Neural Networks as Universal Approximators, and the Issue of Depth

Chapter 2 *of* DEEP LEARNING, Part I

by

Rita Singh *and* Bhiksha Raj

DEEP LEARNING

FIRST EDITION

PUBLISHED INDEPENDENTLY BY BHIKSHA RAJ AND RITA SINGH

Some base artwork licensed from shutterstock.com

First printing, September 2020 Deep Learning: ISBN: 978-1-7344465-8-6

Preface

This is Chapter 2 (of Part I) of our book titled DEEP LEARNING.

This book contains nine parts with multiple chapters under each, numbered consecutively throughout the book. The nine parts of DEEP LEARNING are listed below.

Part I	:	Introductory Block
Part II	:	Training
Part III	:	Convolutional Neural Networks
Part IV	:	Recurrent Networks
Part V	:	What a Network Learns
Part VI	:	Networks as Generators
Part VII	:	Hopfield Networks and Boltzmann Machines
Part VIII	:	Reinforcement Learning
Part IX	:	Applications in AI

Deep Learning is a rapidly evolving field, and it is our intention to update the parts and chapters of this book synchronously with the developments in the field. As such, the content of this chapter may be periodically updated, and (rarely) new parts or chapters may be added to this book's milieu.This book will also be paired with a supplementary volume called DEEP LEARNING: EXTENSIONS AND NEW PARADIGMS, which will contain code, exercises and new ideas (tested or proposed) for projects and research.

The style of this book is highly pictorial, and is designed so to make it accessible to readers from a wide range of backgrounds. If you, as a reader, have comments, spot errors or find any part of it lacking in some aspect, please feel free to email us at bhiksha@gmail.com or rita.singh@gmail.com.

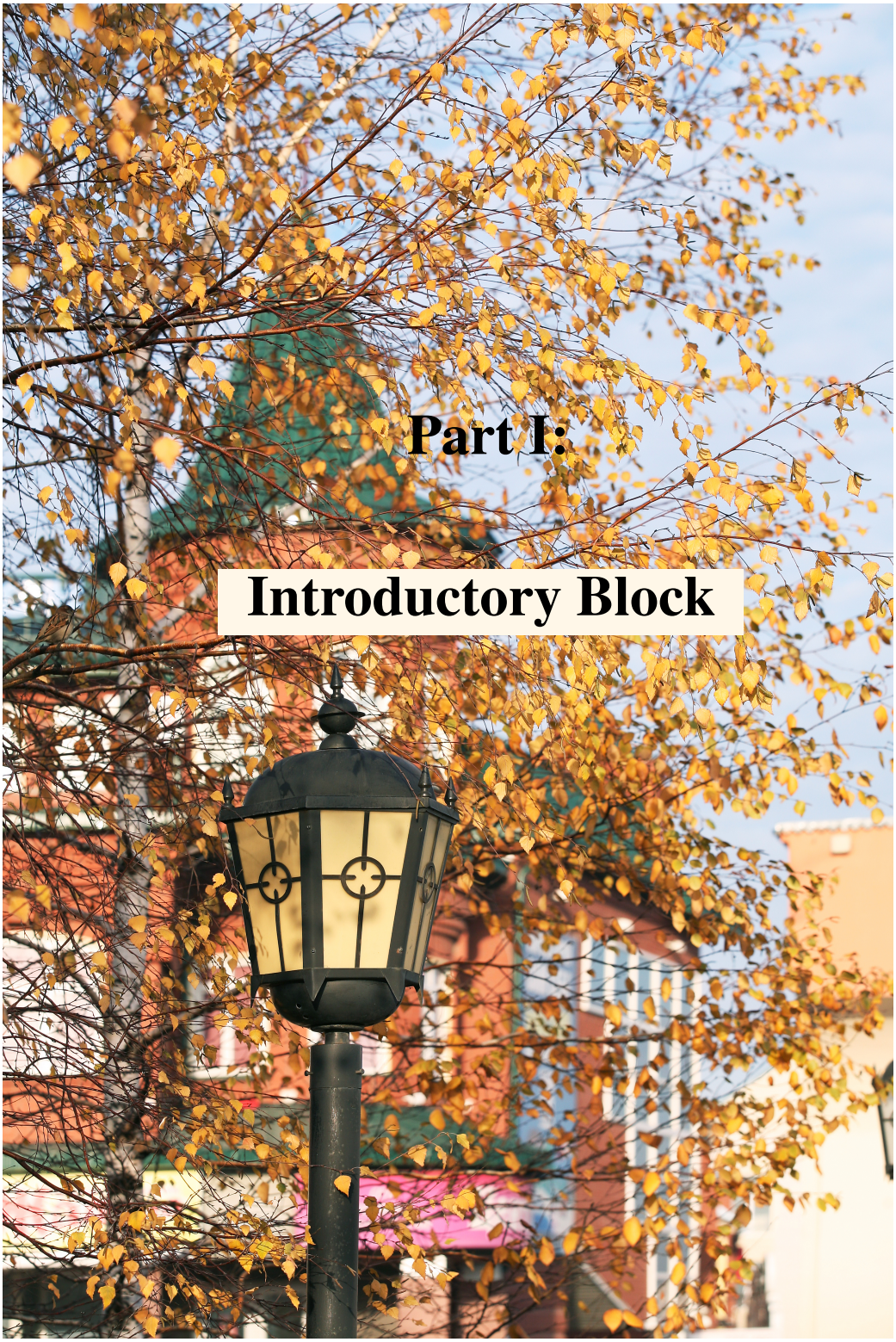
We hope you enjoy this book, and are able to learn easily from it.

Pittsburgh, Pennsylvania.

Bhiksha Raj and Rita Singh
July 2022

Contents

List of Figures



Part I:

Introductory Block

Chapter 2

Neural networks as universal approximators, and the issue of depth

In Chapter 1 we discussed the chronological (and logical) development of neural networks, and saw how neural networks can be viewed as funtions that can model almost any AI task. They can do so because they can operate on discrete or continuous-valued inputs, generate discrete or continuous valued outputs, represent arbitrarily complex decision boundaries in continuous spaces, and approximate arbitrary input-output relations. In this chapter, we will study this topic in greater detail: how neural networks can approximate arbitrary functions, and the limitations of this capacity.

2.1 The perceptron revisited

To be able to accomplish "intelligent" tasks, it was believed that computing machines must emulate the human brain. The basic formalisms for neural networks were focused on realizing this goal. Over time, as the structure and functioning of neurons in the brain came to be better understood, direct analogies to those were created within mathematical frameworks. The perceptron, and later, networks of perceptrons emerged from these attempts.

However, while neural networks sought to approximately mimic the networked structure of the brain, the scale of the brain's structure in terms of the number

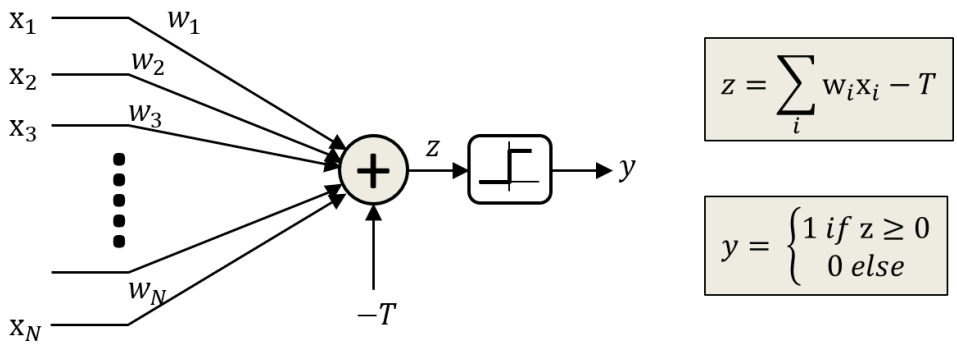
of neurons was difficult to emulate due to computational and logistic limitations. The reasons will become clearer as we learn more about these computational approximations, some of which we discussed in Chapter 1. We must note in this context that the human brain is a massive structure comprising billions of neurons. An average human brain has about 86 billion neurons!

Let us understand further how neural networks sought to mimic the structure of the brain. For this, we must first revisit the Perceptron. We saw in Chapter 1 that each neuron (fashioned after the neurons in the brain) can be approximated with a unit that "fires" or generates an output different from 0 when some function of its inputs exceeds some pre-set threshold. In electrical engineering terms, the perceptron is a *threshold gate*, and implements *threshold logic*. Fig. 2.1 recalls the basic perceptron from Chapter 1.

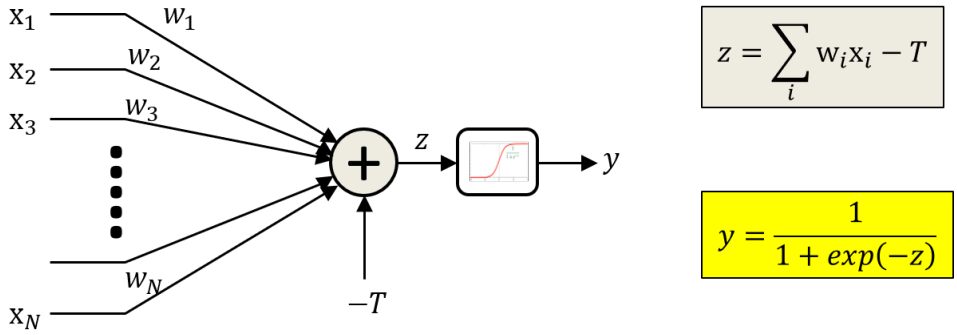
The perceptron in Fig. 2.1(a) is represented as a threshold unit that fires if the weighted sum of inputs exceeds a threshold T. However the figure shows a slightly different way of visualizing the perceptron. Here we first compute an *affine function* z of the inputs $x_1, \cdots, x_N$, which is a weighted sum of inputs plus a bias (which is $-T$). z is then put through an "activation" function, which is a thresholding function that outputs a 1 if the input is zero or greater (i.e. positive), and 0 otherwise.

Fig. 2.1(b) shows a "soft" perceptron, which uses a different activation function, a *sigmoid*. A sigmoid is simply a smooth, continuous version of the threshold function. Unlike the threshold activation, which is discontinuous at 0, the sigmoid is continuous everywhere and its output varies smoothly and monotonically from 0 to 1 as its input increases from $-\infty$ to ∞. Later in this chapter (and in future chapters) we will discuss the benefits of replacing the discontinuous threshold activation by a continuous activation.

The threshold and sigmoid are not the only possible activations that may be applied to the affine value z. Some popular formulations of continuous activations are shown in Fig. 2.2. Of the examples shown, the $tanh$ function is in fact simply a scaled and shifted version of the sigmoid. The threshold, sigmoid and tanh are all instances of "squashing" functions, which squash the infinite span of the input



(a) A perceptron is a unit (depicted as a circle above) that receives many inputs. It has a weight associated with each incoming input. If the weighted sum of all of the inputs, i.e. the sum over all inputs of the product of the input and the corresponding weight, exceeds a threshold, the neuron fires, otherwise it doesn't. Its operation can be expressed as an affine (upper expression) or a linear (lower expression) combination of inputs.

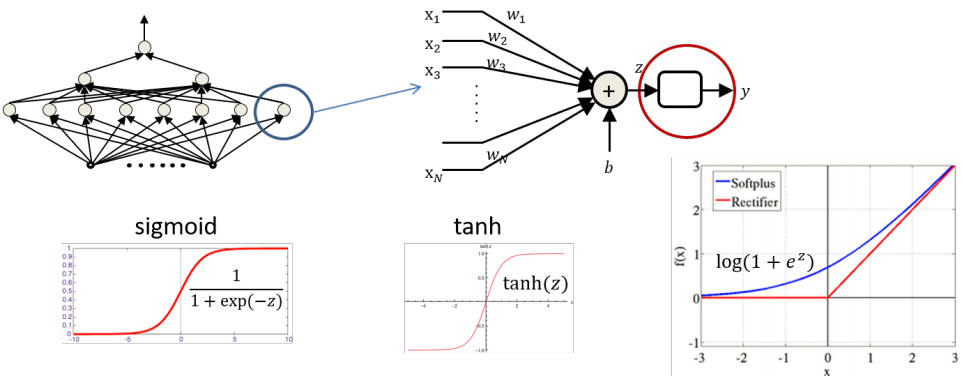


(b) The perceptron with a soft thresholding function, highlighted in yellow (lower expression).

Figure 2.1: Recap: Threshold logic in a perceptron

into a finite range of output values. Activation function need not even be squashing. The activation function shown in Fig. 2.2, which is generally referred to as a "RELU" (for "REctified Linear Unit"), is one such non-squashing function – it is a rectification function that simply truncates negative values to 0, but passes positive values through unchanged. The $softplus$ function is its smooth variant – it is differentiable everywhere. These are just a few examples of continuous activation functions. We will discuss more of these later.

For the purpose of this chapter, let us continue to assume the threshold activation, as it is somewhat easier to understand and intuitions developed from it generalize



The **sigmoid**, **tanh**, **rectifier** and **softplus** activation functions.

Figure 2.2: Examples of some activation functions

well.

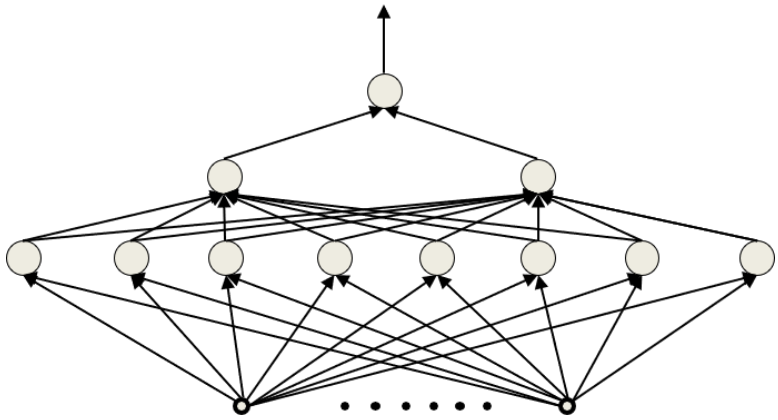
2.2 Deep structures and the concept of depth

Fig. 2.3(a) shows a general "layered" network of neurons. Fig. 2.3(b) shows a "deep" neural network.

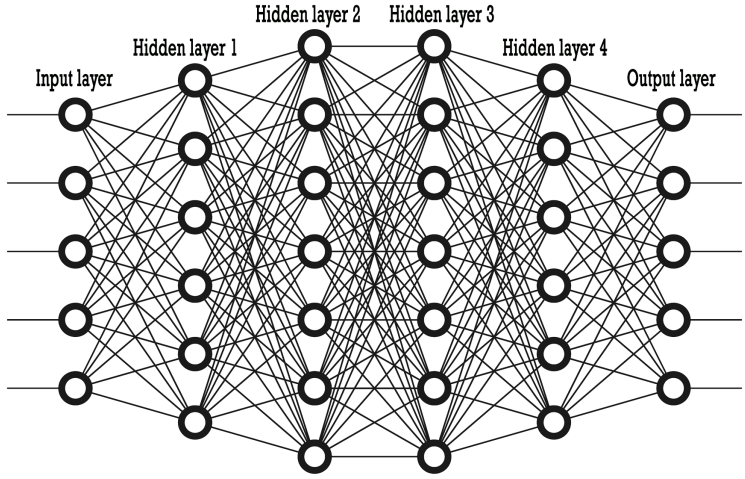
In the networks of perceptrons that we have illustrated thus far, the perceptrons are generally arranged in layers, as shown in Fig. 2.3. When MLPs have many layers, they are said to be *deep* networks. In order to quantify this, however, we must first define what it means for the perceptrons to form a layer, and the formal notion of depth in this context – what are the formal criteria for a network to qualify as a "deep" network. We discuss these below.

2.2.1 The formal notion of depth

In any directed graph with a source and a sink, the *depth* of the graph is simply the length of the path – i.e., the number of edges – on the longest path from source to sink. Fig. 2.4 shows some illustrative examples. In general, in any



(a) A multi-layered network of perceptrons.



(b) Deep layering in a network of perceptrons – a **deep** neural network.

Figure 2.3: The concept of layering and deep layering in a network of perceptrons

directed network of computational elements with input source nodes and output sink nodes,"depth" is the length of the longest path from a source to a sink.

We apply the same criterion to defining depth in neural networks. In a neural network, the computation is generally carried out in a directed fashion – from input to output. The depth of a network is the length of the longest path, i.e. the maximum number of units (or neurons[1]) from input to output.

[1]We often refer to perceptrons as neurons in this book, to be more generic

Left: There are 2 paths from source to sink. One is of length 2, another is of length 1. The depth of this graph is therefore 2. **Right:** This graph has paths of length 1, 2 and 3 from source to sink. The longest path is of length 3, so the depth of this graph is 3.

