Chapter 3

Shallow Networks and Shallow Learning

In this chapter we focus on layered feedforward shallow networks, i.e. feedforward networks with no hidden units, and the corresponding shallow learning problem. Thus, we consider $A(n, 1)$ and $A(n, m)$ architectures. We study the questions of design, capacity, and learning in that order. We begin by showing that the Bayesian statistical framework leads to a fairly complete theory of how to design such networks in the supervised learning setting. In particular, it provides a principled and elegant way to derive the transfer functions and the error functions in both the regression and classification cases, in a way that leads to the same simple gradient equations across the cases.

3.1 Supervised Shallow Networks and their Design

We begin with the supervised learning framework with a single unit for regression or classification problems, or k units for classification into k classes. Assume that the data consists of K input-target pairs: $D = \{(I(t), T(t)) \ t = 1, \ldots, K\}$ and w represents the weights. Then, assuming the data pairs are independent of each other and the inputs are independent of the weights, the likelihood has the form:

$$P(D|w) = \prod_{t=1}^{K} P(I(t), T(t)|w) = \prod_{t=1}^{K} P(T(t)|I(t), w)P(I(t)|w) \tag{3.1}$$

so that:

$$P(D|w) = \prod_{t=1}^{K} P(T(t)|O(t))P(I(t)) \tag{3.2}$$

where O is the output of the network. Throughout this section, probabilities for continuous values should be written with the usual notation $P(x < X < x + \Delta x) = P(x)\Delta x$. Here we omit the Δx terms as they play no role in the calculations and the final answers. For simplicity, we also assume that all the inputs have the same probability ($P(I(t) = c)$ Thus, in this case, the maximum likelihood (ML) estimation problem becomes:

$$\min_w \mathcal{E} = \min_w \sum_{t=1}^{K} \mathcal{E}(t) = \min_w \left[-\sum_{t=1}^{K} \log P(T(t)|O(t)) \right] \tag{3.3}$$

where $\mathcal{E}(t) = -\log P(T(t)|O(t)$ can be viewed as an error function measuring the mismatch between outputs and targets for each training example. The corresponding maximum a posteriori (MAP) problem is given by:

$$\min_w \mathcal{E}' = \min_w [-\sum_{t=1}^{K} \log P(T(t)|O(t)) - \log P(w)] \tag{3.4}$$

Next, we must now specify the probabilistic model needed to compute $P(T(t)|O(t)$, as well as $P(w)$. To simplify the notation, we treat a single generic example, dropping the index t. At the end, this procedure will give the error and learning algorithms corresponding to online learning, i.e. example by example. Alternatively, for batch or mini-batch learning, one must remember to sum the corresponding expressions over the corresponding examples.

3.1.1 Regression

In the case of a regression problem, one can use a simple Gaussian model:

$$P(T|O) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(T-O)^2/2\sigma^2} \tag{3.5}$$

As a result, the ML equation $\min_w -\log P(T|O)$ is equivalent to the usual least square problem:

$$\min_w \mathcal{E} = \min_w \frac{1}{2}(T-O)^2 \tag{3.6}$$

In the case of regression, the targets are not necessarily bounded, and so it is natural to use a linear unit with $O = S = \sum w_i I_i$. In this case:

$$\partial\mathcal{E}/\partial S = -(T-O) \tag{3.7}$$

Note that more complex models, where for instance σ is not the same for all the points, can easily be incorporated into this framework and would lead to a weighted least square problem.

3.1.2 Classification

In the case of binary classification, one can use a simple binomial model, so that:

$$P(T|O) = O^T(1-O)^{1-T} \tag{3.8}$$

As a result, the ML equation $\min_w -\log P(T|O)$ is equal to:

$$\min_w \mathcal{E} = \min_w -[T\log O + (1-T)\log(1-O)] \tag{3.9}$$

This is equivalent to minimizing the relative entropy or Kullback-Leibler (KL) divergence between the distributions T and O which is given by:

$$KL(T,O) = T\log T + (1-T)\log(1-T) - T\log O - (1-T)\log(1-O) \tag{3.10}$$

In the case of binary classification, since the output O is interpreted as a probability, it is natural to use a logistic unit with $O = f(S) = 1/(1+e^{-S})$. As a result:

$$\frac{\partial \mathcal{E}}{\partial S} = \frac{\partial \mathcal{E}}{\partial O}\frac{\partial O}{\partial S} = -(T-O) \tag{3.11}$$

3.1.3 k-Classification

In the case of classification into k-classes, the straightforward generalization is to use a multinomial model, so that:

$$P(T|O) = \prod_{i=1}^{k} O_i^{T_i} \tag{3.12}$$

As a result, the ML equation $\min_w -\log P(T|O)$ is given by:

$$\min_w \mathcal{E} = \min_w -\left[\sum_{i=1}^{k} T_i \log O_i\right] \tag{3.13}$$

This is equivalent to minimizing the relative entropy or Kullback-Leibler (KL) divergence between the distributions T and O which is given by:

Table 3.1: Summary table for shallow supervised learning for the problems of regression, classification into two classes, and classification into k
classes. The corresponding probabilistic models yield the corresponding error functions associated with the negative log likelihood of the data given the weight parameters. Together with the sensible choice of transfer function f, this leads to simple and identical error derivatives for learning.

