The Neural Network Loading Problem is Undecidable

Herbert Wiklicky

Centrum voor Wiskunde en Informatica
P.O.Box 94079, 1090 GB Amsterdam, The Netherlands*

Abstract

There exist several well known results concerning the computational complexity of training, design, etc. of (artificial) neural networks which demonstrated that these problems are at least NP complete. Extending these results we will prove here that the training of neural networks, i.e. the so called "loading problem", in general, is even "unsolvable".

This implies in particular that there exists no general or universal training algorithm which for any given neural network architecture could determine a correct set of "weights" such that a certain desired input/output behavior of the neural network is achieved.

1 Introduction

Artificial neural networks, like the Multi Layer Perceptron (MLP), can be used for approximating any function belonging to certain "reasonable" classes (e.g. continuous or measurable functions) as closely as desired. Related results can be found for example in [8] etc. These results are often celebrated as the theoretical justifications of what lead to a renaissance of neural network models in the second half of the 80s, because these new models could overcome the limitation of simple perceptrons as they were demonstrated by Minsky and Papert in the late 60s [14].

But clearly, such *existence theorems*, demonstrating that something *can* be achieved, immediately raised the question *how* this actually can be realized and how *expensive* this would be. But unfortunately, several well known investigations concerning the computational complexity of neural network training, design, etc. also made obvious that these problems actually can become very "hard" problems. Various authors have shown that for example the so called "loading problem" and several other related design problems for neural networks are in general *NP complete*, e.g. see [10, 2, 11, 1].

We will show here that this is not even the "worst" thing to happen one might encounter, but that the loading problem for recurrent networks or ones using *higher order* units in fact is even *unsolvable* in the sense of classical recursion theory. That implies in particular that no general training algorithm can exist for slightly more complex classes of neural networks than simple feed forward architectures, which were mainly considered in the above mentioned papers. The general "loading problem" of determining an optimal network configuration, i.e.

*This research work was partly supported by a grant from the Austrian Fonds zur Förderung der wissenschaftlichen Forschung (Projekt J0828-PHY); it was started during the author's affiliation with the Austrian Research Institute for Artificial Intelligence, Schottengasse 3, 1010 Wien, Austria.

set of weights, which solves a given learning task, is among those problems which "*indeed are intractable in an especially strong sense*" ([5] p 12).

2 The Loading Problem

It seems that there are relatively few commonly accepted general and formal definitions of the notion of a "neural network". Although our results also hold if based on other formal definitions we will try to stay here very close to the original setting given for Judd's NP completeness result [10], because our results are obtained in a very similar way. Therefore to avoid ambiguities we will base our arguments on the following definitions cited from [10].

Definition 1 *Define a [neural] architecture as a 5-tuple $A = (P, V, S, R, E)$ where*

- *P is a set of posts,*
- *V is a set of n nodes: $V = \{v_1, v_2, \ldots, v_n\} \subseteq P$,*
- *S is a set of s input posts: $S = P - V$,*
- *R is a set of r output posts: $R \subseteq P$, and*
- *E is a set of directed edges: $E \subseteq \{(v_i, v_j) : v_i \in P, v_j \in V, i < j\}$.*

Each node in a [neural] network contributes to the overall retrieval computation [evaluation] by computing an output signal as a function of the signals on its input edges.

The constraints on the edges ensures that no cycles occur in the graph. By dropping this requirement one can easily generalize this notion of a feed forward architecture to that of a recurrent network.

The *evaluation* of a network is achieved in an obvious (synchronous) way. Thus we associate to an p-dimensional input vector a r-dimensional output vector. For recurrent networks this type of *behaviour* might be generalized to sequences of vectors without running into any conceptual problems. We will refer to a specific behaviour we would like to be realized by a certain neural network as a *learning task T*.

Definition 2 *A configuration, $C = \{f_1, f_2, \ldots, f_n\}$, of a [neural] network is a list of n functions corresponding one to one with the set of nodes, V, meaning that f_i is the function that node i computes.*

With this the "loading problem" can be characterized as the problem to find a correct configuration C of N, i.e. an appropriate set of "node functions" (belonging to some given class of possible functions), so that the evaluation of N realizes the behavior required. That is, for all inputs for which an association was specified in the "learning task" T the evaluation of N with configuration C has to compute the desired output, while for unspecified input patterns nothing is imposed.