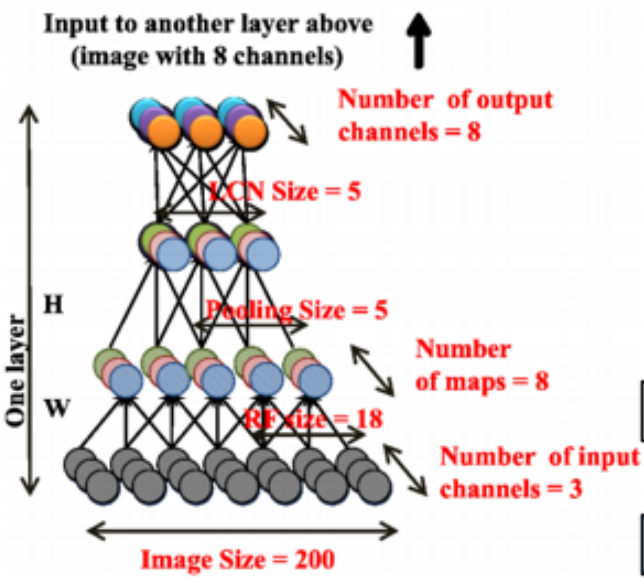
Figure 2.4: Explaining the notion of depth in deep structures

As a more specific example, consider the networks shown in Fig. 2.5. The depth of the network in Fig. 2.5(a) is 3, since the longest path from the input to output is 3. Similarly, the depth of the network in Fig. 2.5(b) is 2, and that in Fig. 2.5(c) is 4.

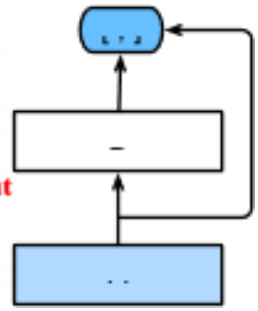
What is a "layer": The notion of a "layer" can now be formalized in terms of depth as defined above: *A layer comprises the set of neurons in which all members have same depth from the input*. For example, in Fig. 2.5(d), all the green neurons have a depth 2, so they form a layer, which we will call the second layer. Note that there is also a direct path of length 1 from the input to these neurons, but they are not part of the first layer. The first layer is the set of neurons colored red, which are all at depth 1 with respect to the input. The yellow neurons form layer 3, while the output neurons, in blue, are layer 4.

2.2.2 The multi-layer perceptron

In chapter 1, we presented a *multi-layer perceptron* as a network of perceptrons (or neurons). The *nodes* in such a network are individual perceptrons. The *edges* represent the connection between perceptrons, and each edge carries a weight. The network performs computations on some input, and produces an output. The inputs can be real valued, or Boolean stimuli. The outputs too can be Boolean or real valued. While we have largely discussed networks with a single output so far, a network can in fact have multiple outputs, or a *vector of outputs*, for any

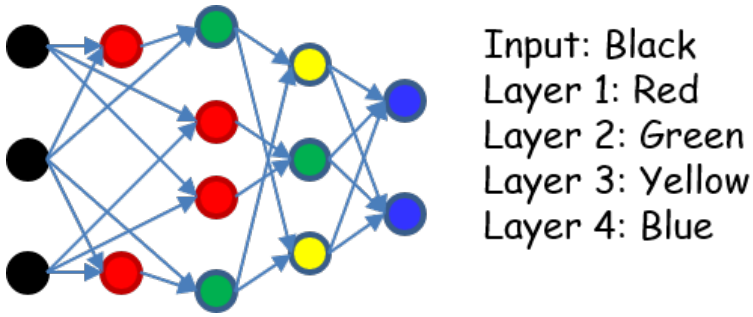


(a) A network of depth 3.



(b) A network of depth 2.

(c) A network of depth 3.



(d) A network of depth 3.

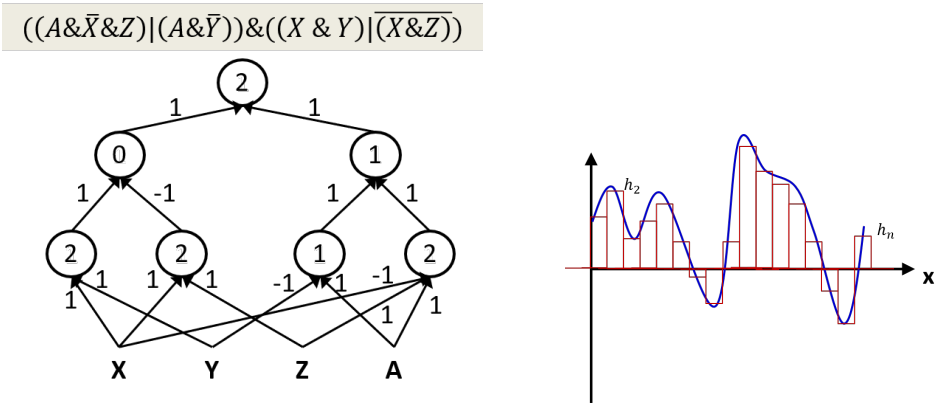
Figure 2.5: Explaining the notion of depth in MLPs

input. The perceptron network (or neural network) thus represents a function that embodies an input-output relationship between its inputs and outputs.

To understand the strengths and limitations of such networks much better, we must have an clear understanding of the types of input/output relationships that such networks can model, and the kinds of functions they can compute. The following sections attempt to build up our understanding of these aspects of MLPs.

2.3 MLPs as approximate functions

In Chapter 1, we have already seen that the MLP can compose, Boolean functions (operating on Boolean inputs and producing Boolean outputs), real-input category functions (that operate on real inputs and output Boolean values), and even real functions (which operate on real inputs and produce real outputs).



(a) A Boolean function modeled by a network.

(b) A continuous-valued function modeled by a network.

Figure 2.6: Examples of Boolean and continuous valued functions modeled by an MLP

Some examples are shown in Fig. 2.6. Fig. 2.6(a) shows a network that models an ugly function of four Boolean variables. Fig. 2.6(b) graphs a function (in blue) and the piece-wise approximation for it that can be obtained from an appropriately constructed network (See Section 1.4.10 for explanation).

An important question in this context is: can we *always* construct a network to compute any function, or are there limitations on the kinds of functions that can be modeled with a network (and if so, what are they)? We will learn the answers to these question in the sections that follow.

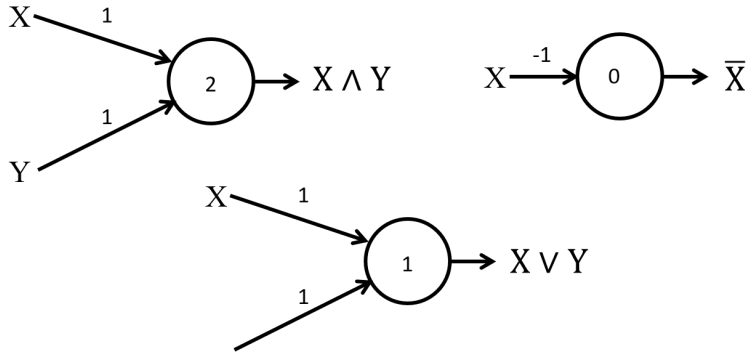
2.4 MLPs as universal Boolean functions

Multi-layer perceptrons can model any Boolean function: given a Boolean function, we can construct an MLP that computes it. The function can be arbitarily complex, and over any number of variables. Does this imply that the MLP that models it must also be complex?

There are several ways of quantifying the complexity of a network. We will consider two: the number of *layers* in the network, and the number of *neurons* in the network.

2.4.1 The perceptron as a Boolean gate

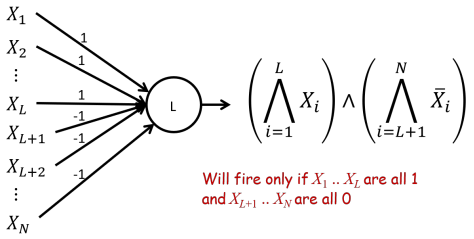
We have seen that individual perceptrons can model Boolean gates. Fig. 2.7 shows examples of how a perceptron can model AND, NOT and OR gates.



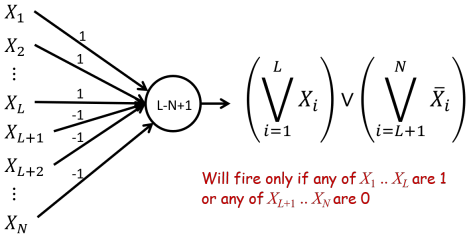
The numbers on the edges represent weights, the numbers in the circles represent thresholds. The perceptron outputs a 1 if the total weighted input equals to, or exceeds the threshold. **Top left:** a perceptron that only outputs a 1 when both inputs are 1. This models an AND gate. **Top right:** This perceptron outputs a 1 when the input is 0 and vice versa. This models a NOT gate. **Bottom:** This perceptron outputs a 1 when either of the inputs is 1. This models an OR gate.

Figure 2.7: AND, NOT and OR gates modeled by a perceptron

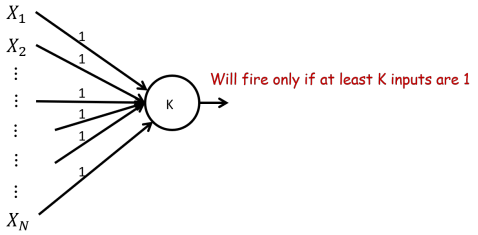
Perceptrons can however model much more. A perceptron can model *generalized*



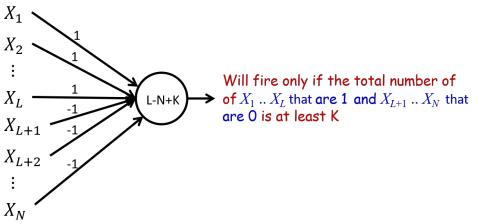
(a) A generalized AND gate. This perceptron will only fire if X_1 through X_L are 1 and X_{L+1} through X_N are 0.



(b) A generalized OR gate. This perceptron will only fire if any of the first L inputs is 1 OR if any of the remaining inputs is 0.



(c) A generalized majority gate. This perceptron will fire if at least K inputs are of the desired polarity.



(d) Another generalized majority gate. This perceptron will fire *only if* the total number of $X_1, X_2, \ldots, X_L$ that are 1 and $X_{L+1}, X_{L+2}, \ldots, X_N$ that are 0, is at least K.

Figure 2.8: Generalized gates modeled by a perceptron

gates. This is illustrated by the panels in Fig. 2.8. In Fig. 2.8(a), the perceptron is a generalized AND gate. It will fire only if $X_1, X_2, \ldots, X_L$ are all 1, and $X_{L+1}, X_{L+2}, \ldots, X_N$ are all 0.

To understand why this is so, consider the case when $X_1, X_2, \ldots, X_L$ are all 1 and the rest are 0. Then the total input is exactly L, and since the threshold is also L, the perceptron will fire. But now if we set any of the inputs from $X_{L+1}, X_{L+2}, \ldots, X_N$ to 1, then because those inputs have a weight of -1, the sum of all weighted inputs will become less than L, and the perceptron will not fire. So also, if any of the first L bits becomes 0, the sum can never be L or more, and the unit will not fire. Thus, this perceptron *only* produces an output of 1 when $X_1, X_2, \ldots, X_L$ are 1 and the rest are 0. In general, we can construct a gate that will only fire under the very specific condition that some chosen subset of inputs

are 1, and the rest are 0, by choosing an appropriate threshold and appropriate weights to apply to the inputs.

In Fig. 2.8(b), the perceptron is generalized OR gate. It will fire if any of the first L inputs is 1, OR if any of the remaining inputs is 0. To understand this, let us assume that all of the first L inputs are 0 and all of the remaining inputs are 1 – the only condition under which it should *not* fire.

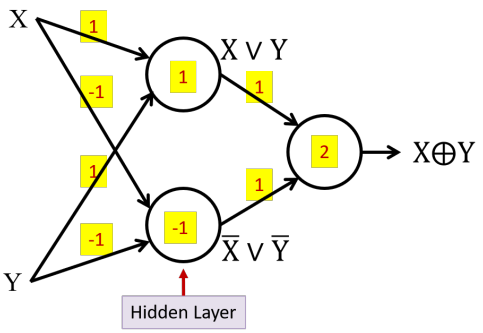
In this situation, there are $N - L$ inputs with value 1, and all *their* weights are -1. Therefore, the total input is $L - N$. Now, if even *one* of the inputs flips its value, the sum becomes $L - N + 1$, which is the threshold at which the perceptron fires. In other words, if any of the first L inputs becomes 1, or if any of the last $N - L$ inputs becomes 0, the perceptron will fire, which is the required OR condition in this gate. If more of the inputs satisfy our condition, the sum will, of course, exceed the threshold of $L - N + 1$.

In Fig. 2.8(c), the perceptron is generalized *majority* gate. The perceptron shown will fire if *at least* K of the inputs are 1.

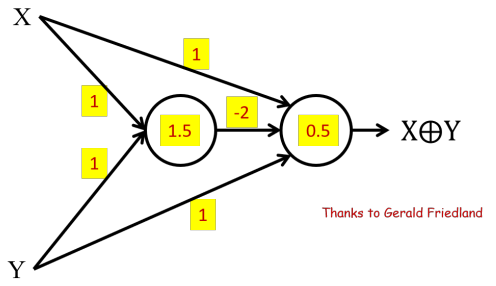
This is so because in the first case, all inputs have a weight of 1, and if at least K of the inputs are 1, the total weighted sum of inputs will equal or exceed the threshold K. The perceptron will then fire. This gate is called a generalized majority gate because for $K > N/2$ the perceptron fires if the majority of inputs is 1.

Similarly, the perceptron in Fig. 2.8(d) is another example of a generalized majority gate. It will fire only if the total number of $X_1, X_2, \ldots, X_L$ that are 1 and $X_{L+1}, X_{L+2}, \ldots, X_N$ that are 0, is at least K.

Thus we see that the perceptron is a very versatile unit, and can model complex Boolean functions. It is probable that such examples and reasoning led Rosenblatt to believe that perceptrons could model all Boolean functions, without exception. However, this is not the case. In spite of such versatility, the perceptron cannot model the XOR function. There is in fact no combination of weights and threshold for which it will compute an XOR.



(a) Example of a perceptron network that computes the XOR.



(b) A more compact MLP that can compute an XOR.

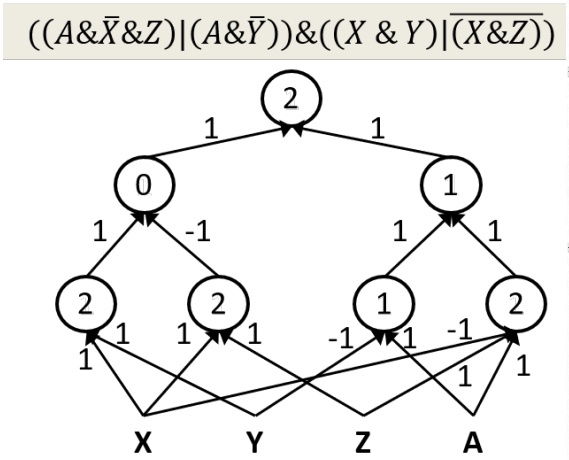
Figure 2.9: Examples of MLPs modeling XOR functions

In fact, to compute an XOR we *must* use a network of perceptrons, or a multilayer perceptron as we discussed in Chapter 1. For example, Fig. 2.9(a) shows a network that computes the XOR. It is a network of 3 perceptrons, with one hidden layer whose outputs we do not directly see. It has one output perceptron. The computation performed by this structure is rather obvious: the first perceptron (on top), in the hidden layer computes $(X \text{ OR } Y)$. The perceptron at the bottom in the hidden layer computes $((\text{NOT } X) \text{ OR } (\text{NOT } Y))$. The final (output) perceptron ORs the outputs of the two hidden-layer perceptrons to yield $(X \text{ XOR } Y)$.

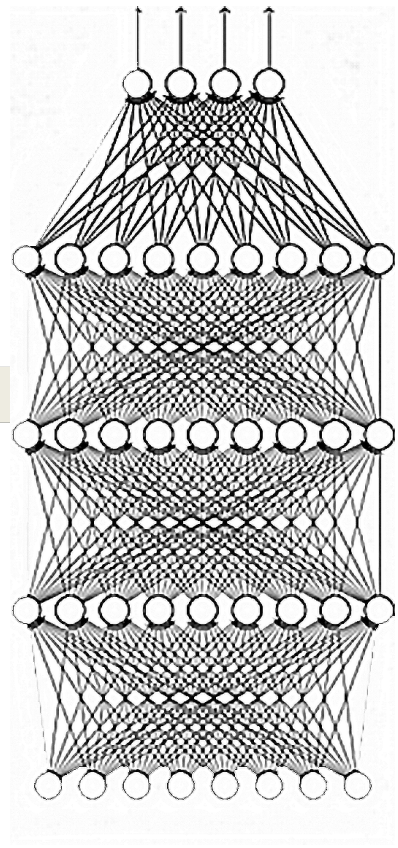
This is of course not the only way to build an MLP to compute an XOR. Fig. 2.9(b) shows a more compact way of building an MLP to compute an XOR. This uses only two perceptrons, uses 5 weights and 2 thesholds, and now the weights are also not all 1.

Since MLPs can compose any Boolean gate, and any network of Boolean gates, MLPs are, in fact, universal Boolean functions. For any Boolean function there is always an MLP that can compute it.

Although MLPs are universal Boolean functions, it is important to know how complex the network must be in order to compute any given Boolean function. In the example shown in Fig. 2.10(a), we see that in order to construct the function shown, we must use 7 neurons in three layers. If we had a more complex



(a) A multi-layered MLP can model any function, such as the one shown on top. An MLP that computes this function has two layers.



(b) A deep network. The depth of a network is relevant to its ability to compute a given Boolean function.

Figure 2.10: MLPs as universal Boolean functions

function, of a larger number of inputs, how can we determine the number of layers that an MLP would need to compute it?

