10

The Complexity of Learning

10.1 Network functions

In the previous chapters we extensively discussed the properties of multilayer neural networks and some learning algorithms. Although it is now clear that backpropagation is a statistical method for function approximation, two questions remain open: firstly, what kind of functions can be approximated using multilayer neural networks, and secondly, what is the expected computational complexity of the learning problem. We deal with both issues in this chapter.

10.1.1 Learning algorithms for multilayer networks

The backpropagation algorithm has the disadvantage that it becomes very slow in flat regions of the error function. In that case the algorithm should use a larger iteration step. However, this is precluded by the length of the gradient, which is too small in these problematic regions. Gradient descent can be slowed arbitrarily in these cases.

We may think that this kind of problem could be solved by switching the learning algorithm. Maybe there is a learning method capable of finding a solution in a number of steps that is polynomial in the number of weights in the network. But this is not so. We show in this chapter that finding the appropriate weights for a learning problem consisting of just one input-output pair is computationally hard. This means that this task belongs to the class of *NP*-complete problems, for which no polynomial time algorithm is known, because it probably does not exist.

10.1.2 Hilbert's problem and computability

The renowned mathematician David Hilbert indirectly provided the problem whose solution will bring us to the core of the function approximation issue. During his keynote speech in 1900 at the International Congress of Mathematicians in Paris, he formulated 23 problems which he identified as those whose solution would bring mathematical research forward in the 20th Century [19]. The thirteenth problem dealt with the old question of solving algebraic equations. Hilbert's hypothesis was: "It is probable that the root of an equation of seventh degree is such a function of its coefficients that it does not belong to the class of functions representable with nomographic methods, that is, it cannot be represented by a finite composition of functions of two arguments. To decide this, it should be proved that the equation

$$f^7 + xf^3 + yf^2 + zf + 1 = 0$$

of seventh degree cannot be solved using functions of two arguments".

This abstract formulation can be better understood by comparing it to similar simpler problems. Analytic formulas to solve equations of the second, third, and fourth degree, which make use exclusively of elementary arithmetic operations and of the square root function, have been known for a long time [67]. A finite number of steps is needed to get a solution. The roots of the quadratic equation $ax^2 + bx + c = 0$, for example, can be computed using a finite composition of algebraic operations, as shown in Figure 10.1.

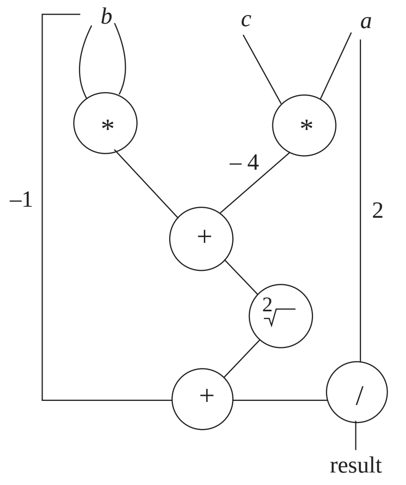

Fig. 10.1. Network for the computation of the roots of quadratic equations

Abel showed that for algebraic equations of degree higher than five, there is no such finite composition of algebraic operations that could compute its roots. Therefore there is no *algebraic formula* to find them and no finite network of algebraic nodes can be built to compute them either. However, the roots of the algebraic equation of seventh degree can be represented as a finite composition of other non-algebraic functions. A method which was very popular before the advent of computers was *nomography*, which works with graphical representations of functions. Figure 10.2 shows an example. The vertical projection of the point where the two curves $u = 2$ and $v = 1.5$ meet gives the value of $F(2, 1.5)$, where F is the function to be computed using the nomographic method. There are many functions that can be represented in this way. The composition of functions of two arguments can be computed nomographically by connecting two graphical representations using special techniques [129]. With nomography it is possible to find an approximate solution for a given equation. However, if Hilbert's hypothesis was correct, this would mean that for equations of degree higher than seven, we cannot find any finite composition of functions of two arguments to compute their roots, and therefore no nomographical composition of the kind discussed above would lead to the solution. This would be a generalization of Abel's result and would restrict the applicability of nomographic methods.

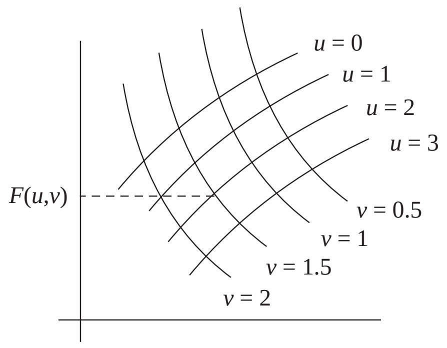

Fig. 10.2. Nomographic representation of the function $F(u, v)$

10.1.3 Kolmogorov's theorem

In 1957, the Russian mathematician Kolmogorov showed that Hilbert's conjecture does not hold. He proved that continuous functions of n arguments can always be represented using a finite composition of functions of a single argument, and addition. The theorem is certainly surprising, since it implicitly says that addition is the only function of more than one argument needed to represent continuous functions with *any* number of arguments. Multiplication, for example, can be rewritten as a composition of functions of one argument and additions, since $xy = \exp(\ln x + \ln y)$.

A modern variant of Kolmogorov's theorem, whose proof can be found in [411], states:

Proposition 14. *Let $f : [0, 1]^n \to [0, 1]$ be a continuous function. There exist functions of one argument g and ϕ_q for $q = 1, \ldots, 2n + 1$ and constants λ_p, for $p = 1, \ldots, n$ such that*

$$f(x_1, x_2, \dots, x_n) = \sum_{q=1}^{2n+1} g\left(\sum_{p=1}^{n} \lambda_p \phi_q(x_p)\right).$$

From the perspective of networks of functions, Kolmogorov's theorem can be interpreted as stating that any continuous function of n variables can be represented by a finite network of functions of a single argument, where addition is used as the only function of several arguments. Figure 10.3 shows the kind of network needed to represent $f(x_1, x_2, \dots, x_n)$.