Problem	Prob. Model	Error $\mathcal{E}$	Unit	$\partial\mathcal{E}/\partial S$
Reg.	Gaussian	$(T-O)^2/2$ (Quadratic)	Linear	$-(T-O)$
2-Class.	Binomial	$-T\log O-(1-T)\log(1-O)$ (KL)	Logistic	$-(T-O)$
k-Class.	Multinomial	$-\sum_{i=1}^k T_i\log O_i$ (KL)	Softmax	$-(T-O)$

$$KL(T,O)=\sum_{i=1}^k T_i\log T_i-\sum_{i=1}^k T_i\log O_i \tag{3.14}$$

In the case of k-class classification, it is natural to use a softmax unit, which generalizes the logistic unit ($k=2$). With a softmax unit: $O_i=e^{S_i}/\sum_j e^{S_j}$, where for every i, $S_i=\sum_j w_{ij}I_j$. As a result, for every $i=1,\dots,k$ we have:

$$\frac{\partial\mathcal{E}}{\partial S_i}=\sum_j\frac{\partial\mathcal{E}}{\partial O_j}\frac{\partial O_j}{\partial S_i}=-(T_i-O_i) \tag{3.15}$$

after some algebra and using the formula for the derivatives from the previous chapter ($\partial O_i/\partial S_i=O_i(1-O_i)$ and $\partial O_j/\partial S_i=-O_iO_j$ for $j\neq i$).

Thus, in short, the theory dictates which error function and which transfer function to use in both regression and classification cases (Table 3.1). In regression as well as binary classification, the gradient descent learning equation for a single unit can be written as:

$$\Delta w_i=-\eta\frac{\partial\mathcal{E}}{\partial w_i}=-\eta\frac{\partial\mathcal{E}}{\partial S}\frac{\partial S}{\partial w_i}=\eta(T-O)I_i \tag{3.16}$$

and similarly in k-classification.

3.1.4 Prior Distributions and Regularization

The Bayesian framework allows one to put a prior distribution on the parameters w. Consider a single unit with weight vector $w=(w_i)$ with $0\leq i\leq n$ including the bias.

A standard approach is to assume uniform or Gaussian prior distributions on the synaptic weights. For instance, in the case of a zero-mean, spherical Gaussian prior distribution (i.e the product of n independent, identical, one-dimensional Gaussian distributions):

$$P(w) = \frac{1}{\sqrt{(2\pi)^n \sigma^{2n}}} e^{-\sum w_i^2/2\sigma^2} \tag{3.17}$$

In the MAP optimization approach, this adds a term of the form $\sum_i w_i^2/2\sigma^2$ to the error function and the minimization process. The variance σ^2 determines the relative importance between the terms derived from the likelihood and the prior. Everything else being equal, the larger the value of σ the smaller the influence of the prior. The influence of the prior is to prevent the weights from becoming too large during learning, since large weights incur a large penalty $\sum_i w_i^2$. From an optimization framework, adding terms to the function being optimized to constrain the solutions is called regularization. In this sense, there is an equivalence between using priors and using regularizing terms. The Gaussian prior leads to a quadratic regularizer or an L2 penalty term. From a learning standpoint, in the case of gradient descent, the presence of a Gaussian prior adds a term $-w_i/\sigma^2$ to the gradient descent learning rule for w_i. This is also called "weight decay" in the literature.

Of course other priors or regularizing terms can be applied to the weights. Another regularization that is often used, instead of or in combination with L2 regularization, is the L1 regularization which adds a term of the form: $\lambda \sum_i |w_i|$ to the error function. More generally, one can define Lp regularization for any $p \geq 0$ based on Lp norms. Other prior distributions or regularization functions can be used in specific cases, for instance in the case where the weights are constrained to have binary values or, more generally, to be limited to a finite set of possible values.

L1 regularization tends to produce sparse solutions where, depending on the strength of the regularizer, a subset of the weights are equal to 0. This can be desirable in some situations, for instance to increase interpretability. Within sparse Bayesian priors alone, L1 is just one of many approaches. The L1 approach was developed early in [566] in relation to geology applications. It was further developed and publicized under the name of LASSO (least absolute shrinkage and selection operator) [627] (see also [628]). Another example of continuous "shrinkage" prior centered at zero is the horseshoe prior [164, 165]. However technically these continuous priors do not have a mass at zero. Thus another alternative direction is to use discrete mixtures [454, 276] where the prior on each weight w_i consists of a mixture of a point mass at $w_i = 0$ with an absolutely continuous distribution.

