongly sensitive functions which are computed by multi-layer* PNN*s 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 2, 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 $2^{2^{n-2}} - 2$. □

Corollary 3. *The set of sensitive functions of $n > 2$ variables which are computed by multi-layer* PNN*s 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, [1, 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 $\bar{f}$, 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 3. *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 3, 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 3 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 3. 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 $d(\mathbf{x}, \mathbf{y}) = 1$ 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 $d(\mathbf{x}, \mathbf{y}) = 1$ 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 3. *The function f is* robust *if for every $\mathbf{x} \in \{0,1\}^n$ there exists $\mathbf{y} \in \{0,1\}^n$ such that $d(\mathbf{x}, \mathbf{y}) = 1$ and $f(\mathbf{x}) = f(\mathbf{y})$.*

Example 7. *Every function f with $T(f) = \{\mathbf{x}, \mathbf{y}\}$ and $d(\mathbf{x}, \mathbf{y}) = 1$ is robust and threshold.*

Proposition 4. *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 3, 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 $1 \leq i \leq n$, $x_i \leq y_i$) we have $f(\mathbf{x}) \leq f(\mathbf{y})$.

Example 8. *Monotone non-constant functions are robust, but not all of them are computable by single-layer* PNN*s.*

Proof. The set of threshold functions of $n > 1$ variables has less than 2^{n^2} functions, [1, Theorem 4.3], which is a smaller subset of the set of monotone functions whose cardinality is the Dedekind number $D_n \geq 2^{\binom{n}{\lfloor n/2 \rfloor}}$, [25], 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 [2, 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 9. *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 $P^m_{\mathbf{w}_S} \colon \mathbf{R}^n \to \{0, 1\}$ computed by a polynomial threshold single-layer PNN with parameter $\mathbf{w}_S$ is

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

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

Example 10. *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 11. *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 $z = w_\emptyset + w_{\{1\}} x_1 + w_{\{2\}} x_2 + w_{\{1,2\}} x_1 x_2$. If we take $S = [2]^2$, $w_\emptyset = -\frac{1}{2}$, $w_{\{1\}} = w_{\{2\}} = 1$ and $w_{\{1,2\}} = -2$, $z = x_1 + x_2 - 2x_1 x_2 - \frac{1}{2}$, then the polynomial threshold single-layer* PNN *computes XOR, see* Table 3. *If we take $S = [2]^2$, $w_\emptyset = \frac{1}{2}$, $w_{\{1\}} = w_{\{2\}} = -1$ and $w_{\{1,2\}} = 2$, then $S = [2]^2$, $z = -x_1 - x_2 + 2x_1 x_2 + \frac{1}{2}$, then the polynomial threshold single-layer* PNN *computes XNOR.*

Theorem 6. [Universality Theorem [27, p. 53]] *Every function of n variables is computable by a degree n polynomial threshold single-layer* PNN.

Corollary 4. *Every function $PARITY_n$ is computable by a degree n polynomial threshold single-layer* PNN.

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

Table 3: Polynomial threshold single-layer PNN for XOR of two variables

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 I can write

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

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 [15, 5].

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 [10, p. 1237] implemented by the *make_ quadratic* function in [30]. As an example, suppose that $x_1 x_2 x_3 \in \{0,1\}^3$. The product of $x_1 x_2$ is replaced by a new variable x_4, $x_1 x_2 x_3 = x_3 x_4$, where $x_4 = x_1 x_2$; to enforce the last equality a penalty function is added to $x_3 x_4$. Accordingly,

$$x_1 x_2 x_3 = \min_{x_4} \{x_3 x_4 + M P(x_1, x_2; x_4)\},$$

where M is the penalty and

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

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

Corollary 5. *A quantum annealing program (on D-Wave) can compute every function with any number of variables.*

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. *A quantum annealing program computes every function $PARITY_n$ (on D-Wave).*

Example 12. *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* [30]. *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 [29]*:*

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

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 5: 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

There are lots of papers showing that PNNs can approximate a wide variety of functions, but comparatively very few discuss their limitations, see for example [14, 3].