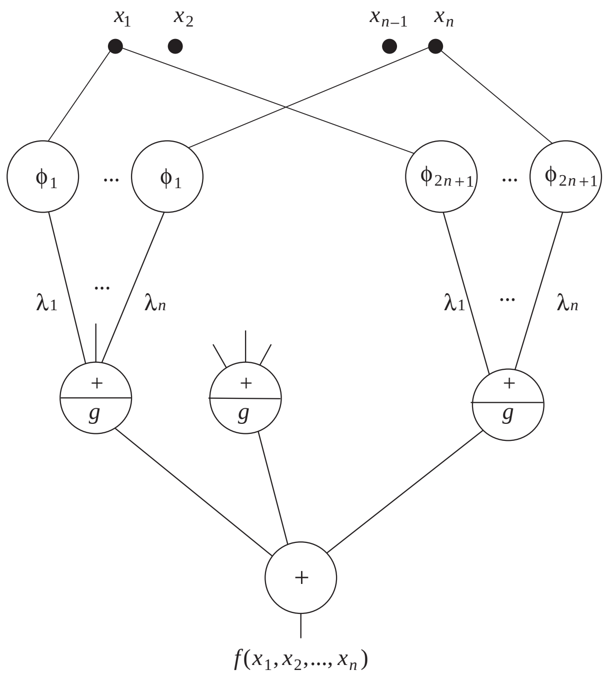

Fig. 10.3. Network for computing the continuous function f

The network is similar to those we have been using. The non-trivial functions ϕ_q can be preselected, so that only the function g and the constants $\lambda_1, \dots, \lambda_n$ have to be found. These *Kolmogorov networks* have been compared to those that have been in use in control theory for many years [90].

Some authors have recently relaxed some of the restrictions imposed by Kolmogorov's theorem on the functions to be approximated and have studied the approximation of Lebesgue integrable functions. Irie and Miyake showed that if the hidden layer contains an unbounded number of elements, a network composed of units with fixed primitive functions at the nodes can be trained by adjusting the network weights [213]. Gallant and White have produced similar results using networks which compute Fourier series [150]. A necessary condition is that the units implement some form of *squashing function*, that is, a sigmoid or any other function of the same general type. Hornik et al. have obtained results for a broader class of primitive functions [203]. If we accept units capable of computing integral powers of the input, then we can implement polynomial approximations of a given function and Weierstrass and Stone's classical result guarantees that any real continuous function can be approximated with arbitrary precision using a finite number of computing units.

10.2 Function approximation

Kolmogorov's theorem is important in the neural networks field because it states that any continuous function can be reproduced *exactly* by a finite network of computing units, whereby the necessary primitive functions for each node *exist*. However, there is a second possibility if we want to approximate functions – we do not demand exact reproducibility but a bounded approximation error. In that case we look for the best possible approximation to a given function f. This is exactly the approach we take with backpropagation networks and any other kind of mapping networks.

10.2.1 The one-dimensional case

What kind of functions can be approximated by neural networks? To answer this question we will discuss first a more special issue. It can be shown that continuous functions $f : [0,1] \to [0,1]$ can be approximated with arbitrary precision. The next proposition deals with this fact, which will be extended in the following section to functions of several arguments.

Proposition 15. *A continuous real function $f : [0,1] \to [0,1]$ can be approximated using a network of threshold elements in such a way that the total approximation error E is lower than any given real $\varepsilon > 0$, that is,*

$$E = \int_0^1 |f(x) - \tilde{f}(x)| dx < \varepsilon,$$

where $\tilde{f}$ denotes the network function.

Proof. Divide the interval $[0,1]$ into N equal segments selecting the points $x_0, x_1, \ldots, x_N \in [0,1]$ with $x_0 = 0$ and $x_N = 1$. Define a function φ_N as follows:

$$\varphi_N(x) = \min\{f(x')|x' \in [x_i, x_{i+1}] \text{ for } x_i \leq x \leq x_{i+1}\}.$$

This function consists of N steps as shown in Figure 10.4. Now, consider φ_N an approximation of f so that the approximation error is given by

$$E_N = \int_0^1 |f(x) - \varphi_N(x)| dx.$$

Since $f(x) \geq \varphi_N(x)$ for all $x \in [0,1]$ the above integral can be written as

$$E_N = \int_0^1 f(x)dx - \int_0^1 \varphi_N(x)dx.$$

The second integral is nothing other than the lower sum of the function f in the interval $[0,1]$ with the segmentation produced by $x_0, x_1, \ldots, x_N$, as is done to define the Riemann integral. Since continuous functions are integrable, the lower sum of f converges in the limit $N \to \infty$ to the integral of f in the interval $[0,1]$. It thus holds that $E_N \to 0$ when $N \to \infty$. For any given real number $\varepsilon > 0$ there exists an M such that $E_N < \varepsilon$ for all $N \geq M$. The function φ_N is therefore the desired approximation of f. We must now show that φ_N can be computed by a neural network.

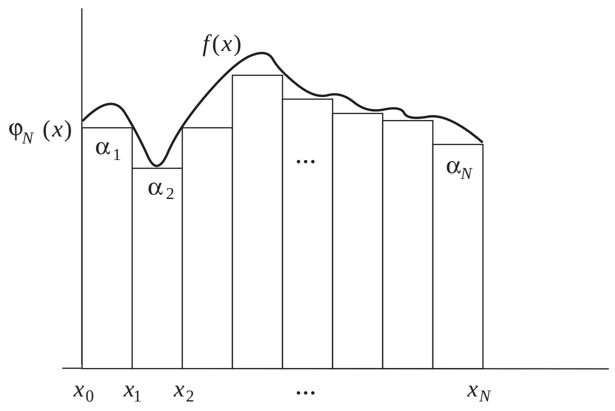

Fig. 10.4. Approximation of f with φ_N

Figure 10.4 shows the graph of φ_N for the interval $[0,1]$. The function is composed of N steps with respective heights $\alpha_1, \alpha_2, \ldots, \alpha_N$. The N segments of the interval $[0,1]$ are of the form $[x_0, x_1), [x_1, x_2), \ldots, [x_{N-1}, x_N)$.

The network shown in Figure 10.5 can compute the step-wise function φ_N. The single input to the network is x. Each pair of units with the weights x_i and x_{i+1} guarantees that the unit with threshold x_i will only fire when $x_i \leq x < x_{i+1}$. In that case the output 1 is multiplied by the weight α_{i+1}. The output unit (a linear element) adds all outputs of the upper layer of units and produces their sum as result. The unit with threshold $x_N + \delta$, where δ is positive and small, is used to recognize the case $x_{N-1} \leq x \leq x_N$.