While priors are useful and can help prevent overfitting, for instance in situations where the amount of training data is limited, the real question is whether a full

Bayesian treatment is possible or not. For instance, for prediction purposes, a full Bayesian treatment requires integrating predictions over the posterior distribution. The question then becomes whether this integration process can be carried in exact or approximate form, and how computationally expensive it is. This may not be a problem for single units; but for large networks, in general a full Bayesian approach cannot be carried analytically. In this case, approximations including Markov Chain Monte Carlo methods become necessary. Even so, for large networks, full Bayesian treatments remain challenging and methods based on point-estimates are used instead. The general area at the intersection of Bayesian methods and neural networks continues to be an active research area (e.g.[425, 426, 473, 401]).

3.1.5 Probabilistic Neural Networks

So far we have described feedforward neural networks as being completely deterministic in how they operate: for a given input they always produce the same output in a deterministic way. Whenever desirable, it is easy to create probabilistic neural networks where the input-output relationship is not deterministic, but rather governed by a joint probability distribution determined by the weights of the networks and possibly a few other noise or sampling parameters. Stochasticity can be introduced in different layers of the architecture.

Stochasticity in the input layer is obtained by sampling. If the input is given by the vector $I = (I_1, \dots, I_n)$, one can interpret each component as being the mean of a normal distribution with standard deviation σ (or any other relevant distribution) and sample accordingly to produce a new input vector $I' = (I_1 + \eta_1, \dots, I_n + \eta_n)$ which is then fed to the network, in lieu of I. In this case, the noise terms η_i are sampled from a normal distribution $\mathcal{N}(0, \sigma^2)$.

Likewise, stochasticity in the output, or any hidden layer, is obtained by interpreting the neurons activities as parameters of a distribution and then sampling from the corresponding distribution (see also [100]). For instance, in the case of a linear unit, its activity can be interpreted as the mean of a normal distribution, and a sample from that normal distribution can be used as the stochastic output. The standard deviation of the normal distribution is either an external parameter, or an output computed by a different unit. In the case of a logistic unit, its output can be interpreted as a Bernoulli probability p which can be sampled, producing a stochastic output equal to 1 with probability p, and 0 with probability $q = 1 - p$. Introducing such stochasticity in the hidden layer is the key ingredient behind, for instance, variational autoencoders [368].

Other forms of stochasticity can be obtained by adding other forms of noise to the units, or to the connections, as is done for instance during the application of the dropout algorithm [605, 91] during learning. This algorithm is studied in a later chapter.

3.1.6 Independence of Units During Learning in Shallow Networks

Consider a feedforward shallow network with n_0 inputs and n_1 output units. Even if the units see the same inputs, they operate independently of each other in the sense that the output of any unit does not depend on the output of any other unit. Unless there is a specific mechanism that couples the units during learning, the units learn independently of each other. In particular, if learning is based on minimizing an error function of the form $\mathcal{E} = \sum_{i=1}^{n_1} \mathcal{E}_i$ where $\mathcal{E}_i$ depends on O_i (and possibly a corresponding target T_i in the supervised case) but not on O_j for $j \neq i$, then each unit will learn independently of all the other units. This result, which of course is not true for deep networks, *implies that in shallow networks it is enough to understand the general behavior of one unit in order to understand the behavior of the entire network*. It must be noted that this result is very general and not tied to the existence of an error function. As we shall see in the chapter on local learning, *it is sufficient that the learning rule be local for the units to learn independently of each other in a network with a single adaptive layer.*

Now that we have addressed how to design single layer architectures we can turn to questions of capacity.

3.2 Capacity of Shallow Networks

3.2.1 Functional Capacity

Because of the independence of the units in a single layer network, it is sufficient to understand the capacity of a single unit. If the activation is linear and the transfer function is the identity (linear unit) then the unit implements a simple linear (or affine) function of the inputs. If the transfer function is the sgn or Heaviside function, or even a sigmoidal function, then we can still visualize the operation of the neuron as being determined by a hyperplane dividing $\mathbb{R}^n$ or H^n into two regions, with the value of the output being +1 on one side of the hyperplane–or growing towards +1 in the sigmoidal case–and conversely on the opposite side, replacing +1 with 0 or -1, depending on the exact transfer function being used. A similar picture is obtained if the activation is ReLU, or even polynomial by replacing the hyperplane by a polynomial surface. Thus, at least in the shallow case, it is possible to get a fairly clear mental picture of the class of functions that can be implemented.

In order to be more quantitative, next we focus primarily on the cardinal capacity of single linear and polynomial Boolean threshold gates. Linear and polynomial threshold functions have been extensively used and studied in complexity theory, machine learning, and network theory; see, for instance, [63, 66, 64, 154, 149, 562, 379, 587, 48, 115, 19, 372, 373, 226, 485, 486, 355] An introduction to polynomial threshold functions can be found in [484, Chapter 5], [39, Chapter 4], and [562]. In the Boolean setting, we know that there are 2^{2^n} Boolean functions of n variables. Some of them can be implemented as linear threshold gates, and some cannot. Thus we wish to estimate the fraction of such Boolean functions that can be realized by linear threshold gates. And similarly, the fraction can be realized by polynomial threshold gates of degree d.