To understand this, let us consider the example shown in Fig. 2.11. Any Boolean function can be expressed as a *truth table*, that explicitly lists every possible combination of input variables and the corresponding outputs as a table. In the example shown in Fig. 2.11, columns in blue represent all combinations of input variables, and the yellow column on the right hand side lists the corresponding output for each row. Note that we need not enter *every* combination of inputs in the table. It is sufficient to only list all the input combinations for which the

output is 1. For the other combinations, the output is implicitly 0.

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$

Any Boolean function can be represented as a truth table. The columns in blue represent all combinations of input variables, and the yellow column on the right hand side lists the corresponding output for each row. The function that this truth table represents is shown below the table.

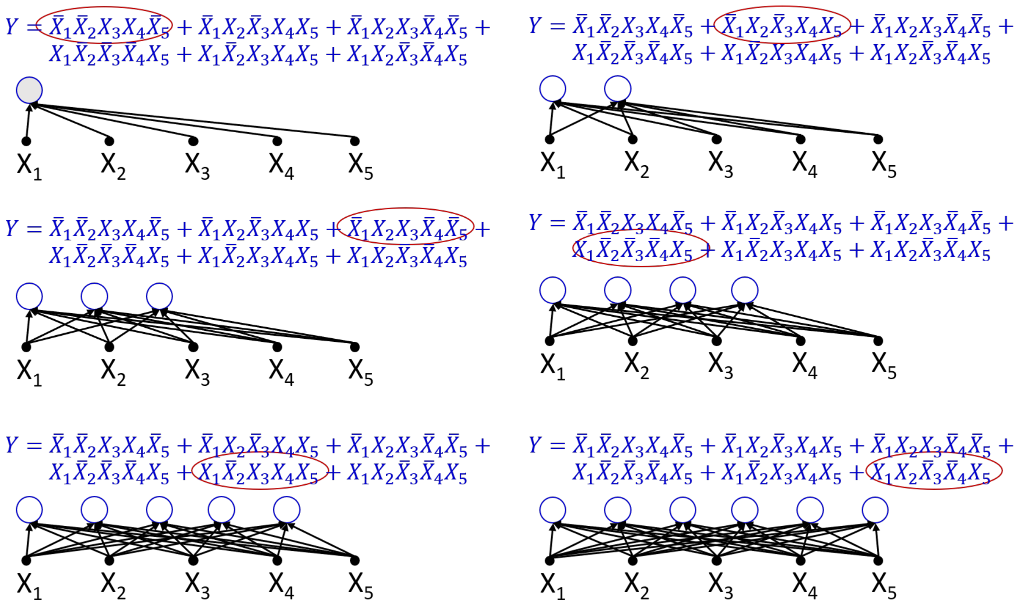
Figure 2.11: Boolean function as a truth table and its disjunctive normal form

The function that the truth table in Fig. 2.11 represents is in fact expressed below it in a *disjunctive normal form* (DNF form or DNF formula).

The DNF formula has *clauses*: each clause represents one of the combinations for which the output is 1, i.e. one of the rows of the truth table. Thus the first clause in the DNF form shown in Fig. 2.11 – (NOT X_1) AND X_2 AND X_3 AND X_4 AND (NOT X_5) – represents the first row in the truth table. Note that by definition, (NOT X_1) becomes true when X_1 is 0. Thus the first clause indicates that the output is 1 when X_1 is 0, X_2 is 0, X_3 is 1 , X_4 is 1 and X_5 is 0. The second clause similarly represents the second row in the truth table: (NOT X_1) AND X_2 AND (NOT X_3) AND X_4 AND X_5. The third clause represents the third row: (NOT X_1) AND X_2 AND X_3 AND (NOT X_4) AND (NOT X_5). The fourth clause represents the fourth row: X_1 AND (NOT X_2) AND (NOT X_3) AND (NOT X_4) and X_5, and so on. Since the truth table lists only six combinations

for which the output is 1, the DNF formula shown has exactly six clauses, one representing each of the combinations in the truth table, and its interpretation is that the function that it represents takes a value of 1 if any of the clauses is true.

Once we have this representation, we can use it to build an MLP to compute it, using perceptrons that model generalized AND gates and OR gates. The perceptrons that model the first to sixth clauses are shown in Fig. 2.12.



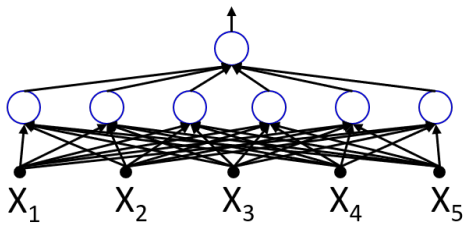
The networks for computing successive terms in the disjunctive normal form.

Figure 2.12: Construction of MLPs corresonding to a DNF/truth table

Finally, as shown in Fig. 2.13, the MLP has one output perceptron that ORs the outputs of the six clauses. Thus we now have an MLP that computes exactly the Boolean formula shown in the example of Fig. 2.11. An interesting point to note is that this network only has one hidden layer, wherein each neuron computes one of the clauses in the DNF formula for the function. In fact, since any Boolean function can be expressed as a truth table, and thereby in DNF form, it can be modeled by an MLP with only one hidden layer. *A one-hidden-layer MLP is a universal Boolean function*!

This leads us to a related question: while any Boolean function can be modeled

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



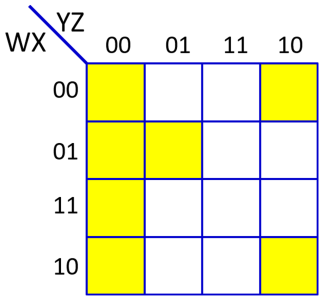
The final network for computing the complete disjunctive normal form.

Figure 2.13: The complete MLP for the DNF/truth table

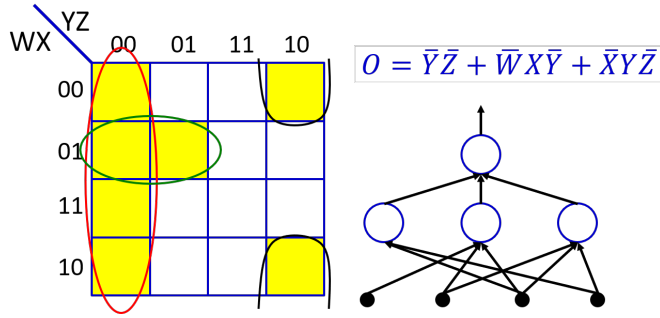
an MLP with just one hidden layer, how large must that layer be, i.e. how many neurons must it contain? Since the one-hidden-layer MLP is derived from the DNF formula for the Boolean function, the equivalent question is – what is the smallest DNF formula (i.e. the DNF formula with the fewest clauses) that can model the function? To determine this, we must reduce the Boolean function to its minimal form. We discuss this below.

2.4.2 Reducing the Boolean function

Fig. 2.14 shows an alternative representation of a truth table. This representation is called a *Karnaugh map*. A Karnaugh map is a way of representing bit patterns such that bit patterns that differ by only one bit are always arranged next to one another. The example in Fig. 2.14(a) is a Karnaugh map over four variables: W, X, Y and Z. The four rows represent the four combinations of W and X. Observe that adjacent rows differ by only one bit. The four columns represent the four combinations of Y and Z. Observe again that adjacent columns differ by only one bit. Thus all combinations of the four variables can be placed on a 4×4 grid. Any grid cell differs from its neighbors by only one bit. Also, the rightmost edge is "connected" to the leftmost edge because patterns in the rightmost column and leftmost column differ by only one bit. For instance the rightmost box in the first row, representing 0010 differs by only one bit from the



(a) A Karnaugh map. It represents a truth table as a grid. Filled boxes represent input combinations for which output is 1; blank boxes have an output of 0. Adjacent boxes can be "grouped" to reduce the complexity of the DNF formula.



(b) Reducing the DNF by grouping filled boxes (3 groups or clauses emerge in this example). The single-hidden layer MLP that computes the Boolean function for this truth table has only 3 neurons in the hidden layer, each of which computes one of these clauses. The black dots in the input layer represent W, X, Y and Z from left to right.

Figure 2.14: The Karnaugh map and its role in reducing DNFs to minimal forms

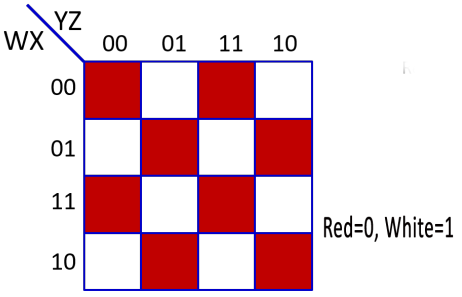
leftmost box in this row, which represents 0000. Thus they can be considered to be "connected" – i.e. are neighbors, and the row represents a "cycle." The top row is connected to the bottom row in the same manner, since 0010 differs from 1010 by only one bit. The square grid in Fig. 2.14(a) is in fact a *torus*, and represents a topologically correct representation of all patterns of the 4 variables used.

The Karnaugh map in the example shown takes the place of a 4-variable truth table: the 16 boxes represent the 16 possible combinations of the 4 input variables. The boxes highlighted in yellow represent input combinations for which the output is 1. For the function represented by this Karnaugh map, there are 7 input combinations that produce the output 1. A naïve DNF would thus have 7 clauses.

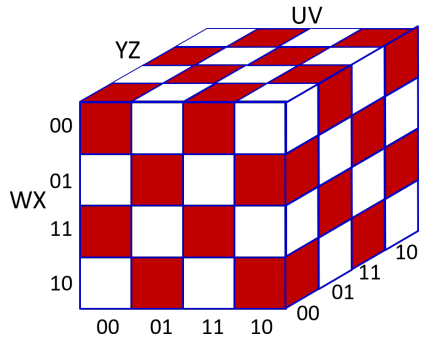
To reduce the size of that DNF, the neighboring boxes that are filled can be grouped. This is shown in Fig. 2.14(b). In this, the four boxes in the first column all produce the output 1. If the combination YZ takes the value 00, it doesn't

matter what WX is, the output is always 1. They can therefore be grouped into the DNF clause $(\text{NOT}\, Y)$ AND $(\text{NOT}\, Z)$. Similarly, the two neighboring second row elements shown in the figure too produce the output 1, which means for $WX = 01$, if $Y = 0$, the output is 1 and Z can be ignored. This gives us the second group which comprises the DNF caluse $(\text{NOT}\, W)$ AND X AND $(\text{NOT}\, Y)$. In the fourth column, the two filled boxes are actually neighbors, as explained earlier, and can be grouped into the clause $(\text{NOT}\, X)$ AND Y AND $(\text{NOT}\, Z)$. As a matter of fact, we could group all four corners together and get an even simpler clause $(\text{NOT}\, X)$ AND $(\text{NOT}\, Z)$.

With this grouping, the original 7-entry truth table can now be reduced to only 3 groups, with a corresponding 3-clause DNF. The reduced DNF formula is shown in Fig. 2.14(b). The figure also shows the reduced perceptron network for this function, which needs only 3 hidden units. Thus, reduction of the DNF reduces the size of the one-hidden-layer network needed to model the corresponding Boolean function. In other words, so long as the entries in the truth table can be grouped based on neighborhood, the DNF formula that expresses it can be made smaller, and the resulting network too will be small. When the DNF formula is minimal, the network that represents it will be the smallest one required to compute the corresponding Boolean function.



(a) A different Karnaugh map. This map cannot be reduced further.



(b) A Karnaugh map in 3 dimensions. This map also cannot be reduced further.

Figure 2.15: Examples of irreducible Karnaugh maps

In this context, it is interesting to note the examples of Karnaugh maps given in Fig. 2.15. These give us an idea of how large the one-layer Boolean MLP can get,

in the worst case. Why is this so? Smaller Boolean MLPs are possible through the process we have just discussed because we can group adjacent highlighted cells in a Karnaugh map. However, this may not always be possible. With a little thought, it is easy to see that the one arrangement of cells in a Karnaugh map in which cells simply cannot be grouped together is the checkerboard pattern, as shown in Fig. 2.15. In the examples shown, alternate entries are highlighted in red in the rows and columns, representing an output of 1 for the corresponding input patterns (the input variables are indicated in each example). The pattern ensures that no two cells can be grouped. These also represent the largest irreducible DNFs. The DNF corresponding to Fig. 2.15(a) would have half as many clauses as the product of the number of cells in the map ($16/2 = 8$ in this example). The corresponding MLP would have 8 neurons in its hidden layer[2].

[2]We are assuming, for purposes of illustration, that the perceptrons are used only to model simple Boolean gates.

Fun facts 4.1: Karnaugh maps or K-maps

Boolean functions can be exrpressed in two ways: in their disjunctive normal form (DNF) (more explicitly canonical disjunctive normal form (CDNF)) and in their conjunctive normal form (CNF) (more explicitly canonical conjunctive normal form (CCNF)).

The CDNF is also called the *minterm* canonical form and the CCNF is called the *maxterm* canonical form. Other canonical forms exist, such as the algebraic normal form, Blake canonical form etc which we will not discuss.

A minterm expression uses logical ANDs – products of variables. A maxterm expression uses logical ORs – sums of variables. These are dual concepts. They are have a complementary (or symmetrical) relationship with each other (they follow De Morgan's laws).

The *Sum of Products* (SoP or SOP) is a sum of minterms: a disjunction (OR) of minterms. The *Product of Sums* (PoS or POS) is a product of maxterms: a conjunction (AND) of maxterms.

Karnaugh maps [1], or K-maps, are used to simplify or minimize complex logic expressions. They are useful in many applications that implement Boolean logic (among others): in circuits that use logic gates, software design etc.

K-maps can use minterm or maxterm expressions. Accordingly, they have two forms: SoP and PoS forms. In the examples above, we have used the SoP form, which is practically realized in hardware implementations using AND gates in the minterms, which are aggregated (summed) by an OR gate. A PoS form would be its alternative, and would use OR gates in the maxterms, which would all be aggregated by an AND gate. In MLPs, the perceptrons used would do the exact equivalent of these operations.

Fun facts 4.1 continued.....

How do we make a K-map for a large (even or odd) number of variables? It is not easy. The problem arises due to neighborhood: two bit patterns are neighbors if they differ in no more than one bit. So, for instance, 0011, 1010, 0110 and 0000 are all immediate neighbors of 0010. To properly display this neighborhood pattern we will need a 2D map, since a linear map will allow no more than 2 neighbors for any cell. But now when we go up to 5 or 6 variables, each bit pattern will have up to six neighbors. This will require a 3-dimensional grid, with the caveat that the grid is not cuboidal, but in fact a hard-to-visualize four-dimensional torus, since the left and right faces of the cube are neighbors, as are the front and back, and top and bottom. In general, with N variables, the K-map equivalent may be viewed either as a hypercube in $\lceil N/2 \rceil$-dimensional space with opposite faces being neighbors, or a hypertorus in even higher dimensional space.

Popular visualizations will instead show "unfolded" versions of the map, which lay "slices" of the hypercube next to one another in a manner that captures the neighborhood along 4 of the bits, leaving higher-dimensional neighborhood relations to the imagination of the viewer. For example, such a visualization of a K-map for 8 variables is shown below:

Fun facts 4.1: continued further.....

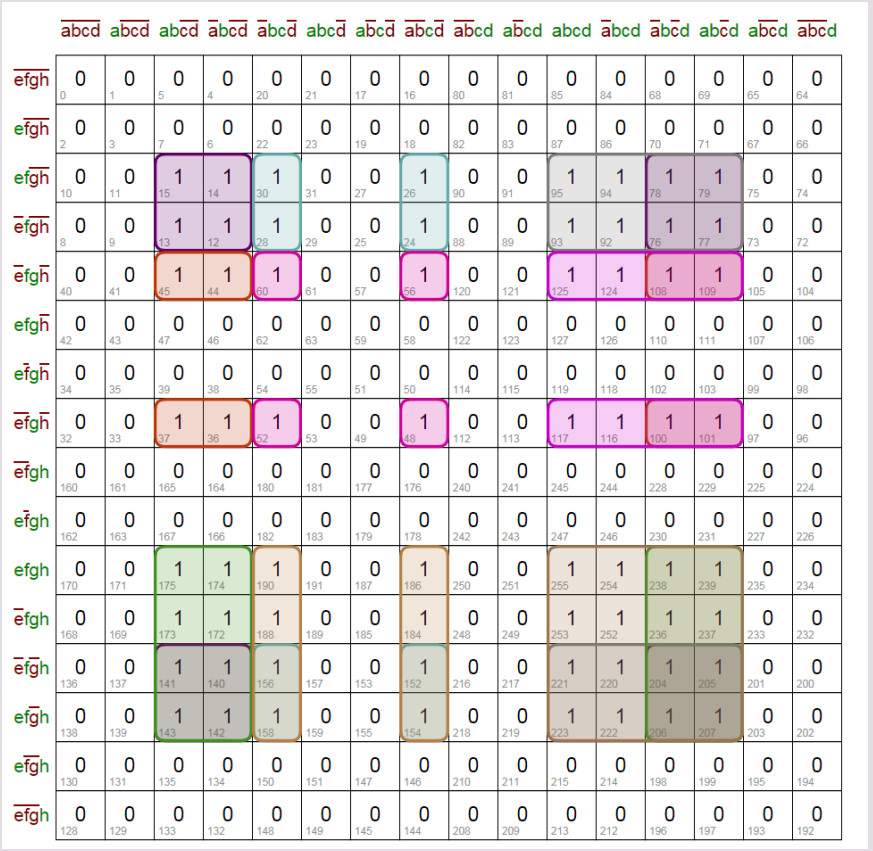


Figure 2.16: An 8-variable Karnaugh map (Source: math.stackexchange.com)

2.4.3 Width of a one-hidden-layer Boolean MLP

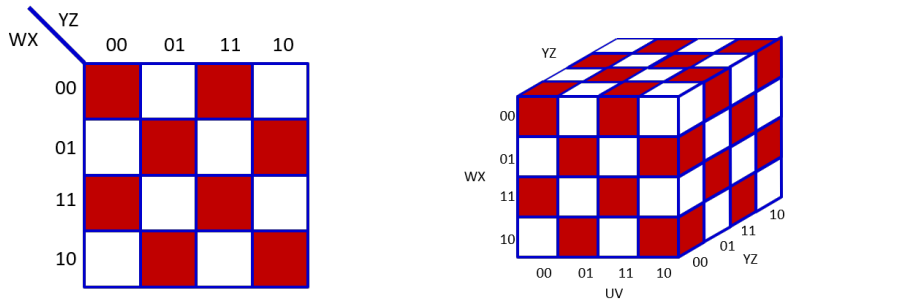
Fig. 2.15(b) shows a 3-dimensional version of the Karnaugh map of six input variables: U, V, W, X, Y and Z. In this case, 32 neurons would be required in the hidden layer for a one-hidden-layer MLP that computes the corresponding Boolean function.