The network shown in Figure 10.5 therefore computes the function φ_N, which approximates the function f with the desired maximal error. □

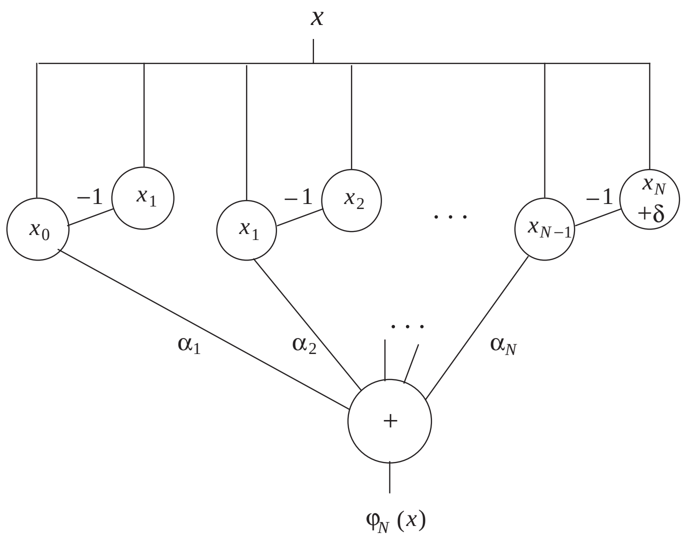

Fig. 10.5. Network for the computation of $\varphi_N(x)$

Corollary 1. *Proposition 15 is valid also for functions* $f : [0, 1] \to (0, 1)$ *with sigmoidal activation.*

Proof. The image of the function f has been limited to the interval $(0, 1)$ in order to simplify the proof, since the sigmoid covers only this interval.

The function f can be approximated using the network in Figure 10.6. The activation of the units with threshold x_i (which is now the bias term $-x_i$) is given by $s_c(x - x_i)$, where

$$s_c(x - x_i) = \frac{1}{1 + \mathrm{e}^{-c(x - x_i)}}.$$

Different values of c produce more or less steep sigmoids. Threshold functions can be approximated with any desired precision and the network of Figure 10.6 can estimate the function φ_N with an approximation error lower than any desired positive bound.

Note that the weights for the edges connecting the first layer of units to the output unit have been set in such a way that the sigmoid produces the desired α_i values as results. It should only be guaranteed that every input x produces a single 1 from the first layer to the output unit. The first layer of the network just finds out to which of the N segments of the interval $[0, 1]$ the input x belongs. $\square$

We can now generalize this results considering the case in which the function to be approximated has multiple arguments.

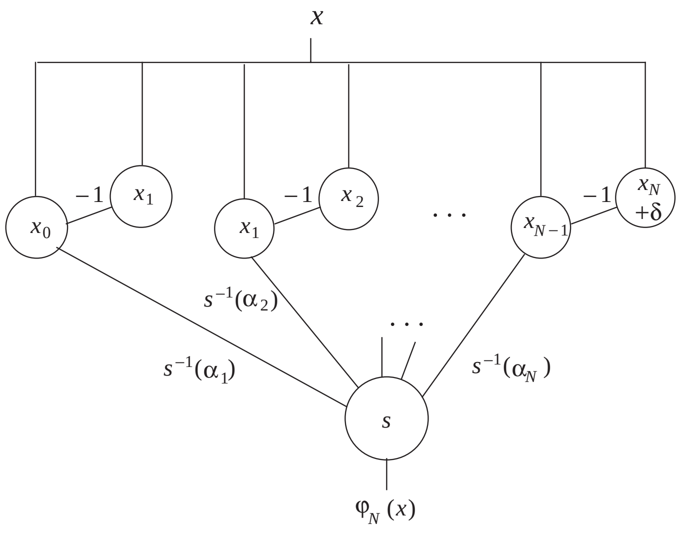

Fig. 10.6. Network for the computation of $\varphi_N(x)$ (sigmoidal units)

10.2.2 The multidimensional case

In the multidimensional case we are looking for a network capable of approximating the function $f : [0,1]^n \to (0,1)$. The network can be constructed using the same general idea as in the previous section. The approximation is computed using "blocks" as shown in Figure 10.7 for the function $\cos(x^2+y^2)$.

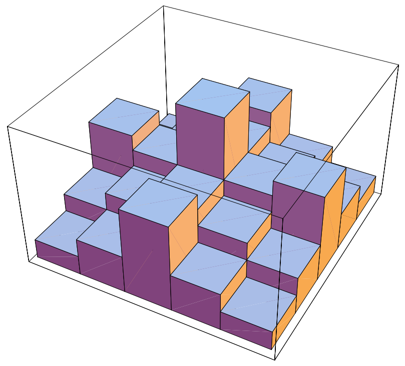

Fig. 10.7. Piecewise approximation of the function $\cos(x^2 + y^2)$

Figure 10.8 shows the necessary network extensions for a two-dimensional function. In the one-dimensional case each interval in the definition domain was recognized by two coupled units. In the two-dimensional case, it is necessary to recognize intervals in the x and y domains. This is done by using a conjunction of the outputs of the two connections as shown in Figure 10.8. The two units to the left are used to test $x_0 \leq x < x_1$. The two units to the right are used to test $y_1 \leq y < y_2$. The unit with threshold 1.5 recognizes the conjunction of both conditions. The output connection has the weight $s_0^{-1}(\alpha_{12})$, so that a unit with the sigmoid as output unit (connected afterwards) can produce the value α_{12}. This number corresponds to the desired approximation to the function f in the interval $[x_0, x_1) \times [y_1, y_2)$.

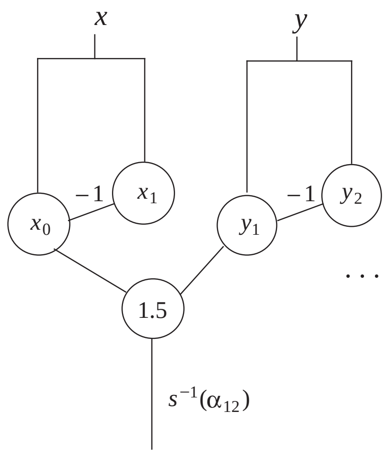

Fig. 10.8. Extension of the network for the two-dimensional case

The two-dimensional network can be completed using this scheme. Other networks for multidimensional cases ($n > 2$) can be crafted using a similar strategy. An arbitrary continuous function can be approximated using this approach but the price that has to be paid is the large number of computing units in the network. In the chapters covering self-organizing networks, we will deal with some algorithms that take care of minimizing the number of computing units needed for a good approximation.