3.2.2 The Capacity of Linear Threshold Gates

As an easy exercise, one can see that a number of well known Boolean functions of n variables are linearly separable and computable by a single linear threshold gate. For instance (see Figure 3.1), using a $\{-1, +1\}$ encoding of the input variables:

- AND, OR, NOT: The basic Boolean operators are all linearly separable. AND can be implemented using all weights equal to $+1$ and threshold equal to $n-1$. OR can be implemented using all weights equal to $+1$ and threshold $-(n-1)$. NOT can be implemented by a linear homogeneous threshold gate with a single weight equal to -1 and threshold 0.

- GEQ_K, LEQ_K (and MAJORITY as a special case): These functions compute whether the total number of $+1$ in the input is larger or smaller than a certain value K. Again they can be implemented using all weights equal to 1 and selecting the appropriate threshold.

- SELECT_k: This is the Boolean function that is equal to the $k-th$ component of its input. It can be implemented using $w_k = 1$, $w_i = 0$ for all other weights, and threshold 0.

- SINGLE_u: This is the Boolean function that is equal to $+1$ on a single given vertex $u = (u_1, \ldots, u_n)$ of the hypercube, and -1 on all other vertices. It can be implemented by a linear threshold gate with $w_i = u_i$ and threshold $n - 1$.

However there are also many functions that are not linearly separable. For instance:

- PARITY: This function takes the value +1 if the total number of +1 component in the input is even, and -1 otherwise.

- XOR: This function takes the value +1 only for input vectors that contain exactly one +1 component.

- CONNECTED: This function takes the value +1 if all the +1 component of the input are "together", with or without wrap around. For instance, with $n = 5$, $(+1, +1, +1, -1, -1)$ should be mapped to +1, whereas $(+1, -1, +1, -1, +1)$ should me mapped to -1.

- PAIR: Fix two vertices on the hypercube and consider the Boolean function that takes the value +1 on those two vertices, and -1 on all the remaining vertices. For the vast majority of such pairs, the corresponding functions is *not* linearly separable.

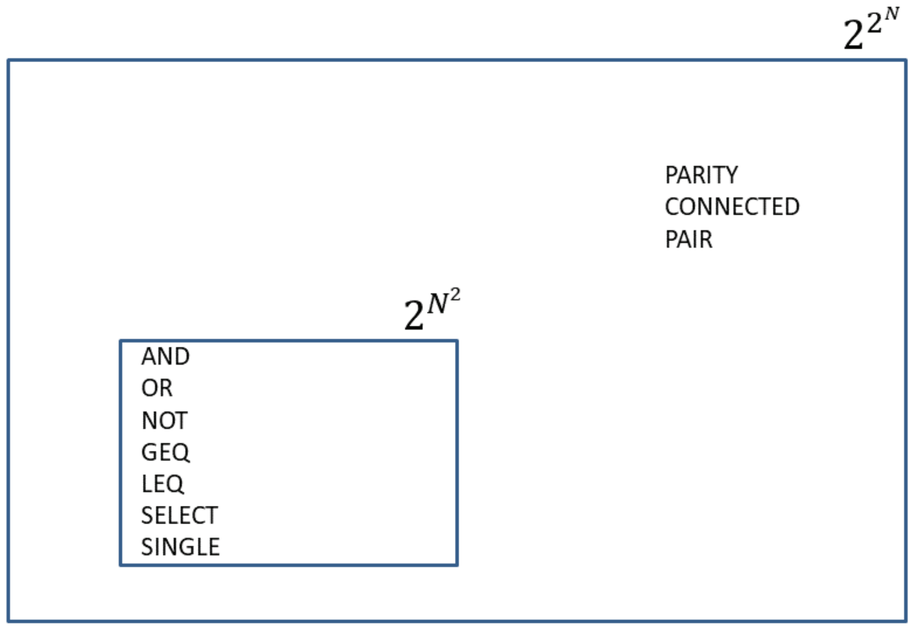

Figure 3.1: Larger box: Set of all Boolean functions of N variables. Smaller box: Subset of all Linear Threshold Boolean functions of N variables.

Thus what we would like to understand is what is the fraction of Boolean functions that can be implemented by linear threshold functions. Equivalently, a Boolean function can be viewed as a coloring of the vertices of the hypercube using two colors: blue and red. We want to estimate the fraction of colorings that are linearly separable.