This reasoning can be generalized. If we have a Boolean function over N variables, and construct a one-hidden-layer MLP to compose the corresponding DNF formula, we will need a maximum of 2^{N-1} neurons in the hidden layer. The size of the network (in terms of the neurons required) is thus exponential in N, the number of inputs.

However, this applies only to a one-hidden-layer MLP. An obvious question in this context is: what would the size of the network be if we allowed multiple hidden layers? We discuss this next.

2.4.4 Size of a deep MLP

The function represented by each checkerboard patterned Karnaugh map in the example above is actually just an XOR, as shown in Fig. 2.17. The four-variable function corresponding to the map on the left is $W \oplus X \oplus Y \oplus Z$, where the symbol "$\oplus$" represents the XOR operation. Similarly, the six-variable function corresponding to the map on the right is $U \oplus V \oplus W \oplus X \oplus Y \oplus Z$.



The Karnaugh maps shown represent XOR functions. **Left:** $W \oplus X \oplus Y \oplus Z$, **Right:** $U \oplus V \oplus W \oplus X \oplus Y \oplus Z$.

Figure 2.17: Karnaugh maps representing XOR functions

Recall that we can compose a single XOR function using an MLP with 3 perceptrons, or in fact even only 2 perceptrons, as shown in Fig. 2.9. Fig. 2.18(a) recalls the 3-perceptron case.

Now let us consider Fig. 2.18(b). For the Karnaugh map shown in Fig. 2.18(b), the corresponding function is shown below it. For this function, we can construct the MLP as a series of XOR MLPs as shown on the right of the Karnaugh map. In this MLP, the first 3 neurons compute $W \oplus Y$. The next 3 neurons compute the XOR of the result of this with Y, and the final three neurons XOR this output with Z. Since the function shown includes only 3 XORs and we need 3 neurons per XOR, we need only 9 neurons when we build a deep MLP for it. Recall that the one-hidden-layer MLP for this, shown in the previous example, also needed 9 neurons – one each for each of the 8 clauses, and one final OR neuron. The current XOR scheme does not present any advantage in this case. However, as we shall see in the subsequent example, when we have a larger number of inputs to contend with, the enormous difference and advantage of using a deep structure becomes apparent.

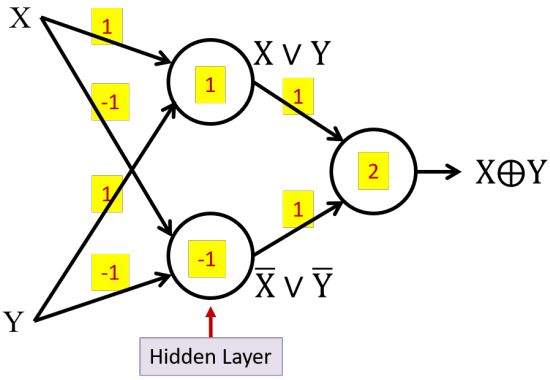
Now consider Fig. 2.19. The Karnaugh map shown in this figure represents a function over 6 variables, and has 5 XORs in the corresponding function, as shown. In this case, the deep MLP would only require 15 neurons, whereas the one-hidden-layer network required 65 neurons, as explained earlier.

This requirement can be generalized for N input variables. The XOR of N input variables will require $3(N-1)$, or even only $2(N-1)$ neurons (if we use the two-neuron model for the XOR) when built in this manner.

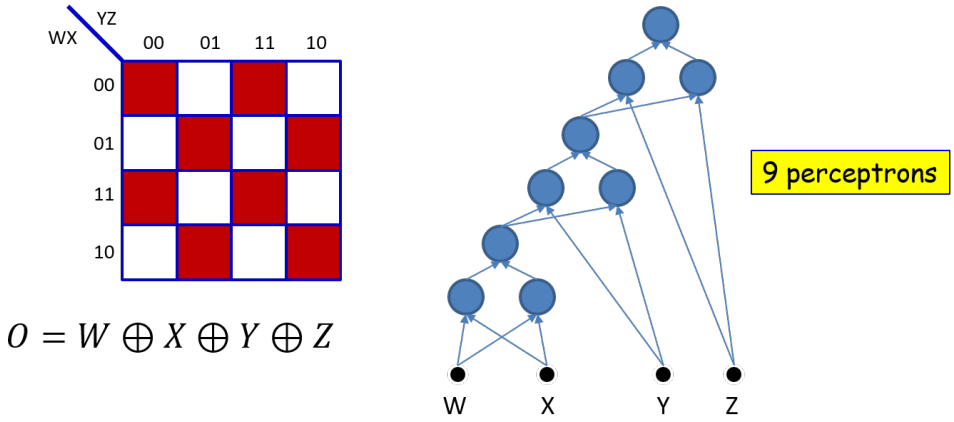
To summarize the differences between a one-hidden-layer Boolean MLP and a deep Boolean MLP, we see that more generally, when we have an XOR of N inputs, a naïve single-hidden-layer MLP will require $2^{N-1}+1$ neurons, which is exponential in N, whereas a deep network will require only $3(N-1)$ perceptrons, which is linear in N. In addition, the $3(N-1)$ neurons can be arranged in only $2log_2(N)$ layers. This is explained through the example in Fig. 2.20[3].

Let us first focus on Fig. 2.20(a). We note that the XOR is associative. This

[3]Once again, we are assuming that perceptrons are used as Boolean gates in this explanation. Smaller single-hidden-layer XOR circuits with $O(N)$ neurons can be constructed using them as threshold gates. Even in that case, the number of *connections*, representing parameters, required by the single hidden layer circuit is $O(N^2)$, whereas the deeper circuit requires $O(N)$ connections.



(a) Recap: XOR using an MLP with 3 perceptrons.

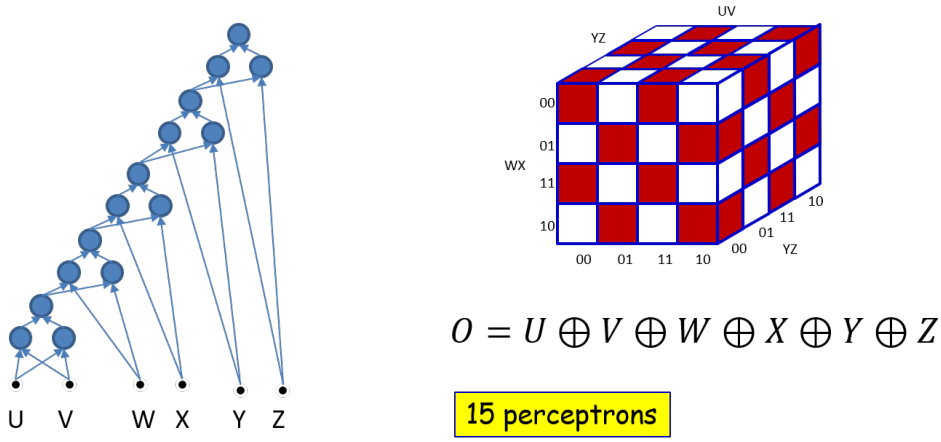


(b) An XOR needs 3 perceptrons. This network will require $3 \times 3 = 9$ perceptrons.

Figure 2.18: Composing an MLP from Karnaugh maps with XORs

means that we can group the operations in any way, and the output will still be the same. So, as shown in Fig. 2.20(a), we can first compute the XORs of pairwise groupings of the variables. That gives us $N/2$ XOR'ed outputs. We then XOR the outputs, also in a pairwise fashion, and continue to do this until we have a single output. At each stage, this strategy reduces the number of variables by 2. There are $log_2 N$ such stages. Each stage requires 2 layers to compute the XOR. Thus we obtain a total of $2log_2 N$ layers.

In Fig. 2.20(b), each pair of layers that produces a set of variables that must be further XOR'd downstream is highlighted in a blue box. The same rules about the size of the net apply from there on.



An XOR needs 3 perceptrons. The network shown in this figure will require $3 \times 5 = 15$ perceptrons. More generally, the XOR of N variables will require $3(N-1)$ perceptrons.

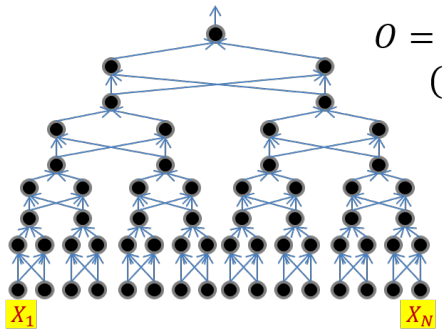
Figure 2.19: Estimating the size of deep MLPs from Karnaugh maps

2.4.5 The challenge of depth

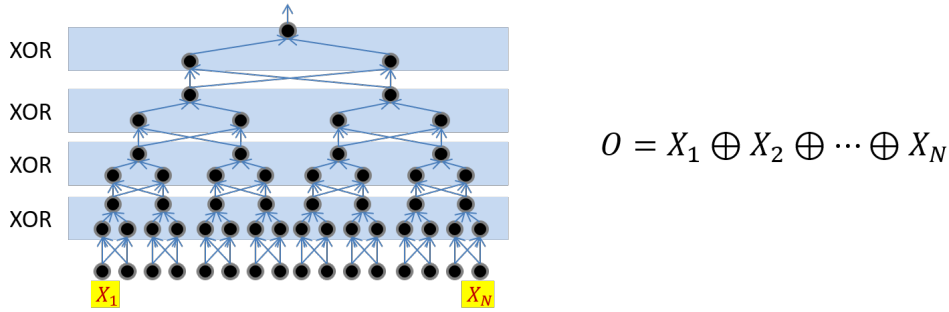
The key point to be noted here is that if we terminate the net after only a few layers, say K layers as shown in Fig. 2.21, we must still compute an XOR over $2^{-K/2}N$ variables, which will require an exponentially-sized network. In general, as we reduce the size of the net below the minimum depth ($O(\log_2(N))$) for an XOR-based network, the number of neurons required to compute the function grows rapidly. For any fixed depth network, the number of neurons that are required will still grow exponentially with the number of inputs. Also, very importantly, if we use fewer than the minimum required neurons, we will not be able to model the function at all! In fact, for an XOR, if we use a total of even one neuron below the minimum requirement, the error incurred in computing the function will be 50%.

2.4.6 The actual number of parameters in the network

Fig. 2.22(a) illustrates another important point: it is not really the number of neurons in the network that governs network size, but the number of *connec-*

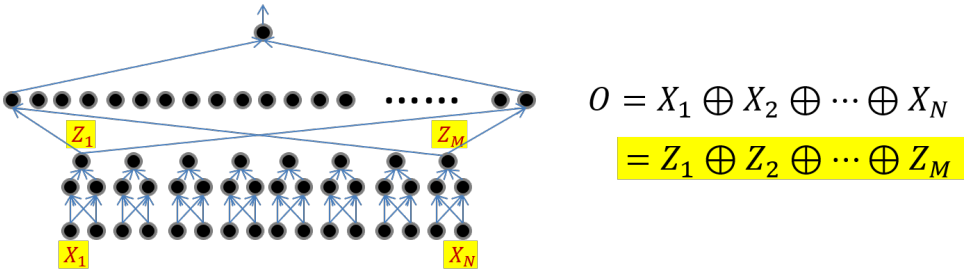


(a) Only $2\log_2 N$ layers are needed and can be obtained by pairing terms: 2 layers per XOR. The final output is given by O.



(b) Each pair of layers that produces a set of variables that must be further XOR'd downstream is highlighted in a blue box.

Figure 2.20: How $3(N-1)$ neurons can be arranged in only $2log_2(N)$ layers.



Using only K hidden layers will require $O(2^{CN})$ neurons in the K^{th} layer, where $C = 2^{(-(K-1)/2)}$, since the output can be shown to be the XOR of all the outputs of the $(K-1)^{th}$ hidden layer. In other words, reducing the number of layers below the minimum will result in requiring an exponentially-sized network to express the function fully. A network with fewer than the minimum required number of neurons cannot model the function.

Figure 2.21: Illustrating the challenge of depth

tions, each of which has an associated weight. The number of *parameters* in the network is the number of weights. In the example in Fig. 2.22(a), it is 30. However, for now we can still think in terms of the number of neurons because networks that require an exponential number of neurons require an exponential number of weights.

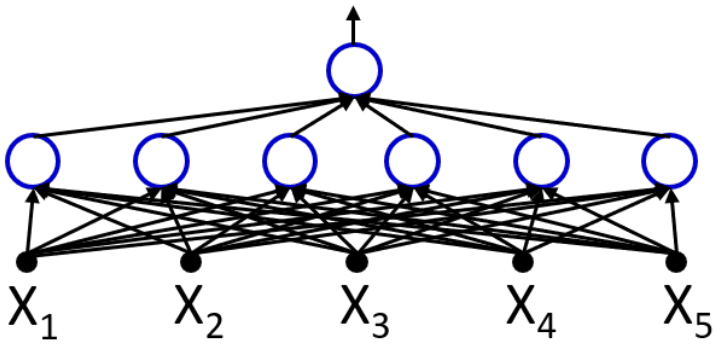
In summary, deep Boolean MLPs that scale linearly in size with the number of inputs, can become exponentially large if they are recast using only one hidden layer. The problem gets worse: if we have any function that can eventually be expressed as the XOR of a number of intermediate variables, as shown in Fig. 2.22(b), from that point on we need depth. If we have a fixed depth from that point on, the network can grow exponentially in size. Having a few extra layers can greatly reduce the size of a network.

2.4.7 Depth vs size in Boolean circuits

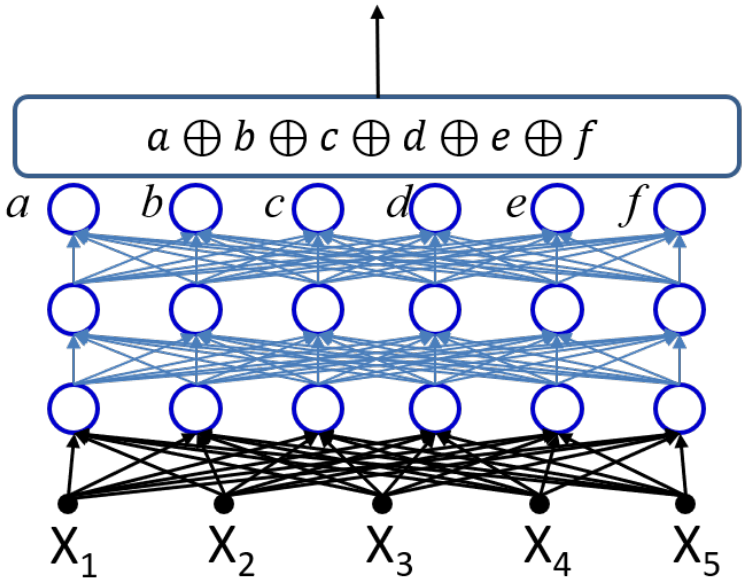
The XOR function we have discussed in the examples above is really a parity problem. It can actually be shown that any Boolean parity circuit of depth d, based on AND, OR and NOT (*i.e.* Boolean) gates, must have size $2^{n^{1/d}}$ [6]. Alternately stated, $parity \notin AC^0$ (set of constant-depth polynomial-size circuits of unbounded fan-in elements) – i.e., the parity function does not belong to the class of functions that can be implemented with fixed depth networks with only a polynomial number of Boolean units. It will require an exponential number of units or gates.

There are some caveats that must be noted in this context:

1. **Caveat 1:** *Not all Boolean functions* have such clear depth vs. size trade-off. Claude Shannon proved that for $N > 2$ there exists a Boolean function of N variables that requires at least $2^N/N$ Boolean gates. In fact he showed that for large N, almost all N-input functions require more than $2^N/N$ gates, regardless of depth. In fact, if all Boolean functions over N inputs could be computed using a circuit that is polynomial in N, then



(a) The actual number of parameters in a network is the number of connections In this example there are 30. This is the number that really matters in software or hardware implementations. Networks that require an exponential number of neurons will require an exponential number of weights.



(b) If we have a fixed depth from any point, the network can grow exponentially in size from there.

Figure 2.22: Relating the number of parameters (connections) and the network size

$P = NP$. Regardless, for working purposes we can retain the lesson that as a network gets deeper, the number of neurons required to represent a

function can fall exponentially. This is why we need depth.