10.3 Complexity of learning problems

Kolmogorov's theorem and Proposition 15 describe the approximation properties of networks used for the representation of functions. In the case of neural networks not only the network architecture is important but also the definition of a learning algorithm. The proof of Kolmogorov's theorem does not give any hint about the possible choice of the primitive functions for a given task, since it is non-constructive. Proposition 15 is different in this respect. Unsupervised learning can recreate something similar to the constructive approach used in the proof of the theorem.

The *general learning problem* for a neural network consists in finding the unknown elements of a given architecture. All components of the network can be partially or totally unknown, for example, the activation function of the individual units or the weights associated with the edges. However, the learning problem is usually constrained in such a way that the activation functions are chosen from a finite set of functions or some of the network weights are fixed in advance.

The *size* of the learning problem depends on the number of unknown variables which have to be determined by the learning algorithm. A network with 100 unknown weights is a much harder computational problem than a network with 10 unknown weights, and indeed much harder than the factor 1:10 would suggest. It would be helpful if the learning time could be bounded by a polynomial function in the number of variables, but this is not so.

It has been proved that the general learning problem for neural networks is *intractable*, that is, it cannot be solved efficiently for all possible instances. No algorithm is known which could solve the learning problem in polynomial time depending on the number of unknown variables. Moreover, it is very improbable that such an algorithm exists. In the terminology of complexity theory we say that *the general learning problem for neural networks is* NP-*complete*.

10.3.1 Complexity classes

Before introducing the main results concerning the complexity of learning problems, we will explain in a few pages how to interpret these results and how complexity classes are defined.

When a computational problem can be solved using a certain algorithm, the very next issue, after solvability, is the time and space complexity of the solution method. By *space complexity* we denote the memory required for the algorithmic solution once a computational model has been adopted. The *time complexity* refers to the number of algorithmic steps executed by the computational model. We are interested in the behavior of the algorithm when applied to larger and larger problems. If the index for the size of the problem is n and the number of execution steps grows asymptotically according to n^2, we speak of a quadratic time complexity of the algorithm.

Normally, space complexity is not considered a primary issue, since the computational models are provided with an infinitely large memory. What is most interesting for applications is time complexity and therefore, when we simply speak of complexity of an algorithm, we are actually referring to its time complexity.

Computational problems can be classified in a hierarchical way according to the complexity of all possible algorithms for their solution. A very common approach is to use a Turing machine as the computational model in order to help us analyze algorithms. The size n of a problem is defined as the length of the input for such a machine, when an adequate coding method has been adopted [156]. If an algorithm exists that is capable of solving *any* instance I of a problem X of size n in $p(n)$ steps, where $p(n)$ denotes a polynomial on the variable n, then X is a member of the set P of problems solvable in polynomial time. However if the solution of I requires an exponential number of steps, then X is a member of the class E of problems solvable only in exponential time. A simple example is the decoding of a binary number m as a string of m ones. Since at each step at most a single 1 can be produced as output, the number of execution steps grows exponentially with the size of the problem, which is in this case the number of bits in the binary coding for m.

However, there are some problems for which it is still unknown whether or not they belong to the class P. It has not been possible to find a polynomial time algorithm to solve any of their instances and it has also not been possible to prove that such an algorithm does not exist (which would in fact put them out of the class P). Such problems are considered *intractable*, that is, there is an inherent difficulty in dealing with them which cannot be avoided using the most ingenious methods.

The class *NP* (*nondeterministic polynomial*) has a broader definition than the class P. It contains all problems for whose solution no polynomial algorithm is known, but for which a guessed solution can be checked in polynomial time. We can explain the difference between both possibilities taking the Traveling Salesman Decision Problem (TSDP) as our benchmark. For a set of n cities in the Euclidian plane whose relative distances are known, we want to know if there is a path which visits all cities, whose total length is smaller than a given constant L. No one has been able to design an algorithm capable of solving every instance of this problem in polynomial time in the number of cities. However a proposed path can be checked in linear time: its total length is computed and we check if all cities have been visited. If more than n cities are given in the path we can reject it as invalid without additional computations. This fact, that a solution can be checked in polynomial time whereas for the problem itself no polynomial time algorithm is known, is what we mean when we say that the TSDP is a member of the class *NP*.

The class of *NP* problems derives its name from the computational model which is used for theoretical arguments and which consists of a nondeterministic computation performed in polynomial time. If a Turing machine is provided with an *oracle*, which is just an entity capable of guessing answers for a given problem, then all we need is to check the solution proposed by the oracle. The oracle works nondeterministically, in the sense that it selects one solution out of the many possible (when they exist) in a random manner. If the checking can be done in polynomial time, the problem is a member of the class *NP*. One can also think of an oracle as a parallel Turing machine which inspects all possible solutions simultaneously, eventually selecting one of the successful ones [11].

The most important open problem in complexity theory is whether the classes P and *NP* are identical or different. Theoreticians expect that it will

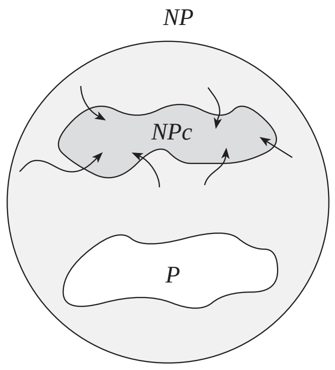

Fig. 10.9. Schematic representation of the classes *NP*, *P*, and *NPc*

eventually be possible to prove that $P \neq NP$, because otherwise a single Turing machine would be in some sense computationally equivalent to the model with an unlimited number of Turing machines. There has been some speculation that although true, the inequality $P \neq NP$ could be shown to be not provable. It has in fact been shown that in some subsets of arithmetic this is actually the case.

Obviously it is true that $P \subseteq NP$. If a problem can be solved in polynomial time, then a nondeterministic solution can also be checked in polynomial time. Just simulate the nondeterministic check by finding a solution in polynomial time. It is not known if there are some problems which belong to the class *NP* but not to the class P. Possible candidates come from the class *NPc* of so-called *NP*-complete problems.