This 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 fundamental as PNNs are broadly used in intelligent systems despite the lack of a theory for understanding and providing guarantees for their behaviour.

The issue of sensitivity vs. robustness has been extensively studied for various classes of NNs, e.g. [28, 21, 26]. These works use different metrics to estimate the impact of variations in either architecture, including connectivity and weights, or dataset characteristics, on overall neural network performance. Their goal is to distinguish design decisions that are important from inconsequential in the intended class of NNs.

This paper adopts a different angle of attack by studying theoretical aspects of a poorly studied problem for NNs: their expressiveness of sensitivity and robustness, i.e. the study of classes of Boolean functions that PNNs can compute and their complexities measured in terms of neurons and parameters needed to compute a given function.

Our results shed light on the well-known problem "How can neural networks approximate functions well in practice when the set of possible functions is exponentially larger than the set of possible networks in practice?" [14]. They suggest that functions with almost no isolated points are easier to compute, which is consistent with the successful application of NNs to massive and robust information classification.

Our starting point was the observation that PNNs are good enough for classifying massive data that exhibit some "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). 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. Our results suggest that the successes of PNNs, or lack of them, are in part determined by the properties of the learned data sets; in particular, data robustness seems essential for computing with PNNs.

Considering PNNs for computing Boolean functions facilitates an elegant mathematical treatment that can be much more twisted for other classes of neural nets computing more general functions. Nevertheless, we conjecture that the distinctions regarding the sensitivity/robustness of the functions to be computed are transferable to other classes of NNs, of which PNNs are the primary building blocks. We mention similar results for Recurrent Neural Networks (RNNs) in favour of this conjecture. Indeed, as every Turing machine can be simulated by an RNN [9], the abstract (Blum) complexity theory [4] 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 [23, 4].

This study raises a host of problems about the NNs computability of classes of functions, in particular the following two. a) Study degrees of sensitivity/robustness of Boolean functions by parameterising these properties. For instance, one can define δ-sensitivity of a Boolean function f of n variables where δ is the ratio $\frac{k}{2n}$, k is the number of "isolated points" x of f, i.e., $f(x) \neq f(x')$ for all x' such that $d(x,x') = 1$. Note that 1-sensitivity is strong sensitivity while 0-sensitivity is robustness because robust functions have no isolated points. The property of sensitivity we defined corresponds to the cases where δ is different from 0 and 1. It is essential to study how the complexity of PNNs changes when the parameter δ varies in the interval $[0,1]$. b) Combine classical and quantum computing with PNN computing to improve the presented results using polynomial threshold single-layer PNNs to compute all Boolean functions with quantum annealing (for example, by reducing the degrees of the polynomials).

Finally, do the theoretical results discussed in the paper have any practical value?

Here is one suggestion: study the δ-sensitivity of (not necessarily Boolean) functions before trying to compute them with NNs.

Acknowledgment

We thank V. Mitrana and the anonymous referees for the comments which improved the paper, J. M. Gottlieb and T. Mittal for insight into D-Wave Advantage and A. Adamatzky for continuous support.

References

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

[2] 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.

[3] S. Ben-David, P. Hrubeš, S. Moran, A. Shpilka, and A. Yehudayoff. Learnability can be undecidable. *Nature Machine Intelligence*, 1(1):44–48, 2019.

[4] C. Calude. *Theories of Computational Complexity*. North-Holland, Amsterdam, 1988.

[5] C. S. Calude, E. Calude, and M. J. Dinneen. Adiabatic quantum computing challenges. *ACM SIGACT News*, 46(1):40–61, March 2015.

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

[7] 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*, 117(30):17491–17498, 2020.

[8] H. Huang. Induced subgraphs of hypercubes and a proof of the sensitivity conjecture. *Annals of Mathematics*, 190:949–955, 2019.