2. **Caveat 2:** So far we have considered a simple "Boolean circuit" analogy for explanations. But an MLP is actually a threshold circuit, not just a Boolean circuit. It is composed of threshold gates, which are far more versatile than Boolean gates. For instance, a single threshold gate can compute the majority function, but a circuit of bounded depth, composed of Boolean gates would require an exponential number of them. Nonetheless, the general principles still hold – functions that require an extremely large (exponential) number of units if modelled by a shallow network, can require far fewer (linear) number of units if the network is permitted to be arbitrarily deep.

In summary, of this section, we see that an MLP is a universal Boolean function, but can represent a given function only if it is sufficiently wide and sufficiently deep. Depth can be traded off for (sometimes) exponential growth of the width of the network. The optimal width and depth of a network depend on the number of variables and the complexity of the Boolean function it must compute, where the term "complexity" refers to the minimal number of terms in DNF formula required to represent it.

Recap 4.1

- Multi-layer perceptrons are Universal Boolean Machines.
- Even a network with a single hidden layer is a universal Boolean machine – however, a single-layer network may require an exponentially large number of perceptrons.
- Deeper networks may require far fewer neurons than shallower networks to express the same function, and can therefore be exponentially smaller in size.

Discussion and info box 4.1:

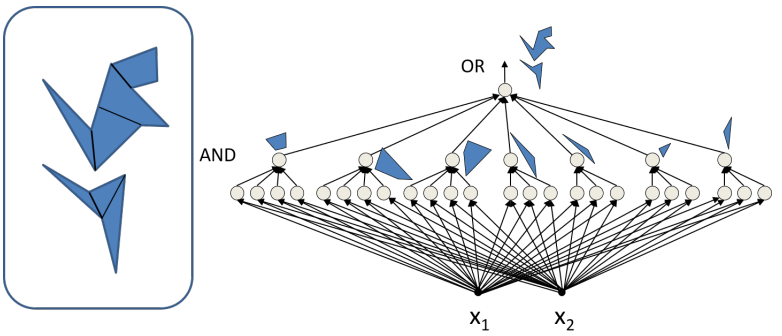
Formally, the class of Boolean circuits is a subset of the class of Threshold circuits. This means that for N inputs we can, in fact compose a depth-2 threshold circuit for parity using only $O(N^2)$ weights. But a network of depth $log(N)$ will still only require $O(N)$ weights. More generally, even for threshold circuits, for large N, for most Boolean functions, a threshold circuit that has size polynomial in N at an optimal depth d can become exponentially large at $d-1$. Other formal analyses of neural networks generally view them as *arithmetic circuits* – circuits that compute *polynomials* over any field.

2.5 MLPs as universal classifiers

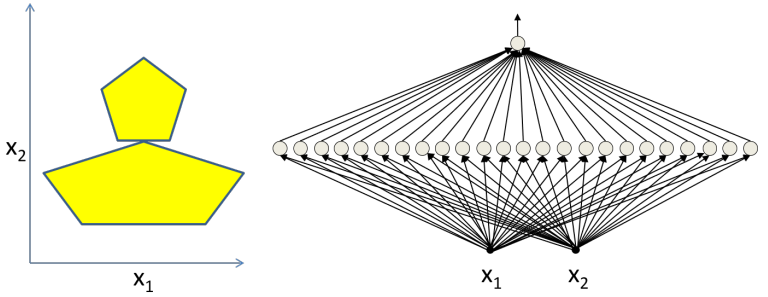
We have already seen how an MLP can operate as a Boolean function over real inputs. A Boolean function will effectively partition the input space into regions where in some regions it outputs a 0, and in other regions it outputs a 1. So we see that the function that computes the output is a classifier, and therefore the MLP effectively becomes a classifer, and computes a classification boundary.

MLPs are also **universal** classifiers. In Chapter 1, we discussed this briefly. Let us recall the examples given in Section 1.4.9, in Figs. 1.33 - 1.36. Through these examples, we saw that MLPs can actually compute very complex decision boundaries. The examples showed that given an arbitrary decision boundary, we can construct an MLP that captures it arbitrarily accurately. In our examples the MLPs that composed these functions were deep, but is this a requirement? We will understand this requirement in more detail in this section.

Let us reconsider the complex decision boundaries shown previously, shown again here in Fig. 2.23.



(a) An arbitrary decision boundary (left) can be composed in a piecewise fashion by an MLP (right).



(b) A double-pentagon decision boundary (left), and a one-hidden-layer MLP (right) that must compose it.

Figure 2.23: Composition of MLPs for complex decision boundaries

In the example shown in Fig. 2.23(a), the decision boundary shown is a complex one. It can be partitioned into many sub-regions, for each of which a subnet can be composed. Finally, all the subnets must be OR'd by a perceptron. This same mechanism can be used to compose *any* decision boundary, regardless of how complex it is.

The MLP composed for the example in Fig. 2.23(a) has two hidden layers. Let us see how this can be done with just one hidden layer, and what the consequences of doing so might be. To understand how, we refer to Fig. 2.23(b), which shows a double pentagon. How can we compose the decision boundary for this with only one hidden layer? To learn how to do this in turn, let us start with the polygon net shown in Fig. 2.24(a). This is a simple diamond-shaped decision boundary. We can compose a net for this with four hidden perceptrons, as shown in the

figure. The sum of the outputs of the first-layer perceptrons is shown against the various regions of the input, in the left panel of Fig. 2.24(a). It is also shown as a function of the two inputs in the three-dimensional plot in the central panel of Fig. 2.24(a). The sum is 4 inside the diamond. In the regions shown outside the diamond shaped boundary, the sum is not 4. It is 3 within the strips shown in yellow (left panel), and 2 in the intermediate regions, as shown. The regions where the sum is not 4 in fact extend all the way to infinity and actually represent infinite area.

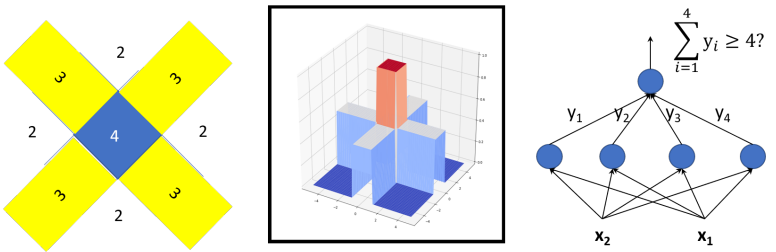
If we try to compose a pentagon istead, as shown in Fig. 2.24(b), we would have hidden 5 perceptrons. The sum of their outputs would be 5 within the pentagon, and 4 within the triangular shaped finite-area regions shaded yellow. Moreover, the yellow shaded regions are smaller than the inner pentagonal region. Also, outside these two types of regions, the regions in which the sum is 3 or 2 have infinite area.

Extending this analogy, we try to compose a hexagon in Fig. 2.24(c). We see that the region where the sum is 6 is within the hexagon, and the finite regions where the sum is 5 are each relatively smaller compared to the hexagon itself. Smaller, in fact than their analogs in the case of the pentagonal decision boundary. This is an important observation, as we will see below.

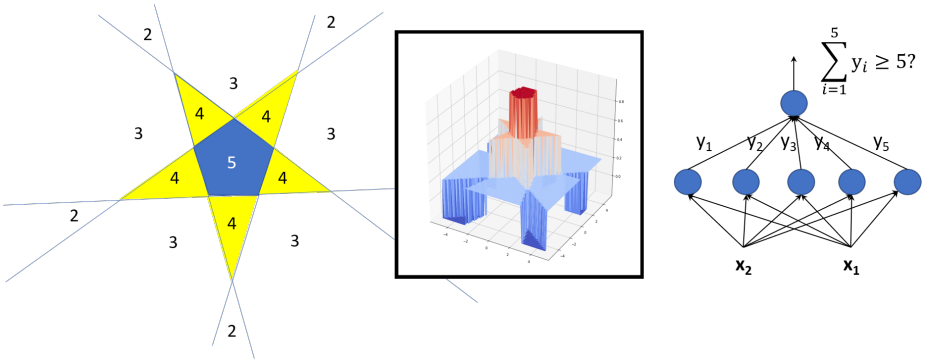
Let us now focus on the 3-dimensional plots, which show the sum of the outputs of the hidden-layer neurons (prior to the application of a threshold to obtain an actual decision boundary), while considering polygons of greater numbers of sides, as in Fig. 2.25. The examples shown include a heptagon, a polygon with with 16 sides, one with 64 sides and one with 1000 sides.

If we compose a heptagon, we obtain the plot shown in Fig. 2.25(a). The plot shown is a 3-dimensional one, with the height showing the sum. The sum is 7 in the central peaky region, the area where the sum is 6 is adjacent to it and is smaller, the region where the sum is 5 is shown by the 7-pointed star in the plot, and it too has finite area. Elsewhere the sum is either 4 or 3.

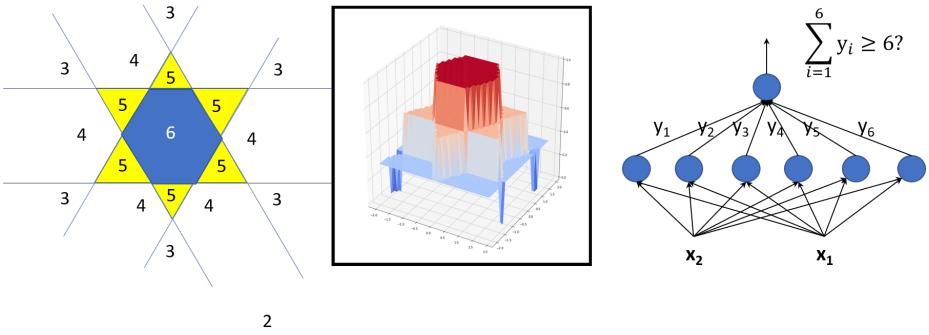
Figs. 2.25(b)-(d) show similar plots for polygons with 16, 64 and 1000 sides,



(a) The polygon net. Composing a diamond.



(b) Composing a pentagon.

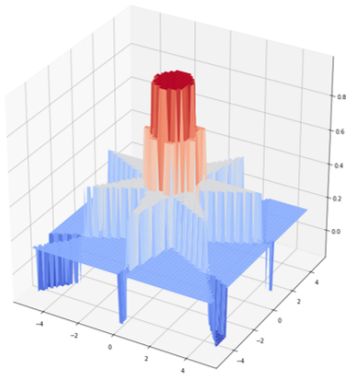


(c) Composing a hexagon.

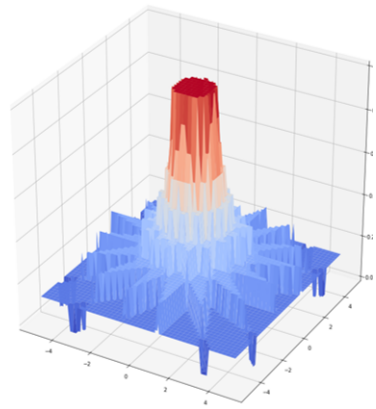
Figure 2.24: Composition of an MLP for the double pentagon

and can be similarly interpeted: there are are increasing numbers of finite-area regions with increasing number of sides, and those in turn are smaller and smaller in size, relative to the central region.

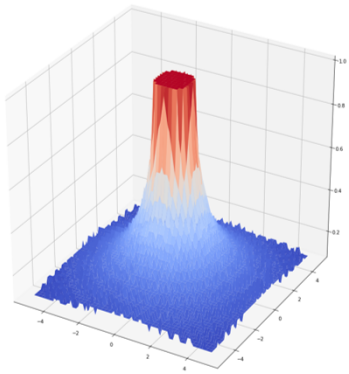
As we see from Fig. 2.26, as we increase the number of sides of an N-sided polygon, the area within which the sum of the outputs of the hidden-layer neurons lies between $N/2$ and N keeps shrinking. In the limit, as N becomes very large



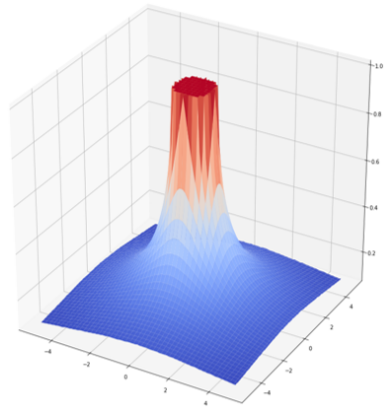
(a) Composing a heptagon.



(b) Composing a polygon with 16 sides.



(c) Composing a polygon with 64 sides.



(d) Composing a polygon with 1000 sides. As we consider the sums in the different regions, a pattern emerges as N (the number of sides of the polygon) increases beyond 6.

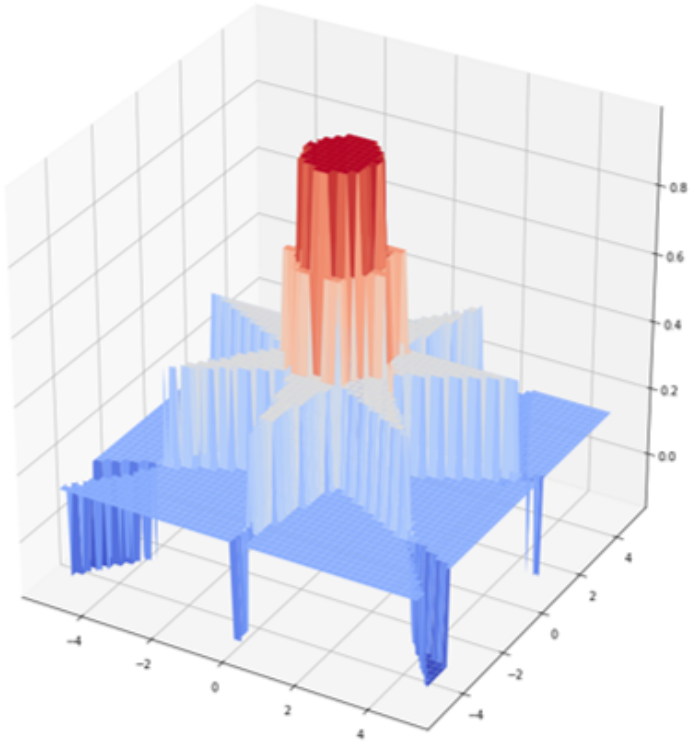
Figure 2.25: Composition of MLPs for polygons with increasing numbers of sides: a 3-D perspective

and tends to infinity, we get

$$\sum_i y_i = N\left(1 - \frac{1}{\pi} arccos\left(min\left(1, \frac{radius}{|x - center|}\right)\right)\right) \tag{2.1}$$

This is a closed-form expression for the sum of the outputs of the hidden-layer – basically the value of the sum at the output unit, as a function of distance from center, as N increases. For large N and small radius, it is a near-perfect cylinder,

with value N within the cylinder and $N/2$ outside of it. Its value quickly tapers off to $N/2$ as we move away from the center of the cylinder.

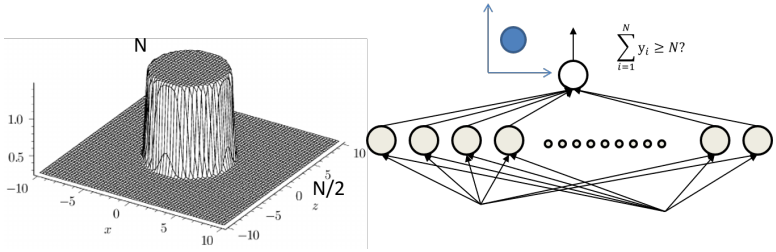


When we compose a polygonal decision boundary of a very large number of sides, increasing the number of sides reduces the area outside the polygon that have $N/2 < \sum_i y_i < N$.

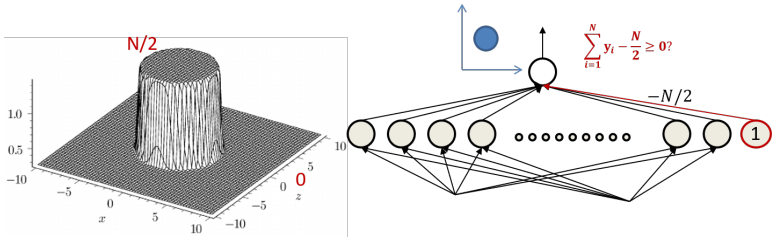
Figure 2.26: Limiting case for MLP composition of a polygonal decision boundary: 3-D perspective

Let us now compose a circle using the same process as we described above. This example is shown in Fig. 2.27(a). Using a very large number N of neurons in the hidden layer, we find that the sum of the outputs of the first-layer neurons will compose a cylinder with value N inside the cylinder, and value $N/2$ outside. If we add a bias term, as shown in Fig. 2.27(b), that subtracts $N/2$ out of the sum, we obtain a one-hidden layer network wherein the sum of the outputs of the hidden-layer neurons is exactly $N/2$ within the cylinder, and 0 outside.

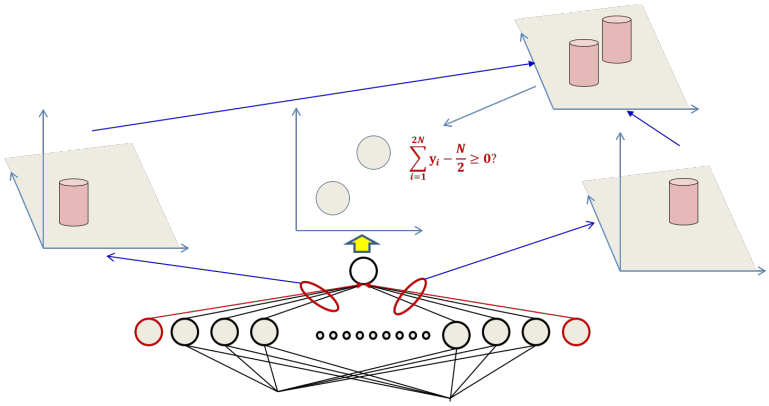
An interesting question arises in this context: what happens when we have *more*



(a) The "circle" net (MLP that models a circular decision boundary). It needs a very large number of neurons. The sum is N inside the circle, $N/2$ almost everywhere outside.