Recall that in this context, the capacity $C(n,1)$ is the logarithm base two of the number of such colorings. Estimating $C(n,1)$ is a fundamental problem in the theory of neural networks and it has a relatively long history [39]. Next, we review the main results leaving the proofs as exercises (with additional complements in the references). To state the results, we will use the standard little o and big O notation. As a reminder, given two real-valued functions f and g defined over $\mathbb{R}$ or $\mathbb{N}$, we write $f(x) = O(g(x))$ when $x \to +\infty$ if and only if there exists a constant C and a value x_0 such that: $|f(x)| \leq C|g(x)|$ for all $x \geq x_0$. Similarly, we write $f(x) = o(g(x))$ for $x = a$ if and only if $\lim_{x\to a} f(x)/g(x) = 0$ (a can be finite or infinite).

The upper bound:

$$C(n,1) \leq n^2 \tag{3.18}$$

for $n > 1$, has been known since the 1960s (e.g. [204] and references therein). Likewise lower bounds of the form:

$$\alpha n^2 \leq C(n,1) \tag{3.19}$$

for $n > 1$ with $\alpha < 1$ were also derived in the 1960s. For instance, Muroga proved a lower bound of $n(n-1)/2$ (e.g. [463]), leaving open the question on the correct value of α. The problem of determining the right order was finally settled by Zuev [708, 709] who proved that:

Theorem 1. *The capacity of linear threshold functions satisfies:*

$$C(n,1) = n^2\,(1 + o(1)) \tag{3.20}$$

as $n \to \infty$.

More precisely, Zuev provided a lower bound of

$$\Big(1 - \frac{10}{\log n}\Big) \cdot n^2 \leq \log_2 T(n,1) \leq n^2. \tag{3.21}$$

J. Kahn, J. Komlós, E. Szemerédi [353, Section 4] further improved this result to:

$$C(n,1) = n^2 - n\log_2 n \pm O(n) \tag{3.22}$$

Thus in short the capacity of a linear threshold gate is approximately n^2, as opposed to 2^n for the total number of Boolean functions. Zuev's result can be derived from a combination of two results, one in enumerative combinatorics and the other in probability. The combinatorial result is a consequence of Zaslavsky's formula for hyperplane arrangements [700], and the probabilistic result is Odlyzko's theorem on spans of random ± 1 vectors [483]. Odlyzko's theorem, in turn, is based on a result on singularity of random matrices, namely that random matrices with ± 1 entries have full rank with high probability (see also [353, 657]).

Intuitively, Zuev's result is easy to understand from an information theoretic point of view as it says that a linear threshold gate is fully specified by providing n^2 bits, corresponding to n examples of size n. For instance, these can be the n support vectors, i.e. the n points closest to the separating hyperplane, taken from the largest class (there is one additional bit required to specify the largest class but this is irrelevant for n large).

Finally, it should be clear that $C(n,m) \approx mn^2$. This is simply because the capacity of an $A(n,m)$ architecture of linear threshold gates is equal to the sum of the capacities of each gate, due to their independence.

3.2.3 The Capacity of Polynomial Threshold Gates

The capacity increases if we use separating polynomial hypersurfaces rather than hyperplanes. Any Boolean function of n variables can be expressed as a polynomial of degree at most n. To see this, just write the function f in conjunctive (or disjunctive) normal form, or take the Fourier transform of f. A conjecture of J. Aspnes *et al.* [48] and C. Wang and A. Williams [667] states that, for most Boolean functions $f(x)$, the lowest degree of $p(x)$ such that $f(x) = \mathrm{sgn}(p(x))$ is either $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$. M. Anthony [38] and independently N. Alon (see [562]) proved one half of this conjecture, showing that for most Boolean functions the lower degree of $p(x)$ is at least $\lceil n/2 \rceil$. The other half of the conjecture was settled, in an approximate sense (up to additive logarithmic terms), by R. O'Donnell and R. A. Servedio [485] who gave an upper bound $n/2 + O(\sqrt{n \log n})$ on the degree of $p(x)$.

However here we are more interested in low degree polynomial threshold functions. While low degree polynomial threshold functions may be relatively rare within the space of Boolean functions, they are of particular interest both theoretically and practically, due to their functional simplicity and their potential applications in biological modeling and neural network applications. Thus the most important question is: How many low-degree polynomial threshold functions are there? Equivalently, how many different ways are there to partition the Boolean cube by polynomial surfaces of low degree? Equivalently how many bits can effectively be stored in the coefficients of a polynomial threshold function? In short, we want to estimate the cardinal capacity $C_d(n,1)$ of polynomial threshold gates of n variables of degree d, for fixed degree $d > 1$, as well as slowly increasing values of d. We provide the solution below. The details of the proof can be found in the references (see also exercises).

The history of the solution of this problem parallels in many ways the history of the solution for the case of $d = 1$. An upper bound $C_d(n,1) \leq n^{d+1}/d!$ was shown in [66], see also [39]. A lower bound $\binom{n}{d+1} \leq C_d(n,1)$ was derived in [562]. This lower bounds is approximately $n^{d+1}/(d+1)!$ which leaves a multiplicative gap $O(d)$ between the the upper and lower bounds. The problem was settled in [107] showing the following theorem, which contains Zuev's result as a special case.