As usual we will analyse this constructive problem via its corresponding decision problem, because the decision problem gives a "lower bound" on the hardness of the related constructive problem [5]. If we could *construct* a correct configuration C for all instances of N and T, it would be trivial to *decide* instantly if a correct configuration exists at all. Thus we will consider the following

Decision Problem 1 ***Loading Problem***
INSTANCE: A neural network architecture N and a learning task T.
QUESTION: Is there a configuration C for N such that T is realized?

A principle technique for analysing the complexity of a certain class of problems is to find another class of problems, of which friendly people already have shown that it indeed is NP complete, unsolvable, etc., and to construct an admissible transformation (with polynomial effort, etc.) which proves that both problems are "equivalent".

Stephen Judd and others could prove this way that the "loading problem" for simple feed forward neural networks is NP complete. In many of those proofs the prototypical NP complete problem is the so called "3SAT Problem". For proving that the loading problem for other reasonable classes of node functions and architectures is unsolvable we need to introduce a different "reference problem" which is known to be *unsolvable.*

3 Hilbert's Tenth Problem

In 1900, at the International Congress of Mathematicians in Paris, David Hilbert proposed 23 mathematical problems, which he expected to play a vital role in the development of mathematics in the then beginning 20^{th} century. And indeed, some of these problems are still unsolved, while others were solved only decades later. Among these problems, as number ten, was the question how to solve diophantine equation [7].

Definition 3 *A* diophantine equation *is a polynomial D in n variables with integer coefficients, that is*

$$D(x_1, x_2, \ldots, x_n) = \sum_i d_i(x_1, x_2, \ldots, x_n)$$

with each term d_i of the form $d_i(x_1, x_2, \ldots, x_n) = c_i \cdot x_{j_1} \cdot x_{j_2} \cdot \cdots \cdot x_{j_{m_i}}$ where the indices $\{j_1, j_2, \ldots, j_{m_i}\}$ are taken from $\{1, 2, \ldots, n\}$ and the coefficient $c_i \in \mathbb{Z}$.

The problem is to find a (non trivial) integer solution, i.e. a vector $\vec{x} = (x_1, x_2, \ldots, x_n)$ with $x_i \in \mathbb{Z}$, such that $D(x_1, x_2, \ldots, x_n) = 0$, or formulated as a decision problem:

Decision Problem 2 ***Hilbert's Tenth Problem***
INSTANCE: Given any diophantine equation.
QUESTION: Is there a solution for this diophantine equation?

Examples of diophantine equations are: $x_1^2 - 25 = 0$, $x_1^2 - 20 = 0$, $x_1^2 + x_2^2 - x_3^2 = 0$, or $x_1^3 + x_2^3 - x_3^3 = 0$. While the first of these equations simply can be solved by $x_1 = 5$ the second one cannot be solved by any integer x_1. Equation number three has so called "Pythagorean Triples" as solutions, e.g. $(x_1 = 3, x_2 = 4, x_3 = 5)$. And finally, the claim that the last equations has no solution is known as Fermat's Last Theorem[1].

Although Hilberts Tenth's Problem might seem to be rather simple (its formulation is actually the shortest of his 23 problems) it lasted 70 years until a negative answer to Hilbert's question was found by Y. Matijasevich in 1970 [13]: There is no general algorithm which would solve any diophantine equation. Or, to put it different, there is no computable predicate for diophantine equations which holds for those diophantine equations for which a solution exists and fails for the others. Hilbert's Tenth Problem is "undecidable" or "unsolvable" ([4] Theorem 7.4).

[1]When the first draft of this paper was prepeared this problem was completely unsolved. By today it is commonly believed that Andrew Wiles has a proof [15].