A problem X is *NP*-complete if any other problem in *NP* can be reduced to an instance of X in polynomial time. This means that a Turing machine can be provided with a description of the original problem on its tape so that it is transformed into a description of an instance of X which is equivalent to the original problem. This reduction must be performed in polynomial time. The class *NPc* is important because if an algorithm with polynomial complexity could be found that could solve any of the problems in the class *NPc*, then any other problem in the class *NP* would be also solvable in polynomial time. It would suffice to use two Turing machines. The first one would transform the *NP* problem into an instance of the *NPc* problem with a known polynomial algorithm. The second would solve the transformed problem in polynomial time. In this sense the class *NPc* contains the most difficult problems in the class *NP*. Figure 10.9 shows a diagram of the class *NP* assuming that $NP \neq P$. The arrows in the diagram illustrate the fact that any problem in the class *NP* can be transformed into a problem in the class *NPc* in polynomial time.

Theoreticians have been able to prove that many common problems are members of the class *NPc*. One example is the *Traveling Salesman Decision Problem*, already mentioned, another the *Satisfiability Problem*, which will be discussed in the next section. The usual method used to prove that a problem A belongs to the class *NPc* is to find a polynomial transformation of a well-known problem B in *NPc* into an instance of the new problem A. This makes it possible to transform any problem in *NP* into an instance of A in polynomial time (going of course through B), which qualifies A as a member of the class *NPc*.

Proving that a problem belongs to the class *NPc* amounts to showing that the problem is computationally difficult. Theoreticians expect that no polynomial time algorithm will ever be found for those problems, that is, they expect that someday it will be possible to prove that the inequality $P \neq NP$ holds. One such computationally hard task is the general learning problem.

10.3.2 *NP*-complete learning problems

It can be shown with the help of the satisfiability problem that an *NP*-complete problem can be reduced to an instance of a learning problem for neural networks in polynomial time. The satisfiability problem is defined in the following way.

Definition 10. *Let V be a set of n logical variables, and let F be a logical expression in conjunctive normal form (conjunction of disjunctions of literals) which contains only variables from V. The satisfiability problem consists in assigning truth values to the variables in V in such a way that the expression F becomes true.*

A classical result from Cook guarantees that the satisfiability problem is a member of the class *NPc* [88]. This result also holds for such expressions F with at most three literals in each disjunction (this problem is called 3SAT in the literature). We will transform 3SAT into a learning problem for neural networks using the network shown in Figure 10.10. We give a simpler proof than Judd but keep the general outline of his method [229].

The activation of the individual units will be computed using threshold functions. An expression F in conjunctive normal form, which includes only the n variables $x_1, x_2, \ldots, x_n$ is given. The disjunctions in the normal form contain at most three literals. We want to find the truth values of the variables that make F true.

The network in Figure 10.10 has been constructed reserving for each variable x_i, a weight w_i and a computing unit with threshold 0.5. The output of the i-th unit in the first layer is interpreted as the truth value assigned to the variable x_i. The units with threshold -0.5 are used to negate the truth value of the variables x_i. The output of each of these units can be interpreted as $\neg x_i$. The third layer of units (counting from top to bottom) implements the disjunction of the outputs of the units connected to them (the clauses in the normal form). If any connection arriving to a unit in the third layer transports the value 1, the unit fires a 1. The connections in Figure 10.10 correspond to

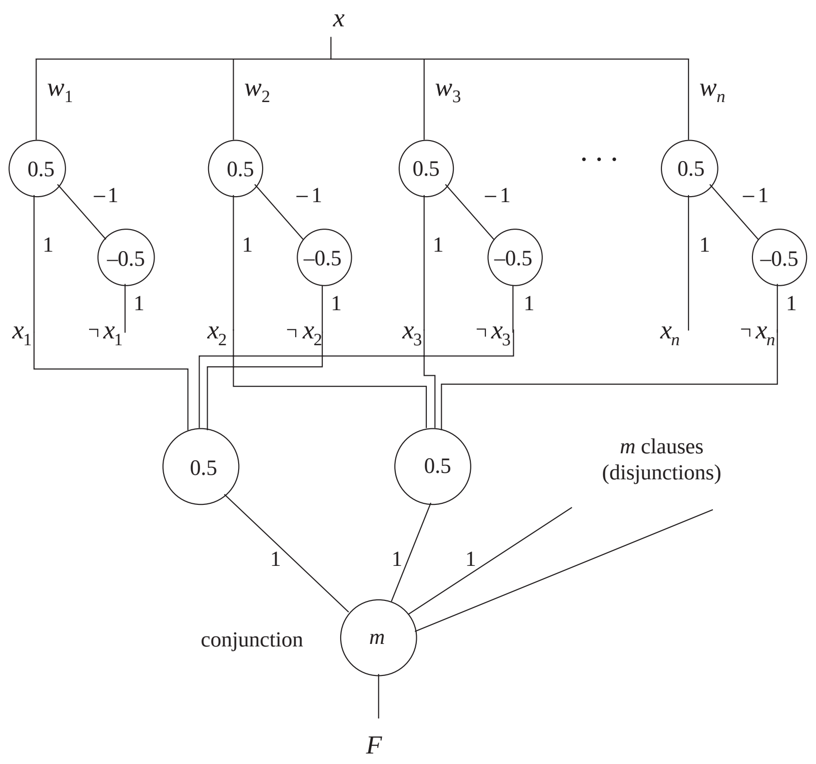

Fig. 10.10. Equivalent network for the 3SAT problem

the following two clauses: $x_1 \vee \neg x_2 \vee \neg x_3$ and $x_2 \vee x_3 \vee \neg x_n$. The last unit implements the conjunction of the disjunctions which have been hard-wired in the network. We assume that the expression F contains m clauses. The value m is used as the threshold of the single unit in the output layer. The expression F is true only if all disjunctions are true.

After this introductory explanation we can proceed to prove the following result:

Proposition 16. *The general learning problem for networks of threshold functions is* NP*-complete.*

Proof. A logical expression F in conjunctive normal form which contains n variables can be transformed in polynomial time in the description of a network of the type shown in Figure 10.10. For each variable x_i a weight w_i is defined and the connections to the units in the third layer are fixed according to the conjunctive normal form we are dealing with. This can be done in polynomial time (using a suitable coding) because it holds for the number m of different possible disjunctions in a 3SAT formula that $m \leq (2n)^3$.

After the transformation, the following learning problem is to be solved: an input $x = 1$ must produce the output $F = 1$. Only the weights in the network are to be found.