Theorem 2. *For any positive integers n and d such that $1 \leq d \leq n^{0.9}$, the capacity of Boolean polynomial threshold functions of n variables and degree d satisfies*[1]

$$\left(1 - \frac{C}{\log n}\right)^d \cdot n\binom{n}{\leq d} \leq C_d(n,1) \leq n\binom{n}{\leq d}$$

In this theorem C denotes a positive absolute constant; its value does not depend on n or d. The exponent 0.9 in the constraint on d can be replaced by any constant strictly less than 1 at the cost of changing the absolute constant C. The upper bound in Theorem 2 holds for all $1 \leq d \leq n$; it can be derived from counting regions in hyperplane arrangements. The lower bound in Theorem 2 uses results on random tensors and Reed-Muller codes [7].

For small degrees d, namely for $d = o(\log n)$, the factor $(1 - C/\log n)^d$ becomes $1 - o(1)$ and Theorem 2 yields in this case the asymptotically tight bound on the capacity:

$$C_d(n,1) = n\binom{n}{\leq d}(1 - o(1)) \tag{3.23}$$

To better understand this bound, note that a general polynomial of degree d has $\binom{n}{\leq d}$ monomial terms. Thus, to communicate a polynomial threshold function, one needs

[1]Here and in the rest of the book, $\binom{n}{\leq d}$ denotes the binomial sum up to term d, i.e. $\binom{n}{\leq d} = \binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{d}$.

to spend approximately n bits per monomial term. During learning, approximately n bits can be stored per monomial term.

In some situations, it may be desirable to have a simpler estimate of $C_d(n,1)$ that is free of binomial sums. For this purpose, we can simplify the conclusion of Theorem 2 and state it as follows:

Theorem 3. *For any integers n and d such that $n > 1$ and $1 \leq d \leq n^{0.9}$, the number of Boolean polynomial threshold functions $T(n,d)$ satisfies:*

$$\Big(1 - \frac{C}{\log n}\Big)^d \cdot \frac{n^{d+1}}{d!} < \log_2 T(n,d) < \frac{n^{d+1}}{d!}$$

The upper bound in Theorem 3 actually holds for all $n > 1$, $1 \leq d \leq n$. For small degrees d, namely for $d = o(\log n)$, the factor $(1 - C/\log n)^d$ becomes $1 - o(1)$ and Theorem 3 yields in this case the asymptotically tight bound on the capacity:

$$C_d(n,1) = \frac{n^{d+1}}{d!}(1 - o(1)) \tag{3.24}$$

In summary, polynomial threshold functions of degree d in n variables provide a simple way to stratify all Boolean functions of these variables (Figure 3.2). In order to specify a polynomial threshold function in n variables and with degree d, one needs approximately $n^{d+1}/d!$ bits. This corresponds to providing the $n^d/d!$ support vectors on the hypercube that are closest to the separating polynomial surface of degree d in the largest class. Equivalently, there are approximately $2^{n^{d+1}/d!}$ different ways to separate the points of the Boolean cube $\{-1,1\}^n$ into two classes by a polynomial surface of degree d, i.e. the zero set of a polynomial of degree d.

3.2.4 The Capacity of Other Units

It is possible to consider other models and compute their capacity (see exercises). For instance, the capacity of linear threshold gates with binary $\{-1,+1\}$ weights $C_B(n,1)$ is linear rather than quadratic:

$$C_B(n,1) \approx n \tag{3.25}$$

In contrast, the capacity of linear threshold gates with positive weights $C_P(n,1)$ remains quadratic:

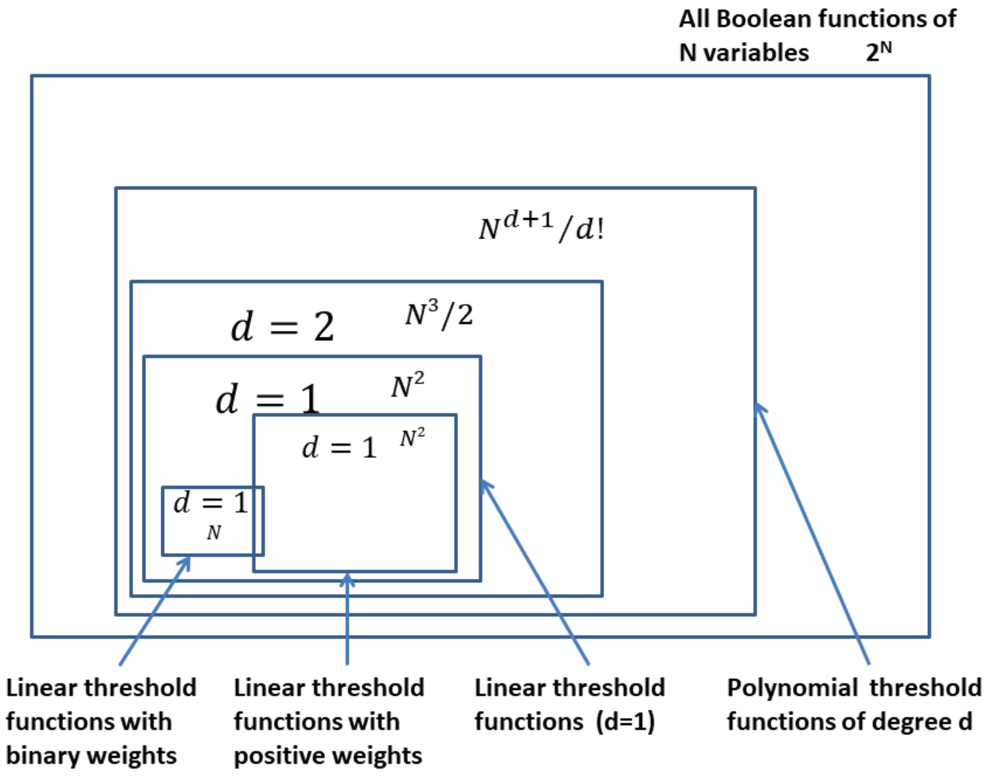
All Boolean functions of N variables
2^N
$N^{d+1}/d!$
$d = 2$
$N^3/2$
$d = 1$
N^2
$d = 1$
N^2
$d = 1$
N
Linear threshold functions with binary weights
Linear threshold functions with positive weights
Linear threshold functions (d=1)
Polynomial threshold functions of degree d

Figure 3.2: Stratified capacity of different classes of Boolean functions of N variables. Linear threshold functions with binary weights have capacity N. Linear threshold functions with positive weights have capacity $N^2 - N$. Linear threshold functions have capacity N^2. Polynomial threshold functions of degree 2 have capacity $N^3/2$. More generally, polynomial threshold functions of degree d have capacity $N^{d+1}/d!$ (fixed or slowly growing d). All these results are up to a multiplicative factor of $(1 + o(1))$. The set of all Boolean functions has capacity exactly equal to 2^N.

$$C_P(n,1) = n^2\,(1 + o(1)) \tag{3.26}$$

Finally, to study the capacity of a ReLU gate, one can imagine concatenating it with a linear threshold gate to produce a binary output. If we consider an $A(n,1,1)$ architecture where the hidden unit and the output unit are linear threshold gates, then it is easy to show that the capacity is unchanged, that is:

$$C(n,1,1) = C(n,1) \tag{3.27}$$

If the hidden units is a ReLU, then the capacity increases but still remains quadratic. Thus, with this definition of the capacity $C_{\mathrm{ReLU}}(n, 1)$ of a ReLU unit, one has:

$$C_{ReLU}(n, 1) = n^2 \left(1 + o(1)\right) \tag{3.28}$$

The same approach can be used for other gates with continuous output. As a side note, with this approach, the capacity of a linear unit–obtained by concatenating a linear unit with a threshold unit–is exactly equal to the capacity of a linear threshold gate.

3.3 Shallow Learning

3.3.1 Gradient Descent

One of the most important learning algorithms when there is a well-defined error function $\mathcal{E}$ is gradient descent, which can generally be written as:

$$\Delta w_{ij} = -\eta \frac{\partial \mathcal{E}}{w_{ij}} \tag{3.29}$$

From the Bayesian statistical framework, we have seen that in the three models for regression, 2-classification, and k-classification the gradient $\partial\mathcal{E}/\partial S$ is the same and equal to $-(T - O)$. Since $\partial S/\partial w_{ij} = I_j$, the gradient descent learning equation for all three cases is given by:

$$\Delta w_j = \eta(T - O)I_j \quad \text{or} \quad \Delta w_{ij} = \eta(T_i - O_i)I_j \quad (\mathrm{k-classification}) \tag{3.30}$$

If we consider an entire output layer of linear units (regression), or logistic units (multiple 2-classifications), then the learning equations for all three cases become identical. As usual, these equations are for a single example and must be summed or averaged over the entire training set (or a minibatch) during any form of batch learning. For instance, in the case of linear regression or logistic regression, the learning equations become:

$$\Delta w_j = \eta \frac{1}{K} \sum_t \left(T(t) - O(t)\right) I_j(t) \tag{3.31}$$

where t runs over the batch or minibatch of size K.

Gradient descent learning in shallow networks is relatively easy to analyze due to convexity considerations [542, 140]. We are going to see that the error function in the linear and 2-classification cases are convex in the synaptic weights. We leave the k-classification case as an exercise. Thus, in general, gradient descent learning will converge to a global minimum of the error function in these shallow cases.