(b) Same as above with a bias term included. The sum is now $N/2$ inside the circle. The circle itself can be at any location.



(c) The "sum" of two "circle" subnets is exactly $N/2$ inside either circle, and 0 almost everywhere outside.

Figure 2.27: Composition of MLPs for a circular decision boundary

than one circle, as might be in the case of disjoint decision boundaries? To answer this, we consider Fig. 2.27(c): if we compose two sets of hidden-layer neurons, each one composing a cylinder at a different location, and if we sum the outputs of both sets of neurons, the result will have two cylinders, each of height $N/2$ within the decision regions, and 0 elsewhere. Consequently, if the output

neuron has a threshold of $N/2$, the result will be a decision boundary with two disconnected circles!

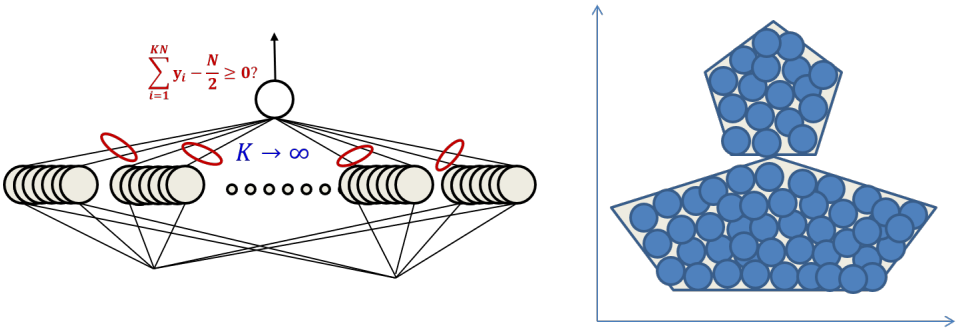
2.5.1 Composing an arbitrarily shaped decision boundary

If we want to compose an arbitrarily shaped decision boundary (e.g. the double pentagon shown in Fig. 2.28(a)) using only one hidden layer, we can do so by approximating it as a union of many small circles, and by including one subset of hidden neurons for each circle in the aggregate MLP. This approximation can be made arbitrarily precise, by filling it with more, and smaller, circles. We can thus approximate the double pentagon in Fig. 2.28(a) to arbitrary precision using only one hidden layer, though this will require an impossibly large number of neurons in the hidden layer.

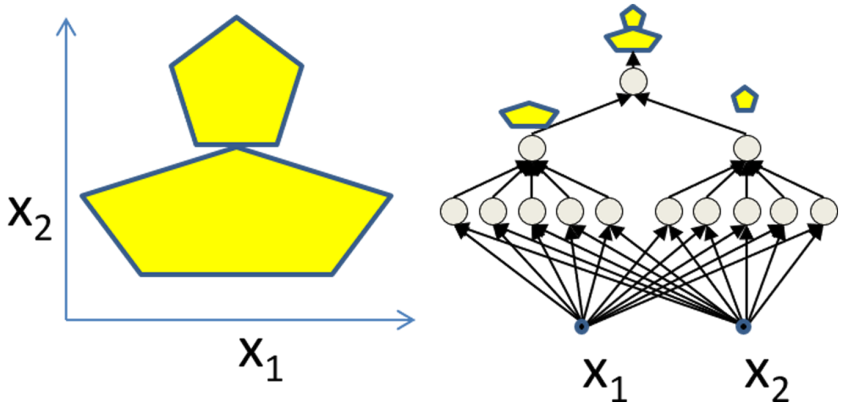
Thus, we see that MLPs can capture any classification boundary. Even a one-layer MLP can model any classification boundary, as we see in Fig. 2.28(a). In other words, regardless of the decision boundary, we can always compose an MLP, or even a one-hidden layer MLP, that captures the decision boundary to arbitrary precision. *MLPs are therefore universal classifiers*.

Fig. 2.28(b) illustrates a final point, relating to depth in the context of a universal classifier. Simply making the network deeper can result in a massive reduction in the number of neurons required to model the function. For the double pentagon shown, a one-hidden layer network will require an impossibly large number of neurons, but if we do it with two hidden layers, we only need 13 neurons!

This of course raises the issue of optimal depth for an MLP. A lot of formal analyses has been done on the topic of optimal network size and depth. These analyses typically view the networks as instances of arithmetic circuits, and approach the problem as one of computing polynomials over any field. Multiple hypotheses have been offered for this: for example, accoring to Valiant et. al. [2, 3], a polynomial of degree n requires a network (or circuit) of depth $log^2(n)$, to keep the circuit from blowing up. A shallower network cannot suffice. The



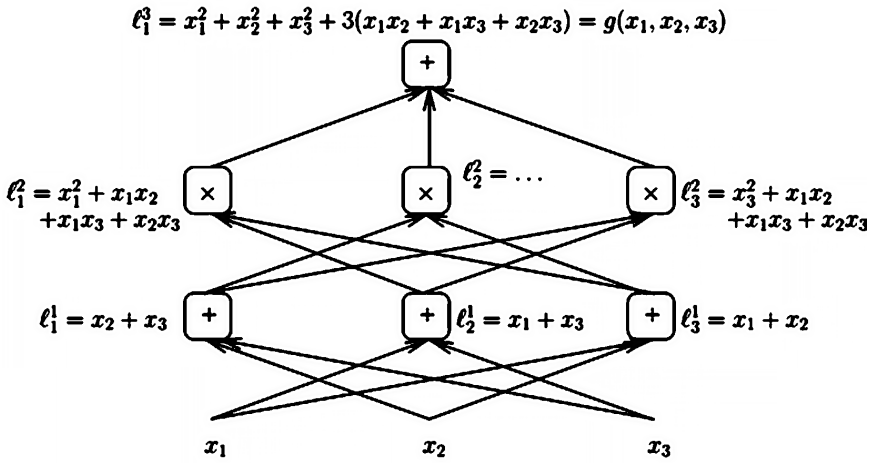
(a) Approximating an arbitrarily shaped decision boundary with circles. We can approximate the double pentagon to arbitrary precision using only one hidden layer in the corresponding MLP, though this will require an impossibly large number of neurons in the hidden layer. More accurate approximation with greater number of smaller circles can achieve arbitrary precision.



(b) An MLP with more layers requires fewer neurons to model the same classification boundary and as MLP with just one layer. With two layers, the MLP requires 13 neurons to model the decision boundary shown on the left.

Figure 2.28: Composition of MLPs for an arbitrarily shaped decision boundary

majority of functions are very high (possibly infinite) order polynomials. Bengio et. al. [4] show a similar result for sum-product networks, but only consider two-input units, as shown in Fig. 2.29. This has been generalized by Mhaskar et. al. [5] to all functions that can be expressed as a binary tree. Analysis of optimal depth and the size of arithmetic circuits is an ongoing research topic. From our perspective, it is sufficient to remember that when neural nets get deeper, they can get much smaller, sometimes exponentially smaller, to compute the same function.



"Shallow vs deep sum-product networks," (From [4]). For networks where layers alternately perform either sums or products, a deep network may require an exponentially fewer number of layers than a shallow one.

Figure 2.29: Sum product network

Fun facts 5.1: Depth in sum-product nets

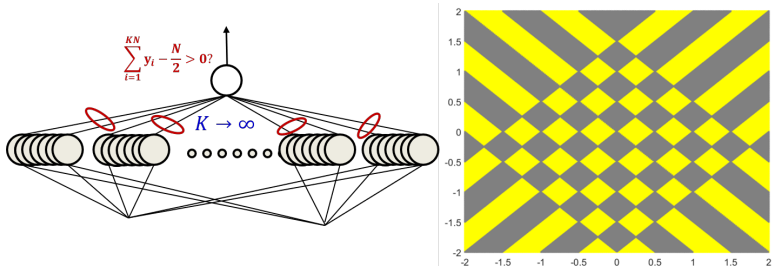
Theorem 5: A certain class of functions F of n inputs can be represented using a deep network with $O(n)$ units, whereas it would require $O(2^{\sqrt{n}})$ units for a shallow network.

Theorem 6: For a certain class of functions G of n inputs, the deep sum-product network with depth k can be represented with $O(nk)$ units, whereas it would require $O(n-1)^k$ units for a shallow network.

Let us now consider the subject of optimal depth in *generic* nets. We know through the discussions so far that deeper networks are better in that they require much fewer neurons to represent most functions. They are also more versatile when it comes to representing decision boundaries. To understand this better, let us look at a different kind of decision boundary – something that could be considered to be a "difficult" decision boundary. The example we choose is shown in the panel on the right in Fig. 2.30. Let us analyze threshold-activation

networks in the context of this one case. The analysis should generalize to other networks as well.

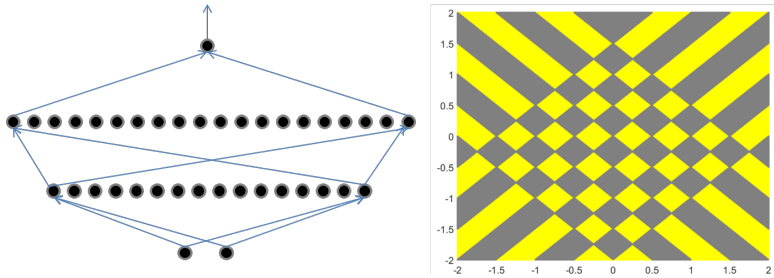
As we see in Fig. 2.30 to 2.39, a naïve one-hidden-layer neural network will require infinite hidden neurons to model this boundary[4].



Right: A worst-case decision boundary (right). The network shown (left) must output a 1 when the input is in the yellow regions, and 0 for the gray regions.

Figure 2.30: Optimal depth for a generic net that models a worst-case decision boundary: Panel 1

However, if we do this with two hidden layers, we need only 56 hidden neurons, as shown in Fig. 2.31.



A two hidden-layer network will need 56 hidden neurons.

Figure 2.31: Optimal depth for a generic net that models a worst-case decision boundary: Panel 2

This is because the decision boundary is composed of 16 lines, as shown in Fig. 2.32. We need 16 perceptrons in the first hidden layer to capture each of the 16 lines.

[4]It can in fact be modelled by a *smart* one-hidden-layer network. We leave the derivation of this to the reader. We do not consider this solution for our illustrative exposition, however

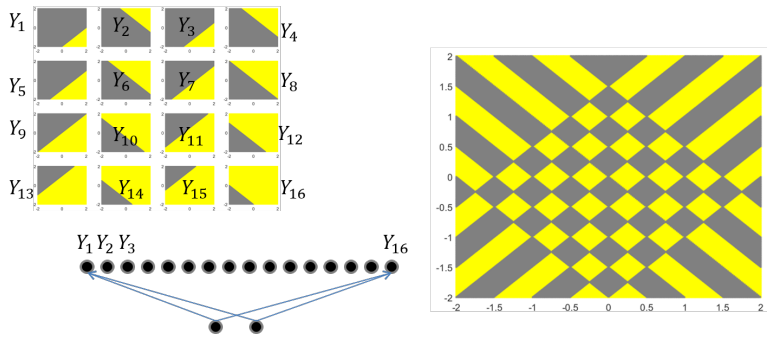


Figure 2.32: Optimal depth for a generic net that models a worst-case decision boundary: Panel 3

We see from Fig. 2.33 that there are 40 yellow regions in all. Therefore we will need 40 perceptrons in the second hidden layer, one to capture each of the yellow regions using the linear boundaries captured by the first layer. We must also use one additional final perceptron that ORs the outputs of the 40 perceptrons. So in total we have $40 + 16 = 56$ hidden neurons, and one output neuron, giving us a total of 57 neurons.

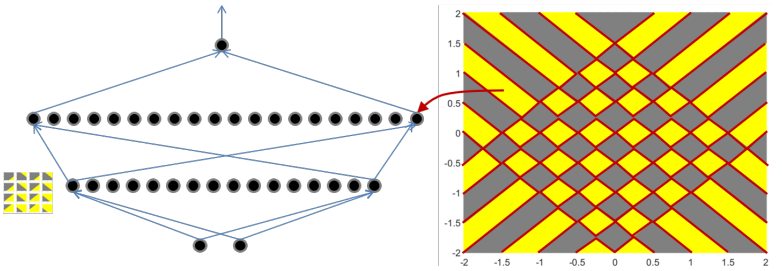
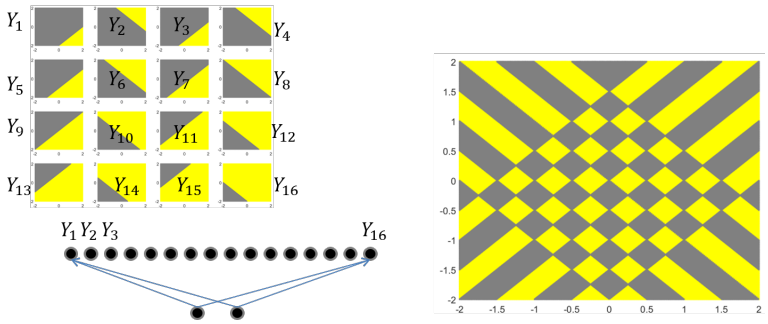


Figure 2.33: Optimal depth for a generic net that models a worst-case decision boundary: Panel 4

Fig. 2.34 shows that this pattern is just an XOR pattern. If we number the outputs of the first 16 neurons which capture the 16 lines, and call them Y_1 through Y_{16}, then the entire decision boundary is simply expressed as $Y_1 \oplus Y_2 \oplus \ldots \oplus Y_{16}$ (where $\oplus$ denotes XOR).

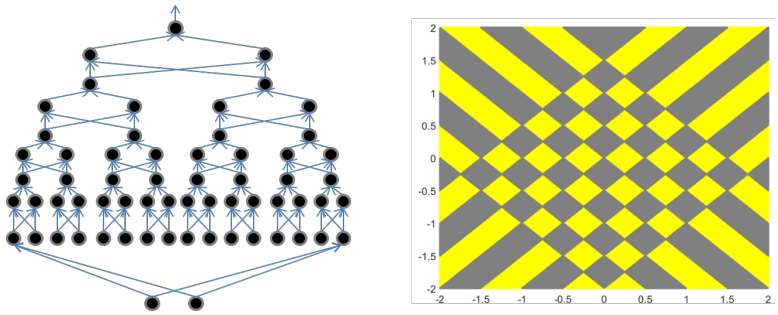
If instead we compose the decision boundary using an XOR network, as shown



The network shown corresponds to $Y_1 \oplus Y_2 \oplus \ldots \oplus Y_{16}$.

Figure 2.34: Optimal depth for a generic net that models a worst-case decision boundary: Panel 5

in Fig. 2.35, the XOR part of it will only require $15 \times 3 = 15$ XORs, each requiring 3 neurons. Thus the total number of neurons we would use is 61. In fact, if we use the two-neuron version of XOR, which uses 2 neurons per XOR, it will require only 46 neurons. The gain from modeling this example as an XOR may not look like much, but the advantage becomes apparent when we scale up the problem a little, as we do in Fig. 2.36.

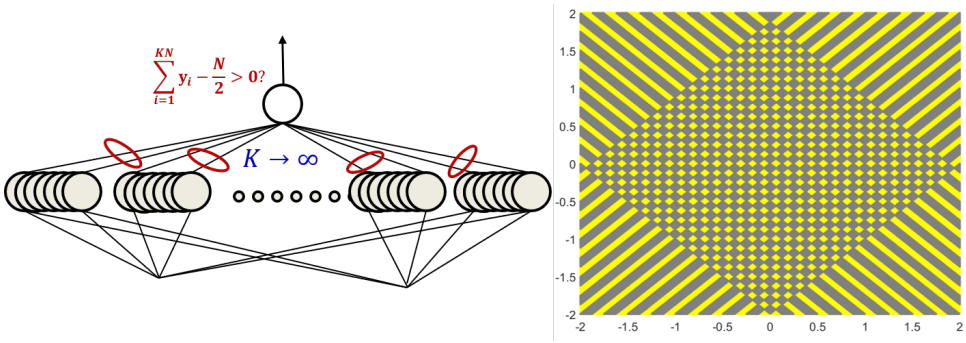


Composing the decision boundary using an XOR network would require 61 neurons.

Figure 2.35: Optimal depth for a generic net that models a worst-case decision boundary: Panel 6

The decision pattern in Fig. 2.36 is more complex. Once again, a naïve one-hidden-layer network will require infinite hidden neurons to model this.