If an instantiation A with logical values of the variables x_i exists, such that F becomes true, then there exist weights $w_1, w_2, \ldots, w_n$ that solve the learning problem. It is only necessary to set $w_i = 1$ if $x_i = 1$. If $x_i = 0$ we set $w_i = 0$, that is, in both cases $w_i = x_i$. The converse is also true: if there exist weights $w_1, w_2, \ldots, w_n$ that solve the learning problem, then the instantiation $x_i = 1$, if $w_i \geq 0.5$, and $x_i = 0$ otherwise, is a valid instantiation that makes F true.

This proves that the satisfiability of logical expressions can be transformed into a learning problem for neural networks. We must now show that the learning problem belongs in the class *NP*, that is, that a solution can be checked in polynomial time.

If the weights $w_1, w_2, \ldots, w_n$ are given, then a single run of the network can be used to check if the output F is equal to 1. The number of computation steps is directly proportional to the number n of variables and to the number m of disjunctive clauses, which is bounded by a polynomial in n. The time required to check an instantiation is therefore bounded by a polynomial in n. This means that the given learning problem belongs to the class *NP*. □

It could be argued that the learning problem stated above is more difficult to solve than in the typical case, because many of the network weights have been selected and fixed in advance. Usually all weights and thresholds are considered as variables. One could think that if all weights are set free to vary, this opens more regions of weight space for exploration by the learning algorithm and that this could reduce the time complexity of the learning problem. However this is not so. Proposition 16 is still valid even when all weights are set free to be modified. The same is true if the threshold function is substituted by a sigmoid [229]. It has even been shown that the training of a three-unit network can be *NP*-complete for a certain training set [324].

10.3.3 Complexity of learning with AND-OR networks

Since the general learning problem is *NP*-complete, we can try to analyze some kinds of restricted architectures to find out if they can be trained in polynomial time.

As a second example we consider networks of units which can only compute the AND or the OR function. This restriction limits the number of available combinations and we could speculate that this helps to avoid any combinatorial explosion. However, this is not the case, and it can be shown that the learning problem for this kind of network remains *NP*-complete.

In the proof of Proposition 16 we constructed a network which mirrored closely the satisfiability problem for logical expressions. The main idea was to produce two complementary output values for each logical variable, which were identified as x_i and $\neg x_i$. These outputs were connected to disjunction units, and the output of the disjunctions was connected to the output unit (which computed the AND function). Any logical expression can be hardwired in such way. For the proof of the following proposition we will use the same approach, but each unit will be more flexible than before.

Proposition 17. *The learning problem for neural network architectures whose units can only compute the AND or the OR function is* NP*-complete.*

Proof. Just as we did in the proof of Proposition 16, we will show that a polynomial time learning algorithm for the networks considered could also solve the 3SAT problem in polynomial time. A logical expression F in conjunctive normal form, with at most three literals in each disjunction, will be computed by the network shown in Figure 10.11.

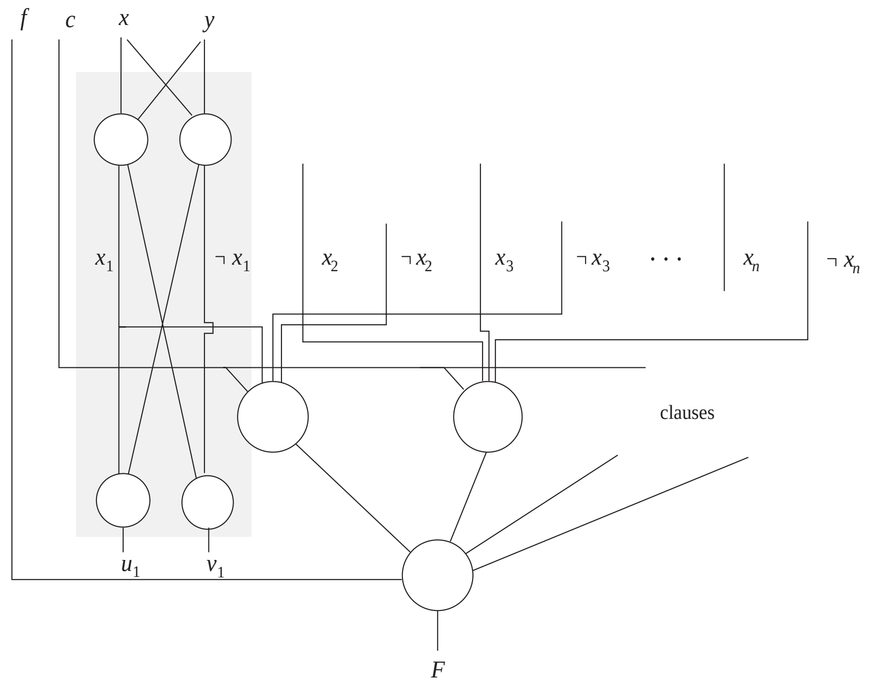

Fig. 10.11. Equivalent network for the 3SAT problem with AND-OR units

The arrangement is similar to the one in Figure 10.10, but with some differences. There are now four inputs f, c, x, y. The values x and y are connected to each column of units in the network (to avoid cluttering the diagram we show only one module for each variable). The shaded module is present n times in the network, that is, once for each logical variable in the expression F. The outputs of each module (x_i and $\neg x_i$) are shown in the diagram.

The weights of the network are unknown, as are the thresholds of each unit (whose activation function is a simple threshold function). Only those combinations of weights and thresholds are allowed that lead to the computation of the functions AND and OR at the nodes. The learning problem for the network is given by the following table, which consists of three input-output pairs:

	x	y	f	c	F	u_1	v_1	u_2	v_2	$\cdots$	u_n	v_n
(a)	0	0	1	1	1	0	0	0	0	$\cdots$	0	0
(b)	0	0	1	0	0	0	0	0	0	$\cdots$	0	0
(c)	0	1	1	1	1	0	1	0	1	$\cdots$	0	1

Since the individual units only compute AND or OR functions, this means that the input $x = y = 0$ makes all output lines $x_1, \neg x_1, x_2, \neg x_2, \ldots, x_n, \neg x_n$ equal to 0. From row (b) of the training set table, we know that $F = 0$. Since the only input to the output unit are the disjunctions of the literals (which are all equal to 0) and $f = 1$, this means that the output unit must compute the AND function. Row (b) therefore determines the kind of output unit that we can use.