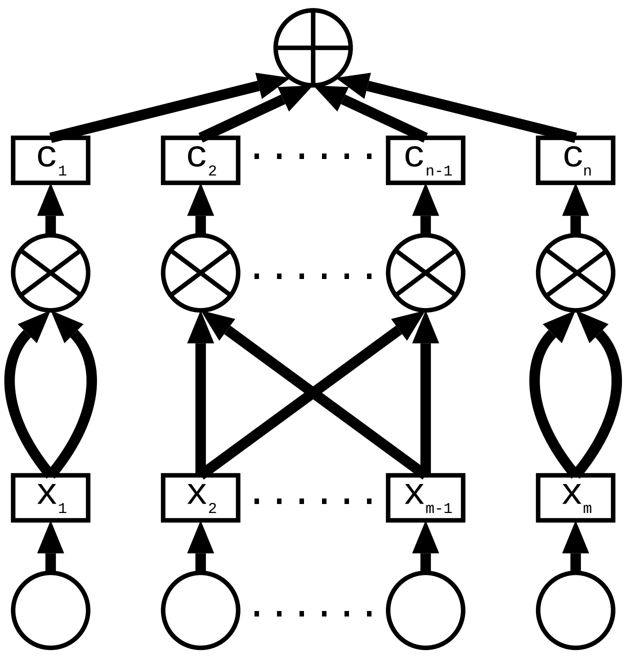

Figure 1: A network for $c_1x_1^2 + c_2x_2x_{m-1} + \ldots + c_{n-1}x_2x_{m-1} + x_m^2$

4 Networks With Higher Order Units

To prove now that for neural networks in general – at least for those using higher order units ($\Sigma\Pi$-units) – no universal learning algorithm exits, that is that the loading problem is undecidable, we will show the equivalence to Hilbert's Tenth Problem. The construction is quite straightforward.

As usual, we will identify certain coefficients of the node function – as introduced above – with certain "weights" associated to the (incomming) connections. The actual incoming signal to a unit is the product of the original signal multiplied by this weight parameter. We then have the following elements to build up our networks.

- First, we have a simple "summarizing" Σ-unit, i.e. the output of these units is computed as the sum over all incoming signals. We represent it by this icon: $\oplus$
- Second, as a higher order element, we need a kind of "production" Π-unit where the output is the product of all incoming signals. The corrsponding graphical representation is this: $\otimes$
- Third, we will use a little bit more complex connection type, which can supply the same signal to several receiving units. We will (first) allow only integer values as weights for all involved connections. Connections are symbolized by a simple square icon labeled with the corresponding weight value.

Note, that we also could have used only one combined unit type, which usually is referred to as a $\Sigma\Pi$-unit. That is we could just consider polynomials (with integer coefficients) as node functions, cf. [12]. Furthermore, it is easy to see that connections of the introduced type might also be replaced by a combination of simple links together with a primitive dummy unit.

The remaining task is now to construct with these types of units for each diophantine equation a corresponding neural network. The construction is shown in an example in Fig.1.

For each variable x_i of the diophantine equation we have one primitive input unit (or just an input signal). For each term d_i in the diophantine equation we take a Π-unit. We then connect those input units (signals) to a "term" unit for which the corresponding variable occurs in the term d_i. Perhaps multiple connections have to be used for this. The corresponding (integer) weights x_i will be the only variable parameters in our construction.

To end the construction of a neural network for each diophantine equation, all term units are finally connected via some further integer links to one Σ-unit. We fix the weights from those term units to the final Σ-unit with the corresponding coefficients c_i for d_i.

The learning task for this architecture is now simply represented by the following mapping:

$$(1, 1, \ldots, 1) \mapsto (0)$$

Theorem 1 *The loading problem for higher order neural networks with integer weights is unsolvable.*

First, it is trivial to see that if we had a solution of the original diophantine equation and take the solution vector as weights for the incoming links, this would also solve the loading problem for this architecture. Because this architecture just computes the polynomial which represents the diophantine equation if the input is a pattern of all 1s.

On the other hand, if there was an algorithm which would determine the correct (integer) weights for any such network, then the weights would also solve the original diophantine equation. Furthermore, all involved transformations between the solutions of both problems are trivial ones, and can be computed effectively (even in linear time). Therefore, we can conclude

5 Learning the Polynomial Coefficients

There might be several arguments which claim that this type of neural network is just "artificial" (which it is indeed!) and that this result therefore is somehow pathological and restricted, althought the network elements we used are consistent with the definitions given in several papers [10, 12], etc. Let us deal with some of these possible objections.