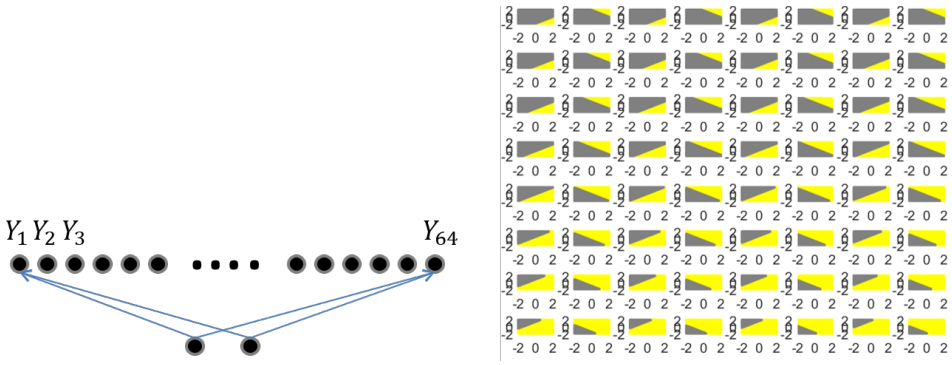
In Fig. 2.37 we consider the actual linear boundaries that compose the pattern. We see that the complex pattern shown is composed of 64 lines, each of which



A more complex version of the decision bondaries shown above.

Figure 2.36: Optimal depth for a generic net that models a worst-case decision boundary: Panel 7

will require a perceptron to capture. We can capture the 64 lines with one hidden layer of 64 perceptrons.

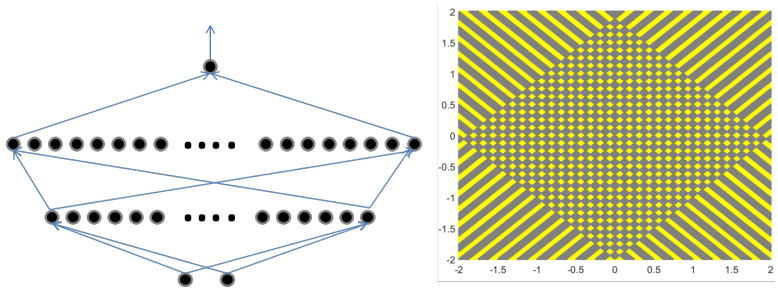


64 linear units can be captured with one layer of 64 perceptrons.

Figure 2.37: Optimal depth for a generic net that models a worst-case decision boundary: Panel 8

Fig. 2.38 shows the full network. The pattern has 544 yellow regions. For this, we need 544 neurons in the second layer. Including the output neuron, a two-layer network will therefore require 609 total neurons.

However, once again, as shown in Fig. 2.39, this is an XOR over the 64 lines. An XOR network that uses 3 neurons per XOR can do this with only 253 neurons. If we compose it with 2 neurons per XOR, the network will require only 190 neurons. We see from this example that the difference in size between the deeper

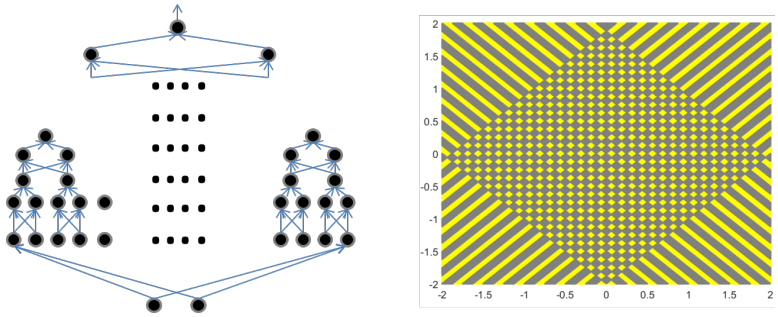


For the decision boundary shown, we need 544 neurons in the second layer, and a total of 609 neurons.

Figure 2.38: Optimal depth for a generic net that models a worst-case decision boundary: Panel 9

optimal network, and the shallower net tends to increase rapidly with increasing pattern complexity and increasing input dimensionality.

To summarize, the number of neurons required by a shallow network is potentially exponential in the dimensionality of the input, or alternately in the number of statistically independent features in the input, while a deep network can require far fewer neurons to capture the same function.



An XOR network with 12 hidden layers. It has 253 neurons (190 neurons with 2-gate XOR). The difference in size between the deeper optimal (XOR) net and shallower nets increases with increasing pattern complexity and input dimension.

Figure 2.39: Optimal depth for a generic net that models a worst-case decision boundary: Panel 10

Fun facts 5.2

In the problem of the complex decision boundary discussed above, the 2-layer network was quadratic in the number of lines, with $(N+2)^2/8$ neurons in the 2^{nd} hidden layer. Thus, even though the pattern is represented as an XOR pattern, the composition is not exponential in the number of neurons.

This is so because the data are two-dimensional. They only have two fully independent features, and the pattern is exponential in the dimension of the input (two).

For the general case of N mutually intersecting hyperplanes in D dimensions, we will need $O(\frac{N^D}{(D-1)!}))$ weights, assuming $N > D$.

Increasing the input dimensions can increase the worst-case size of the shallower network exponentially, but not the XOR network. The size of the XOR net depends only on the number of first-level linear detectors (N).

Recap 5.1

1. Multi-layer perceptrons are Universal Boolean Machines
2. Even a network with a single hidden layer is a universal Boolean machine
3. Multi-layer perceptrons are Universal Classification Functions
4. Even a network with a single hidden layer is a universal classifier
5. A single-layer network may require an exponentially large number of perceptrons than a deep one
6. Deeper networks may require far fewer (exponentially fewer) neurons than shallower networks to express the same function Could be exponentially smaller
7. Deeper networks are more expressive

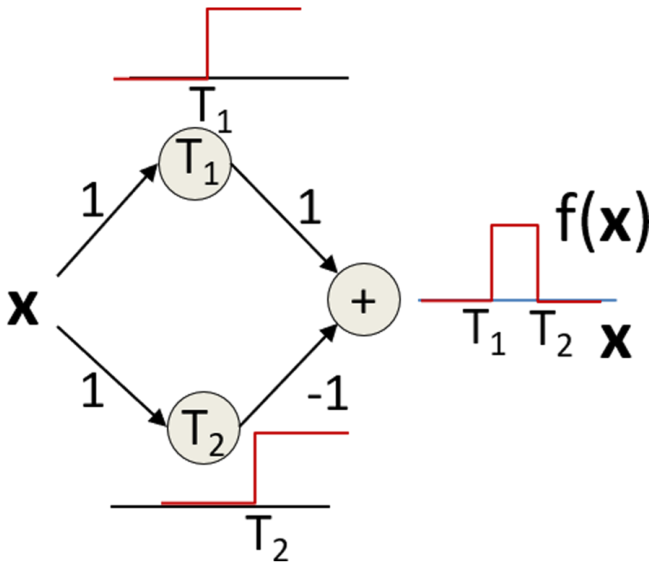
So far in this chapter we have seen that MLPs are universal Boolean functions, and are also universal classifiers. Let us now understand how MLPs are also **universal approximators** of any function.

2.6 MLPs as universal approximators

MLPs are also **universal approximators**. Given any input-output relationship, it is possible to construct an MLP that approximates it to arbitrary precision (of course, for this to be true, requirements for optimal depth and width must be met).

To understand this, we first consider an MLP as a *continuous valued regression*. In Chapter 1, we saw how an MLP with 2 perceptrons can compute a step func-

tion of a scalar input. This example is shown again in Fig. 2.40 for reference. The two perceptrons in this example have different thresholds, T_1 and T_2 (assuming w.l.o.g. that $T_1 < T_2$), and their outputs are summed with weight 1 and -1, so the overall net produces a 1 for input between T_1 and T_2, and 0 elsewhere.



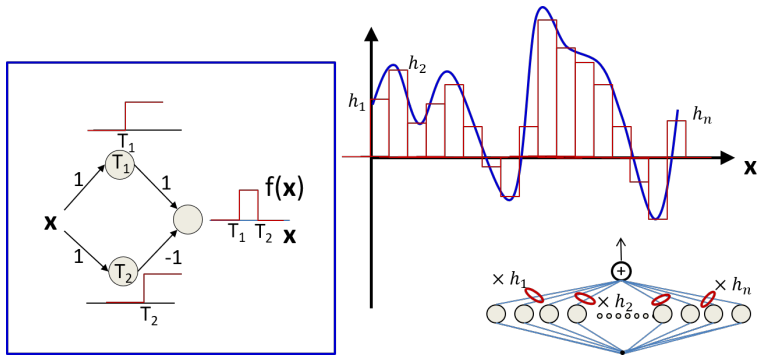
A simple 3-unit MLP with a "summing" output unit can generate a "square pulse" over an input. The output is 1 only if the input lies between T_1 and T_2. T_1 and T_2 can be arbitrarily specified.

Figure 2.40: Computing a step function of a scalar input with an MLP with 2 perceptrons

A one-hidden-layer MLP can model arbitrary scalar function of a scalar variable. For the example in Fig 2.41, we can subsequently compose any scalar function of a scalar variable with a one-hidden layer MLP, by approximating the function as the sum of many such pulses, all scaled to the right height.

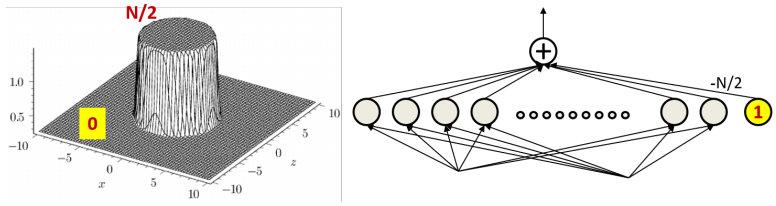
For functions in higher dimensions, this gets more involved. Let us understand the case of higher dimensions with the help of the example in Fig. 2.42. We begin by recalling that a one-hidden-layer MLP can compose a cylinder that has height $N/2$ in the center and 0 outside.

Thus, to compute an arbitrary function in higher dimensions, we approximate



Left: A simple 3-unit MLP can generate a "square pulse" over an input. **Right:** An MLP with many units can model an arbitrary function over an input to arbitrary precision. For this, we simply make the individual pulses narrower. A one-hidden-layer MLP can model an arbitrary function of a single input.

Figure 2.41: MLP as a continueous-valued regression

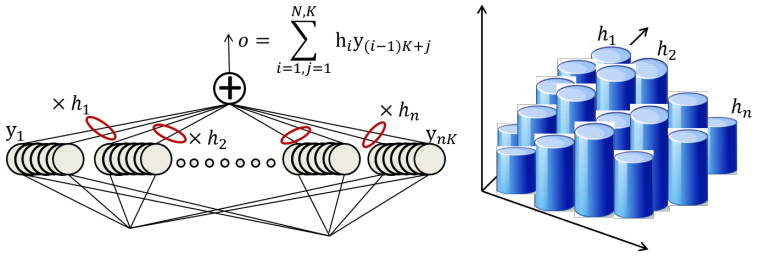


An MLP can compose a cylinder. The value is $N/2$ in the circle, and 0 outside.

Figure 2.42: MLP composition of a cylinder

it by summing appropriately scaled and shifted cylinders, as illustrated in Fig. 2.43). We can approximate the function to arbitrary precision by making the cylinders thinner. Thus, the one-hidden layer MLP can be viewed as a universal function approximator.

Although MLPs with additive output units are universal approximators, there is one caveat to this, which can also be understood with the help of Fig. 2.43. Note the manner in which the MLP in this figure is composed: the neurons in the hidden-layer compose the cylinders. The output neuron simply added the scaled outputs of the hidden layer neurons. The caveat is that the entire universal approximator theory, as we explained it, *only holds if the output neuron doesn't*



MLPs can actually compose arbitrary functions in any number of dimensions with arbitrary precision, even with only one layer, as sums of scaled and shifted cylinders. This can be done by making the cylinders thinner.

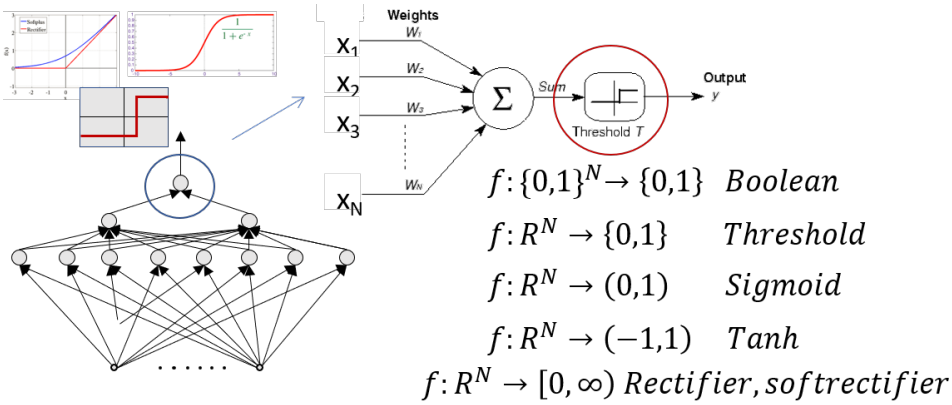
Figure 2.43: MLP composition of an arbitrary function

have a non-linear activation function.

In most "proper" networks that are used in practical applications, the output neuron will generally have an activation function, like a sigmoid, tanh, rectification function or softplus, as shown in Fig. 2.44. What can we say about the universality of such networks? The network shown in Fig. 2.44 can be viewed as a function. When the output neuron has an activation function with some range, e.g. (0,1), then the entire network can only produce outputs in that range. In this case the network is actually a *universal map* from the input domain to the range of the activation function.

In other words, the MLP is a universal approximator for the *entire class of functions* that it represents, and it includes all possible maps from the domain of inputs to the range of outputs of the final activation.

As the final topic in this Chapter, we will briefly discuss the notion of sufficiency of width and depth, and its relation to the activation function.



In "proper" networks, the output neuron may have actual "activation," such as the ones metioned in this figure: Threshold, Sigmoid, Tanh, Softplus, Rectifier, etc. The output unit with activation function is shown circled in blue, and it corresponding perceptron representation is shown on the right, with the activation function circled in red. The network is actually a universal map from the entire domain of input values to the range of the output activation function values. These ranges are listed for some functions on the bottom right.

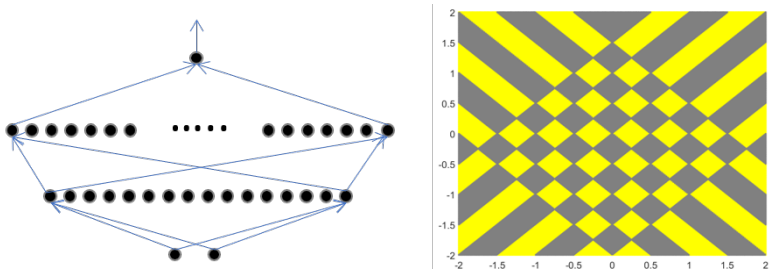
Figure 2.44: The MLP as a universal map into the range of activation function values

2.7 The issue of depth

We have seen that a single-hidden-layer MLP is a universal function approximator. It can approximate any function to arbitrary precision, although it may need a very large number of hidden neurons. More generally, deeper networks will require far fewer neurons than shallower networks to achieve the same accuracy of approximation of a desired function. This is true regardless of whether we use them to model Boolean functions, real-valued functions or as (discrete-valued) classifiers. However, there are some limitations that we must be aware of.

The first is **sufficiency of architecture**. A neural network can represent any function provided it has sufficient capacity – i.e., is sufficiently broad and sufficiently deep to represent the function. Not all architectures can represent all functions. For example, consider the classification function in Fig. 2.45. Let us suppose we want to compose a network to produce 1 in the yellow regions and 0 in the grey

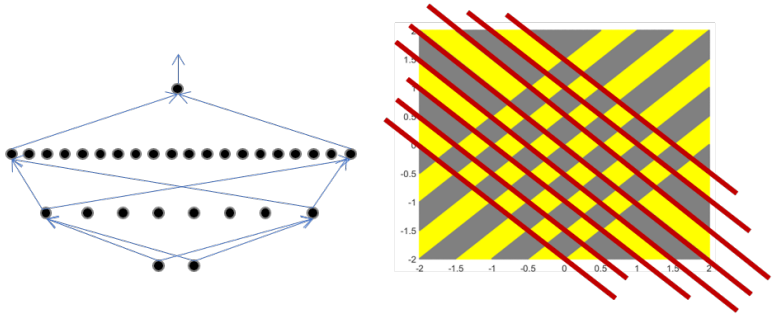
regions. Consider a model where the perceptrons we use have threshold activations. Since the pattern we have is composed using 16 lines, a network with 16 or more neurons in the first layer would be capable of representing this decision pattern perfectly.



Since the pattern we have is composed using 16 lines, a network with 16 or more neurons in the first layer would be capable of representing this decision pattern perfectly.

Figure 2.45: Sufficiency of architecture: Part 1

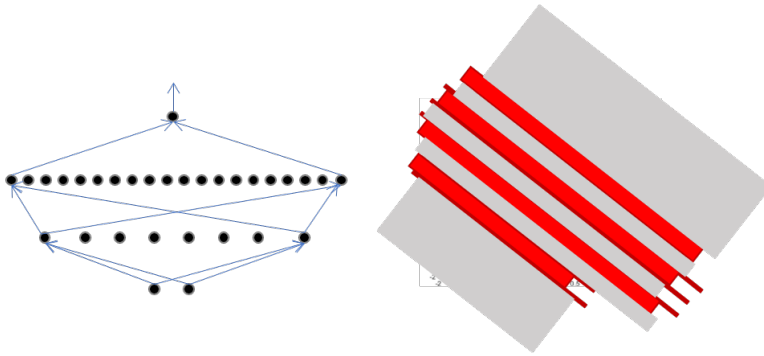
But if the network has less than 16 neurons in the first layer, for example if it has only 8 neurons, it cannot represent the pattern shown exactly. In Fig. 2.46, let us say the 8 neurons capture the 8 lines shown in red.