Row (a) of the learning problem tells us that the "clause" units, to which the values x_i and $\neg x_i$ are hard-wired, must compute the OR function. This is so because in row (a) $F = 1$ and $c = 1$. Note that c is connected to all the clause units and that all literals are zero (since, again, $x = y = 0$). The only possibility to get $F = 1$ is for all clause units to produce a 1, so that the AND output unit can also produce a 1. This can only happen if the clause units are disjunctions.

Rows (a) and (b) together force the network to compute a conjunctive normal form. Since we have taken care to connect the literals x_i and $\neg x_i$ in the order required by expression F, we have a similar situation to the one we had in Figure 10.10.

Row (c) of the learning problem provides us with the final argument to close the proof. Now we have $x = 0$ and $y = 1$. Since we want the values of u_i and v_i to be complementary, as row (c) demands, this can only happen if x_i and $\neg x_i$ are themselves complementary. Otherwise u_i would be always equal to v_i, because the AND and the OR function produce the same outputs for the input combinations (0,0) and (1,1). The connections x_i and $\neg x_i$ are thus in fact complementary.

If the learning problem has a solution, the values of the connections $x_1, x_2, \ldots, x_n$ provide the instantiation of the logical variables that we are looking for to satisfy the expression F. The additional details of the proof can be worked out as in the proof of Proposition 16. □

Judd has also proved a more general theorem. Even if all linear threshold functions can be selected as activation functions, the learning problem remains *NP*-complete [229].

10.3.4 Simplifications of the network architecture

The complexity results of the last section show that training neural networks is a computationally hard problem. If the architecture of the network is only loosely defined and if the training algorithm must find a valid configuration in a large region of weight space, then no polynomial time algorithm can be used.

This situation makes it necessary to improve the known training methods for a fixed size of the learning problem. We discussed this when we analyzed fast variations of backpropagation in Chap. 8. A more promising approach for breaking the "curse of dimensionality", brought by the combinatorial explosion of possible weight combinations, is to test different kinds of simplified architectures. Judd has analyzed some kinds of "flat" neural networks to try to discover if, at least for some of them, polynomial time learning algorithms can be designed.

The motivation behind the theoretical analysis of flat neural networks is the structure of the human cortex. The cortex resembles a two-dimensional structure of low depth. It could be that this kind of architecture can be trained more easily than fully connected networks. However, Judd could show that learning in flat two-dimensional networks is also *NP*-complete [228]. Another kind of network, rather "unbiological" one-dimensional strings of low depth, can be trained in linear time. And if only the average case is considered (and not the worst-case, as usually done in complexity arguments) the training time can even be reduced to a constant, if a parallel algorithm is used.

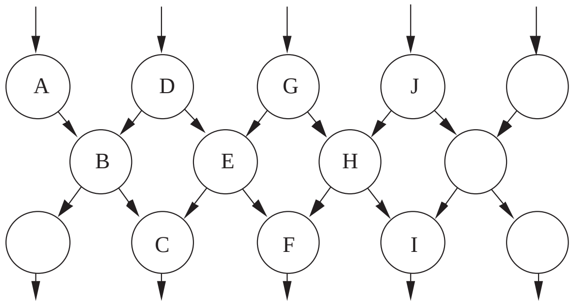

Fig. 10.12. A network of three layers in one direction

Figure 10.12 shows a "one-dimensional" network. The inputs come from the top and the output is produced by the third layer of units. The interconnections are defined in such a way that each unit propagates signals only to its neighbors to the left and to the right. No signal is propagated over more than three stages. The *i*-th bit of the input can affect at most three bits of the output (Figure 10.12).

Flat three-stage networks can be built out of modules that determine the output of a unit. The output of unit C, for example, depends only on information from the units A, B, D, E, and G. These units constitute, together with C, an *information column*. The units D, E, F, G, H, and J constitute the neighboring column. Both structures overlap and it is necessary to find a correct combination for a given learning problem. Assume that the input and the output is binary and that the units in the network are threshold elements of two inputs (which can compute 14 different logical functions). Each information column made of 6 units can assume any of 14^6 configurations. Not all of them are compatible with the configuration of the neighboring column. The combinatorial problem can be solved in two steps: firstly, the valid configurations of the columns are determined using the given input and output vectors. Secondly, only those configurations compatible with those of the neighbors are accepted. In a one-dimensional network with n output bits we must combine the output of n different columns. Judd has shown that this computation can be performed in linear time [229]. In another paper he took the next step and assumed that a processor is available to train each column and considered the average case [230]. Some computer experiments show that in the average case, that is, when the input-output pairs are selected randomly, the time required for coordination of the information columns is bounded by a constant. The network can be trained in constant time independently of its width.

Although the one-dimensional case cannot be considered very realistic, it gives us a hint about a possible strategy for reducing the complexity of the learning problem: if the network can be modularized and the information exchange between modules can be limited in some way, there is a good chance that an efficient learning algorithm can be found for such networks. Some of these ideas have been integrated recently into models of the human cortex [78]. The biological cortex has a flat, two-dimensional structure with a limited number of cortical layers and a modular structure of the cortical columns [360]. In Burnod's model the *cortical columns* are the next hierarchical module after the neurons, and are also learning and functional substructures wired as a kind of cellular automata. It seems therefore that modularization of the network structures, in biological organisms as well as in artificial neural networks, is a necessary precondition for the existence of efficient learning algorithms.

10.3.5 Learning with hints

The example of the one-dimensional networks illustrates one method capable of stopping the combinatorial explosion generated by learning problems. Another technique consists in considering some of the properties of the function to be modeled before selecting the desired network architecture. The number of degrees of freedom of the learning problem can be reduced exploiting "hints" about the shape or the properties of the function to be approximated.

Assume that we want to approximate a real *even* function f of n arguments using a three-layered network. The network has n input sites and a single output unit. The primitive function at the units is the symmetric sigmoid, i.e., an odd function. We can force the network to compute exclusively even functions, that is those for which it holds that $\varphi(x_1, x_2, \ldots, x_n) = \varphi(-x_1, -x_2, \ldots, -x_n)$. This can be guaranteed by replicating each unit and each weight in the second layer. Figure 10.13 shows how to interconnect the units in the network. Both inputs, x_i or $-x_i$, lead to the same output of the network. If the whole network is hardwired following this approach (for each single variable), it will only be capable of computing even functions. This produces a reduction of the search region in the space of functions [5].