First, one might argue that we did handle only a somehow too "restricted" loading problem, because some the weights were already pre-set. It is possible to answer this by slightly extend the architecture and the learning task. For weights we would like to be set to a certain value (by any learning algorithm), we just had to connect the adjacent units directly to the environment and extend the learning task in an appropriate way.

For example, to train also for the plynomial coefficients c_j we had to introduce additional incoming links or signals to all the "term units" in Fig.1. Furthermore, these term units become regular $\Sigma\Pi$-units, i.e. the output of these units is computed as the sum of two terms – the first one is the product as computed before, the second one is the newly added signal from outside. Finally, the learning task had to be extended. That is – while the original input signals are all set to zero – one association for each coefficient forces the corresponding link to be set correctly.

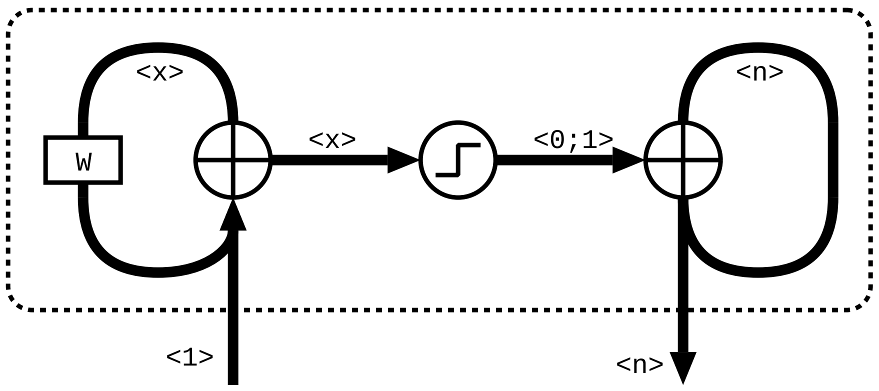

Figure 2: A counter element based on weights

6 Recurrent Networks

A second objection concerning the above model could be that we restricted the set of possible weights to just integers. Although, several accepted neural network paradigms actually operate with integer weights and activations, e.g. the Adaptive Resonance Theory (ART) paradigm is even restricted to only binary values (at least ART1). This argument shall be answered by introducing a recurrent alternative to these integer weights.

The basic idea is to introduce a "counter module" as it is shown in Fig.2 which replaces the incoming integer weights which corresponded to the solution vector for the diophantine equation. With this counter module the integer value of the link weight is replaced by the float weight of a special recurrent link.

The structure of this counter module is the following. The left side unit just summarizes all its input signals. The middle unit is a *threshold unit* which outputs the signal 0.0 if the incoming signal is smaller than some (fixed) threshold θ and 1.0 otherwise. The right side unit again just summarizes all its inputs. Almost all connections operate directly, i.e. the signals are passed immediately without modifications. Only the recurrent connection for the left side unit is "weighted" by some weight $0.0 < w < 1.0$, i.e. the outgoing signal is multiplied by w and then sent back to the left side unit.

The operation is as follows. At some initial time a signal of value 1.0 is sent to the left side unit. After this, that incoming signal is reset again to 0.0. Therefore, just the initial "seed activation" starts circling via the weighted link, each time becoming a little bit smaller. At the same time this signal also goes to the central unit, which then sends signals of value 1.0 to the right side unit, at least as long as the circling activation of the left side unit is larger then the threshold θ of the central unit. The right side unit is just "collecting" the signals from the middle unit. Thus it counts the repetitions in the left side unit as long as its activation is above the threshold θ.

We can conclude that the final, maximum, value which is provided by the right side unit is some integer $n \in \mathbb{N}$ for which $w^n > \theta > w^{n+1}$ holds. Therefore, we have a correspondence between the left side real valued weight w and the integer sum n provided by the right side unit, which can be used instead of the incoming signals in the architecture we already described above and which was equivalent to some diophantine equation (Fig.1). Thus we have also shown

Theorem 2 *The loading problem for (higher order) recurrent neural networks with real weights is unsolvable.*

Iff the training of such a recurrent network could be achieved, then the solution to the corresponding diophantine equation could be reconstructed by the following expression, which relates the sum n to the weight w and the threshold θ (which also could be modified). Simply take $n = \lfloor \frac{\ln \theta}{\ln w} \rfloor$, where $\lfloor x \rfloor$ denotes the so called floor, i.e. the largest integer which is smaller or equal to x. On the other hand we also can construct an appropriate weight w to solve the loading problem, provided that we know a solution to the corresponding diophantine equation by choosing w from the interval $\left(\exp\left(\frac{\ln \theta}{n}\right), \exp\left(\frac{\ln \theta}{n+1}\right)\right)$.