Assuming the 8 neurons capture the 8 lines shown in red.

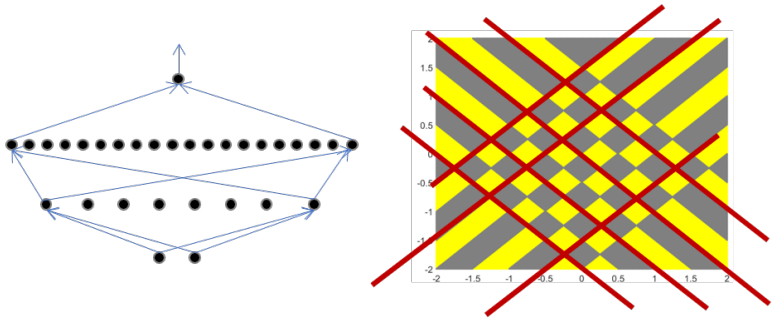
Figure 2.46: Sufficiency of architecture: Part 2

We see that the output of the first layer will only retain the kind of information shown in the Fig. 2.47. It can tell us which of the 9 strips an input is in, but it cannot tell us *where* it is in the strip. It absolutely will not give us any information about the position transverse to the lines shown. It doesn't matter how many layers or neurons we add after this, we simply cannot construct the grid-shaped decision boundaries we need.



The output of the first layer can tell us which of the 9 strips the input is in, but it cannot tell us *where* it is in the strip. We cannot construct a grid from this output.

Figure 2.47: Sufficiency of architecture: Part 3

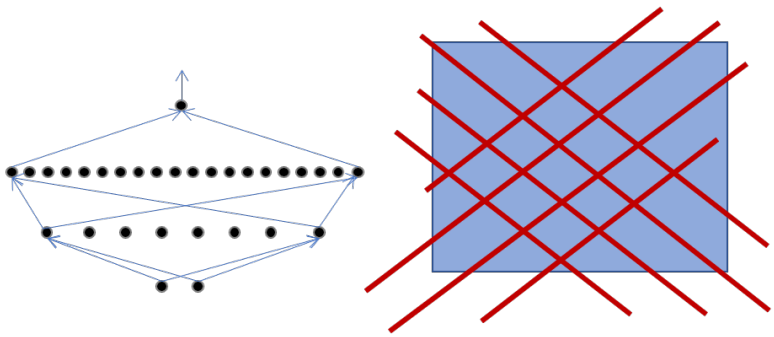


We cannot construct a grid from this output even if the 8 neurons capture these 8 lines.

Figure 2.48: Sufficiency of architecture: Part 4

Even if the 8 neurons capture the 8 lines shown in Fig. 2.48, we will only have the level of information shown in Fig. 2.49 – we can determine which of the larger cells of Fig. 2.49 the input is in, but the output of first layer remains constant within any of the cells. Thus, while can determine which of the 25 regions an input lies in, there is insufficient information to make additional distinctions within the cells itself. As a result, regardless of how many new layers or neurons we add in layers above the initial 8 neuron layer, that fact that we had only 8 neurons in the initial layer essentially restricts us from ever being able to model the function.

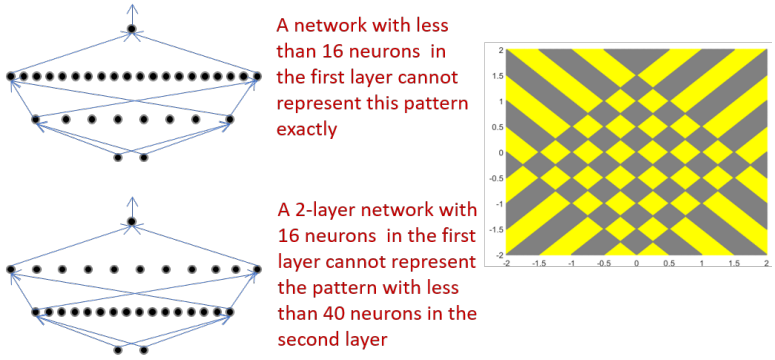
As shown in Fig. 2.50, even if we have the full complement of 16 perceptrons in



Regardless of how many new layers or neurons we add in layers above the initial 8 neuron layer, that fact that we had only 8 neurons in the initial layer essentially restricts us from ever being able to model the function.

Figure 2.49: Sufficiency of architecture: Part 5

the first hidden layer, if we have only 2 hidden layers, and fewer than 40 neurons in the second layer we cannot compose the function shown perfectly. In other words, if we make our network deep, rather than wide, we have to ensure that each layer is sufficiently wide. If this is not so, then it does not matter how deep we make our network, the function cannot be modeled.



Even if we have the full complement of 16 perceptrons in the first hidden layer, if we have only 2 layers, and fewer than 40 neurons in the second layer we cannot compose the function shown perfectly.

Figure 2.50: Sufficiency of architecture: Part 6

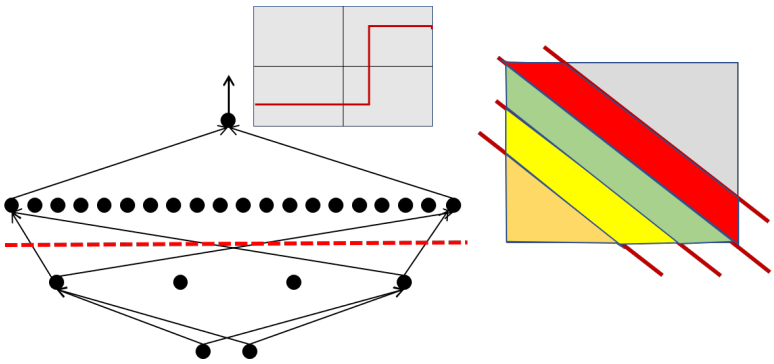
There us a caveat to this, though. We saw that a network with fewer than 16 neurons in the first hidden layer could never compose the decision pattern shown in the figures above (see especially Fig. 2.49). Why would this be so?

The answer to this question may become evident from the examples in Fig. 2.51(a)-(c): it is because in all of our explanations, we used the threshold activation, which gates information in the input and prevents it from reaching later layers. So, if we had only 4 neurons in the first layer, each neuron will tell us whether the input is on one side or the other of a line or hyperplane, but it will not tell us how far we are from the line. As a result, if we had only 4 neurons in the first hidden layer, the output of the first layer will partition the input space into a small set of regions, like the strips shown in 2.51(a). An input that falls anywhere in one of the regions shown, e.g. the yellow strip, will produce the exact same output from the first layer regardless of its position within the strip. As a result subsequent layers in the network do not have sufficient information to partition the space as needed.

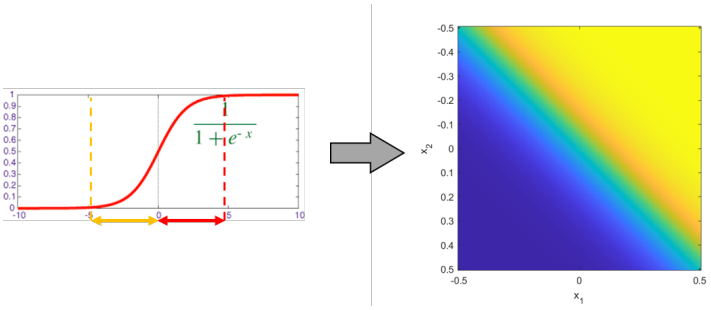
This happens because we used threshold activations which only output a binary characterization of which side of a boundary an input lies in, but lose information about how far it is from the boundary. If, on the other hand we used a continuous activation function like a sigmoid, it will produce a graded output (that looks roughly as shown in Fig. 2.51(b) when we view it from the top – the value of the output indicates the distance from the boundary). This information is passed to subsequent layers, which can use it to capture information missed by the first layer. It is important to note here that while the use of a sigmoid would give us information about the distance from the boundary, it would do so only for a limited distance. Beyond a point the sigmoid saturates (to within arithmetic precision), and we lose further information about how far we are.

A better activation is one that will continue to give us useful information about the distance from the boundary, no matter how far we are. A RELU does this for instance, but only on one side of the boundary. A better activation would probably be the leaky RELU, which gives us information on both sides of the boundary. Other continuous activations do this too, albeit to differing degrees.

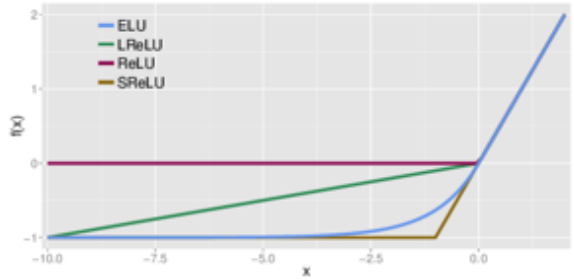
The discussion above helps us understand the interplay of width, activations and depth in the composition of such networks. We see that narrow layers can pass sufficient information to subsequent layers, provided the activation function is



(a) The pattern of outputs within any colored region is identical. Subsequent layers do not obtain enough information to partition them. This effect is because we use the threshold activation. It gates information in the input from later layers.



(b) Continuous activation functions result in graded output at the layer. The gradation provides information to subsequent layers, to capture information "missed" by the lower layer (i.e. it "passes" information to subsequent layers).



(c) Activations with more gradation (e.g. RELU) pass more information.

Figure 2.51: Sufficiency of architecture: Part 7

sufficiently graded. However, in order to *completely* capture the function, we will need greater depth in the network, to permit later layers to capture the details of

the function.

The notion of the *capacity of a network* now becomes clear. While there are various definitions of capacity, from our perspective, it relates to the largest number of disconnected regions of the input that it can represent. A network with insufficient capacity cannot exactly model a function that requires a greater minimal number of convex hulls that the capacity of the network, although it can model it with error.

Fun facts 7.1: Capacity of a network

The capacity of a network has various definitions:

1. **Information or Storage capacity:** how many patterns can it remember
2. **VC dimension:** bounded by the square of the number of weights in the network
   - Koiran and Sontag (1998): For "linear" or threshold units, VC dimension is proportional to the number of weights. For units with piecewise linear activation it is proportional to the square of the number of weights.
   - Batlett, Harvey, Liaw, Mehrabian (2017) [7]: For any W, L s.t. $W > CL > C^2$, there exisits a RELU network with $\leq L$ layers, $\leq W$ weights with VC dimension $\geq (WL/C)log_2(W/L)$.
   - Friedland, Krell (2017) [8]: VC dimension of a linear/threshold net is $O(MK)$, M is the overall number of hidden neurons, K is the weights per neuron.

A network with insufficient capacity cannot exactly model a function that requires a greater minimal number of convex hulls than the capacity of the network, but can approximate it with some error.

Recap 7.1

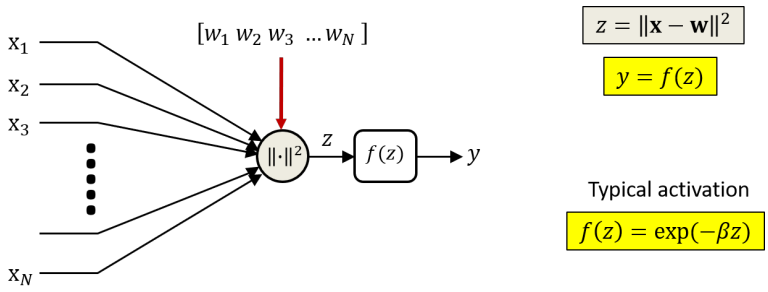
1. MLPs are universal Boolean function
2. MLPs are universal classifiers
3. MLPs are universal function approximators
4. A single-layer MLP can approximate anything to arbitrary precision, but could be exponentially or even infinitely wide in its inputs size
5. Deeper MLPs can achieve the same precision with far fewer neurons – i.e., deeper networks are more expressive

2.8 RBF networks: a brief introduction

The outputs of the activations of the perceptrons we have seen so far varied with the distance from a hyperplane – a linear surface in the input space. An alternate type of unit is the "Radial Basis Function" or RBF unit, whose output varies with the (radial) distance from a point – a "center". Networks composed of these units are RBF networks.

Fig. 2.52(a) shows a typical RBF. The "center" w is the parameter specifying the unit. The unit computes the ℓ_2 distance of the unit from its center (instead of the affine combination computed by regular perceptrons), and passes this value through an activation. The most commonly used activation is the exponent, shown in the figure. In this, β is a "bandwidth" parameter. Other similar activations may also be used. The key aspect of these is their radial symmetry (instead of linear symmetry).

Due to their radial symmetry, RBFs can compose cylinder-like outputs with just



(a) The "center" w is the parameter specifying the unit. The output is a function of the distance of the input from this center.

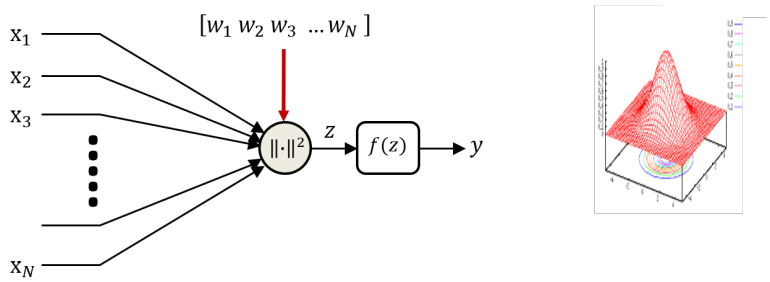
Figure 2.52: The relationship of output to center of a unit

a single unit with appropriate choice of bandwidth (and activation), as shown in Fig. 2.53(a), as opposed to $N \to \infty$ units for the linear perceptron.

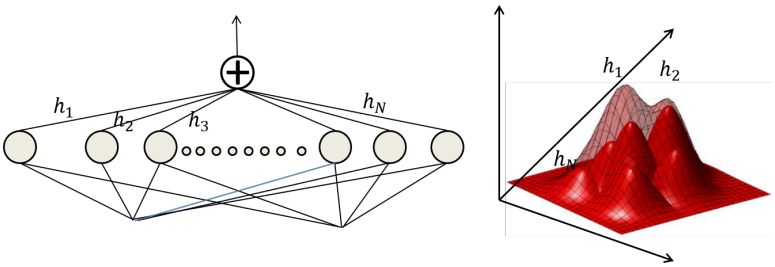
From Fig. 2.53(b) we see that RBF networks are more effective approximators of continuous-valued functions – a one-hidden-layer net only requires one unit per "cylinder." From Fig. 2.53(c), we see that RBF networks are also universal approximators, and can model any complex decision boundary.

RBF networks are in fact more effective than conventional linear perceptron networks in some problems.

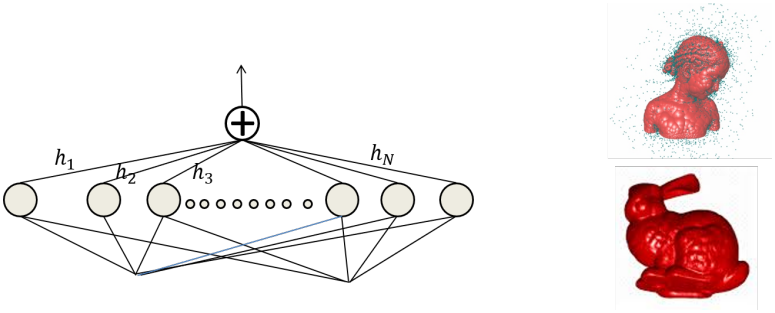
In this chapter we learned that MLPs can emulate any function. However, we must still find out how we can use an MLP to emulate any specific *type* of desired function, e.g. a function that takes an image as input and outputs the labels of all objects in it, or a function that takes speech input and outputs the labels of all phonemes in it etc. This – the *training* of an MLP to emulate a desired function – will be the topic of the next Chapter.



(a) Radial basis functions can compose cylinder-like outputs with just a single unit with appropriate choice of bandwidth (or activation function), as opposed to $N \to \infty$ units for the linear perceptron.



(b) RBF networks are more effective approximators of continuous-valued functions A one-hidden-layer net only requires one unit per "cylinder."



(c) RBF networks can model complex functions.

Figure 2.53: Radial Basis Function networks

Bibliography

[1] Karnaugh, M., 1953. The map method for synthesis of combinational logic circuits. Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics, 72(5), pp.593-599.

[2] Valiant, L.G. and Skyum, S., 1981, August. Fast parallel computation of polynomials using few processors. In International Symposium on Mathematical Foundations of Computer Science (pp. 132-139). Springer, Berlin, Heidelberg.

[3] Judd, J.S., 1990. Neural network design and the complexity of learning. MIT press.

[4] Delalleau, O. and Bengio, Y., 2011. Shallow vs. deep sum-product networks. In Advances in neural information processing systems (pp. 666-674).

[5] Poggio, T., Mhaskar, H., Rosasco, L., Miranda, B. and Liao, Q., 2017. Why and when can deep-but not shallow-networks avoid the curse of dimensionality: a review. International Journal of Automation and Computing, 14(5), pp.503-519.

[6] Furst, M., Saxe, J.B. and Sipser, M., 1984. Parity, circuits, and the polynomial-time hierarchy. Mathematical systems theory, 17(1), pp.13-27.

[7] Harvey, N., Liaw, C. and Mehrabian, A., 2017, June. Nearly-tight VC-dimension bounds for piecewise linear neural networks. In Conference on Learning Theory (pp. 1064-1068).

[8] Friedland, G. and Krell, M., 2017. A capacity scaling law for artificial neural networks. arXiv preprint arXiv:1708.06019.