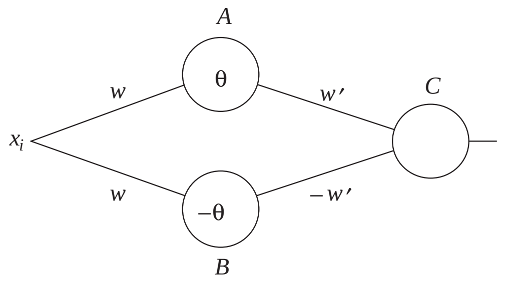

Fig. 10.13. Network to compute even functions

This is the general method to include hints or conditions in a network. If backpropagation is used as the learning method we must also take care to keep the identity of all duplicated weights in the network. We already saw in Chap. 7 how this can be done. All corrections to a weight with the same name are computed independently, but are added before making the weight update. In the case where the weights have the same name but a different sign, the weight corrections are multiplied by the corresponding sign before being added. In this way we keep the original structure of the network, and weights which should be identical remain so (up to the sign).

Some other kinds of invariance are more difficult to implement. Duplicating the network and comparing the respective output values helps to do this. Assume that a network must produce the same real results when two different n-dimensional input values $\mathbf{x}_1$ and $\mathbf{x}_2$ are presented. The difference between the two vectors can reflect some kind of invariance we want to enforce. The vector $\mathbf{x}_2$ could be equal to $-\mathbf{x}_1$ or the components of $\mathbf{x}_2$ could be some permutation of the components of $\mathbf{x}_1$. We would like to compute a function invariant under such modifications of the input.

Figure 10.14 shows the general strategy used to solve this problem. A single untrained network produces the output y_1 for the input $\mathbf{x}_1$ and the output y_2 for the input $\mathbf{x}_2$. The network can be forced to bring y_1 and y_2 closer by minimizing the function $(y_1 - y_2)^2$. This is done by duplicating the network and by performing the additional computation $(y_1 - y_2)^2/2$. Since we want the network to produce a target value t, the difference between y_2 and t must also be minimized. The right side of the additional computing units shown in Figure 10.14 shows the difference that must be minimized, whereas the left side contains the derivative according to y_2. The extended network of Figure 10.14 can be trained with standard backpropagation.

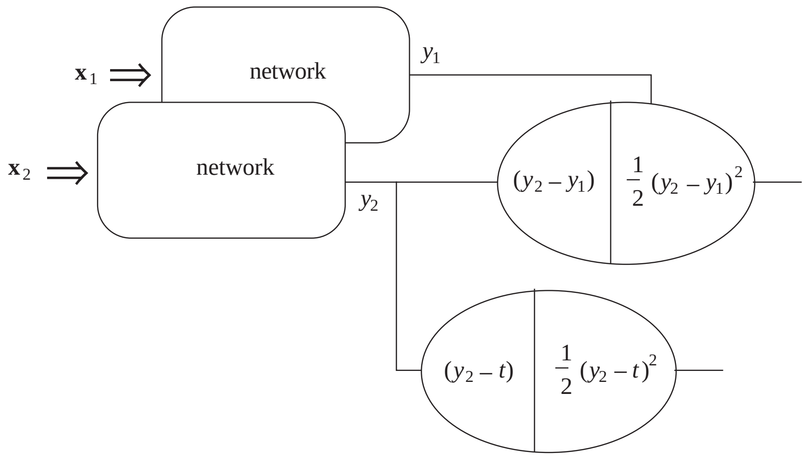

Fig. 10.14. Extended network to deal with invariants

The learning algorithm must take care of duplicated weights in the network as was discussed in Chap. 7. Using gradient descent we try to force the network to produce the correct result and to respect the desired invariance. An alternative approach is to specialize the first layers of the network to the production of the desired invariance and the last ones to the computation of the desired output.

Hints, that is, knowledge about necessary or desired invariances of the network function, can also be used to preprocess the training set. If the network function must (once again) be even, we can expand the training set by introducing additional input-output pairs. The new input data is obtained from the old just by changing the sign, whereas the value of the target outputs remains constant. A larger training set reduces the feasible region of weight space in which a solution to the learning problem can be found. This can lead to a better approximation of the unknown function if the VC dimension of the search space is finite [6]. The technique of "learning with hints" tries to reduce the inherent complexity of the learning problem by reducing the degrees of freedom of the neural network.

10.4 Historical and bibliographical remarks

A. N. Kolmogorov's theorem was rediscovered in the 1980s by Hecht-Nielsen [1987b] and applied to networks of functions. He relied heavily on the work of Lorentz]1976], who had described the full scope of this theorem some years earlier. Since then many different types of function networks and their properties have been investigated, so that a lot of results about the approximation properties of networks are already available.

Minsky and Papert [1969] had already made some complexity estimates about the convergence speed of learning algorithms. However, at that time it was not possible to formulate more general results, as the various classes of complexity were not defined until the 1970s in the work of Cook [1971] and others. Karp [1972] showed that a broad class of problems is *NP*-complete. In his Turing Award Lecture he described the intellectual roots of this complexity research.

In his dissertation Judd [1990] presented the most important theorems on the complexity of learning algorithms for neural networks. His results have been extended over the last few years. It has been shown that when the number of degrees of freedom and combinational possibilities of evaluation elements goes beyond a certain threshold, the learning problems become *NP*-complete. Parberry has collected many interesting results and has helped to systematize the study of complexity of neural networks [335].

The efforts of authors such as Abu-Mostafa, Judd, and others to reduce the complexity of the learning problem are reflected in the research of authors such as Kanerva [1992], who have set up simpler models of biological networks in order to explain their great ability to adapt and their regular column structure.

The existence theorems in this chapter and the complexity estimations made for the learning problem may appear superfluous to the practitioner. However, in a field in which numerical methods are used intensively, we must be aware of the limits of efficient computations. Only on this basis can we design network architectures that will make faster solutions to learning problems possible.

Exercises

1. Compute the approximation error of a sigmoid to the step function for different values of the constant c, where $s(x) = 1/(1+\exp(-cx))$. What is the approximation error to a function approximated by the network of Figure 10.6?
2. Rewrite the function $((xy/z) - x)/y$ using only addition and functions with one real argument.
3. Prove that the network of Figure 10.13 computes an even function.
4. Propose a network architecture and a learning method to map a set of points in $\mathbb{R}^5$ to a set of points in $\mathbb{R}^2$, in such a way that points close to each other in $\mathbb{R}^5$ map as well as possible to neighboring points in $\mathbb{R}^2$.