7 Conclusions

First, one should stress once again that the fact that no general algorithm exists for higher order or recurrent networks, which could solve the loading problem (for all its instances), does not imply that *all* instances of this problem are unsolvable or that no solutions exist. Moreover it is also known that the problem of solving linear diophantine equations (instead of general ones) is polynomially computable [9].

One could think, that in most relevant cases – whatever that could mean – or, when we restrict the problem, a sub-class of problems everything might become tractable. But the difference between solvable and unsolvable problems often can be very small. For example, we already noted, that the problem of linear diophantine equations is solvable in polynomial, while if we go to quadratic diophantine equations the problem already becomes NP complete [9]. And for general diophantine the problem is even unsolvable.

In any case, one should interpret these "negative"results on NP complexity as well as on undecidability not as restrictions for neural networks, but as facts indicating their computational power. It was shown that concrete neural networks can be constructed, so that they realize what is called a "universal Turing machine" [16, 3]. Actually, it is also a well known fact that even relatively simple "relatives" of neural networks, so called "cellular automata" also might realize a universal Turing machine [6].

But it is also known that, for example, the so called "halting problem" for Turing machines, i.e. the question if a Turing machine will accept some input string, is *undecidable*. So it might not be astonishing that in the general case some of the problems related to a most general class of computing devices, like neural networks, are also undecidable. In these cases it is the complexity of the problems one might try to solve by using a neural network which simply cannot "disappear" and not some intractability of the neural network approach.

References

[1] Martin Anthony and Norman Biggs. *Computational Learning Theory – An Introduction*, volume 30 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1992.

[2] Avrim L. Blum and Ronald L. Rivest. Training a 3-node neural network is NP-complete. *Neural Networks*, 5:117–127, 1992.

[3] Michael Cosnard, Max Garzon, and Pascal Koiran. Computability properties of low-dimensional dynamical systems. In *Symposium on Theoretical Aspects of Computer Science (STACS '93)*, volume 665 of *Lecture Notes in Computer Science*, pages 365–373, Berlin – New York, 1993. Springer-Verlag.

[4] Martin Davis. Hilbert's tenth problem is unsolvable. *American Mathematical Monthly*, 80:233–269, March 1973.

[5] Michael R. Garey and David S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York, 1979.

[6] Eric Goles and Servet Martinez. *Neural and Automata Networks*, volume 58 of *Mathematics and Its Applications.* Kluwer Academic Publisher, Dordrecht, 1990.

[7] David Hilbert. Mathematische Probleme. *Nachr. Ges. Wiss. Göttingen, math.-phys.Kl.*, pages 253–297, 1900.

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

[9] David S. Johnson. A catalog of complexity classes. In *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 2, pages 67–161. Elsevier – MIT Press, Amsterdam – Cambridge, Massachusetts, 1990.

[10] J. Stephen Judd. *Neural Network Design and the Complexity of Learning.* MIT Press, Cambridge, Massachusetts – London, England, 1990.

[11] Jyh-Han Lin and Jeffrey Scott Vitter. Complexity results on learning by neural networks. *Machine Learning*, 6:211–230, 1991.

[12] Wolfgang Maass. Bounds for the computational power and learning complexity of analog neural nets. Technical report, Institute for Theoretical Computer Science, Technische Universität Graz, Klosterwiesgasse 32/2, A-8010 Graz, Austria, October 1992.

[13] Yuri Matijasevich. Enumerable sets are diophantine. *Dokl. Acad. Nauk.*, 191:279–282, 1970.

[14] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, Massachusetts, 1969. second printing 1988.

[15] Kenneth A. Ribet. Wiles proves taniyama's conjecture; fermat's last theorem follows. *Notices Amer. Math. Soc.*, pages 575–576, 1993.

[16] Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. In *Workshop on Computational Learning Theory (COLT '92)*, pages 440–449, 1992.