[9] H. Hyötyniemi. Turing machines are recurrent neural networks. In T. H. Jarmo Alander and M. Jakobsson, editors, *Proceedings of STeP'96*, pages 13–24. Publications of the Finnish Artificial Intelligence Society, 1996.

[10] H. Ishikawa. Transformation of general binary MRF minimization to the first-order case. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(6):1234–1249, 2010.

[11] S. Jukna. *Boolean Function Complexity - Advances and Frontiers*, volume 27 of *Algorithms and combinatorics*. Springer, 2012.

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

[13] N. L. W. Keijsers. Neural Networks. In K. Kompoliti and L. V. Metman, editors, *Encyclopedia of Movement Disorders*, pages 257–261. Academic Press, Oxford, Jan. 2010.

[14] H. W. Lin, M. Tegmark, and D. Rolnick. Why does deep and cheap learning work so well? *Journal of Statistical Physics*, 168(6):1223–1247, 2017.

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

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

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

[18] N. Pippenger. The shortest disjunctive normal form of a random boolean function. *Random Structures & Algorithms*, 22(2):161–186, 2003.

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

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

[21] H. Shu and H. Zhu. Sensitivity analysis of deep neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):4943–4950, Jul. 2019.

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

[23] C. H. Smith. A note on arbitrarily complex recursive functions. *Notre Dame J. Formal Log.*, 29(2):198–207, 1988.

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

[25] T. Stephen and T. Yusun. Counting inequivalent monotone boolean functions. *Discrete Applied Mathematics*, 167:15–24, 2014.

[26] A. van Duynhoven and S. Dragićević. Exploring the sensitivity of recurrent neural network models for forecasting land cover change. *Land*, 10(3), 2021.

[27] C. Wang and A. Williams. The threshold order of a Boolean function. *Discrete Applied Mathematics*, 31(1):51–69, 1991.

[28] Y. Zhang and B. Wallace. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. In Seong-Bae Park and Thepchai Supnithi, editor, *Proceedings of the IJCNLP 2017, Tapei, Taiwan, November 27 - December 1, 2017, System Demonstrations,*, pages 253–263. Association for Computational Linguistics, 2017.

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

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

Appendix: QUBO for $PARITY_4$

We have used the function *make_ quadratic* from Dimod Library of the Ocean SDK [30] to generate the QUBO for $P^4_{\mathbf{w}_S}$ in Example 12.

```
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)

# printed a
[{(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]
```

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)

# printed sampleset
    x0 x1 x2 x3 x4 x5  energy  num_oc.  chain_b.
0    0  0  1  0  0  0    -1.0       71       0.0
1    0  1  1  1  0  1    -1.0      113       0.0
2    1  0  1  1  0  1    -1.0      108       0.0
3    1  1  1  0  1  0    -1.0       31       0.0
4    1  0  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       0.166
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]
```

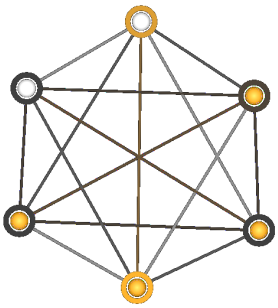

Figure 4: Q graph

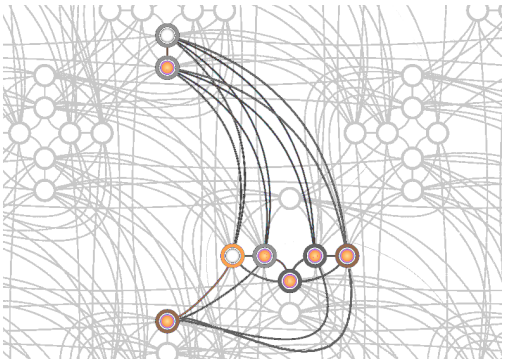

Figure 5: Graph Q in Pegasus graph