To study convexity, it is easy to see from the definition of convexity that a linear combination of convex functions with positive coefficients is also convex. Since the batch error functions are sums of individual-example error functions, it is sufficient to show that the error functions for each training example are themselves convex with respect to the synaptic weights. And in cases where, for each example, the error function is a sum of component-wise error functions (e.g. in vector regression), it is sufficient to deal with each individual component of the error for each example.
‘

3.3.2 The Linear Case

We leave it as an exercise to check that the error function is convex. The linear case, however, corresponds to linear regression and one can analytically derive an expression for the optimal weights. The critical equation is obtained by setting the gradient to zero. Using the matrix notation described in the previous chapter and the expectation symbol to denote averages over the training set, this yields:

$$E\left((T-O)I^t\right) = 0 \quad \text{or} \quad E(TI^t) - E(WII^t) = 0 \tag{3.32}$$

where W is the matrix of weights. We introduce the two covariance matrices of the data $E(TI^t) = \Sigma_{TI}$ and $E(II^t) = \Sigma_{II}$. Using these matrices, yields the equivalent form: $\Sigma_{TI} = W\Sigma_{II}$. Thus in the general case where Σ_{II} is invertible, the optimal weight matrix is given by:

$$W^* = \Sigma_{TI}\Sigma_{II}^{-1} \tag{3.33}$$

Note that the same equation applies regardless of whether the output layer contains a single or multiple units. With n input units (possibly including the clamped unit for the bias) and m output units, the matrix W is $m \times n$.

3.3.3 The Logistic Case

To prove convexity in the case of logistic regression with a single logistic unit, we simply show that the Hessian is semi-definite positive. Using the results obtained above it is easy to compute the second order derivatives, as follows:

$$\frac{\partial^2 \mathcal{E}}{\partial w_i^2} = \frac{\partial}{\partial w_i}\frac{\partial \mathcal{E}}{\partial w_i} = -\frac{\partial}{\partial w_i}(T-O)I_i = I_i\frac{\partial O}{\partial w_i} = I_i^2\frac{\partial O}{\partial S} \tag{3.34}$$

using $\partial\mathcal{E}/\partial w_i = -(T-O)I_i$ and $\partial O/\partial w_i = \partial O/\partial S \times \partial S/\partial w_i = \partial O/\partial S \times I_i$. Likewise,

$$\frac{\partial^2 \mathcal{E}}{\partial w_i w_j} = \frac{\partial}{\partial w_j}\frac{\partial \mathcal{E}}{\partial w_i} = -\frac{\partial}{\partial w_j}(T-O)I_i = I_i\frac{\partial O}{\partial w_j} = I_iI_j\frac{\partial O}{\partial S} \tag{3.35}$$

Thus the Hessian H is given by $\frac{\partial O}{\partial S}II^t$. The logistic function is monotone increasing ($\partial O/\partial S > 0$). Thus for any vector x of dimension n: $x^tHx = \frac{\partial O}{\partial S}x^tII^tx \geq 0$ and $\mathcal{E}$ is convex.

3.3.4 The Perceptron (Linear Threshold Function) Case

Here we consider a linear threshold function. As mentioned in the Introduction, the perceptron learning rule can be written as a gradient descent rule of the form $\Delta w_i = \eta(T-O)I_i$. However this is not how the rule is usually presented. To see the equivalence between the two forms, let us first notice that as long as the rule is also applied to the bias and all the weights are initialized to zero, then the learning rate is irrelevant. It only influences the scale of the weights and thus of the activation, but not the output of the perceptron. As a result, for simplicity, we can assume that $\eta = 0.5$. Next, note that the term $T-O$ is non-zero only when there is an error, and in this case it is equal to +2 or -2. Thus, we can write the more standard, but equivalent, form of the perceptron learning rule:

$$\Delta w = \begin{cases} I, & T=+1 \text{ and } O=-1 \\ -I, & T=-1 \text{ and } O=+1 \\ 0 & \textit{otherwise.} \end{cases} \tag{3.36}$$

which holds for any input I. The perceptron learning algorithm initializes the weight vector to zero $w(0) = 0$ and then at each step it selects an element of the training set that is mis-classified and applies the learning rule above. The perceptron learning theorem states that if the data is separable, then the perceptron algorithm will converge to a separating hyperplane in finite time. One may suspect that this may be the case because the rule amounts to applying stochastic gradient descent to a unit with a sigmoidal (logistic or tanh) transfer function, which is similar to a perceptron. In addition, the rule above clearly improves the performance on an example I that is mis-classified. For instance if the target of I is $+1$ and I is mis-classified and selected at step t, then we must have $w(t)\cdot I < 0$ and $w(t+1) = w(t)+I$. As a result, the performance of the perceptron on example I is improved since $w(t+1)\cdot I = w(t)\cdot I + ||I||^2$, and similarly for mis-classified examples that have a negative target. However none of these arguments is sufficient to give a definitive proof of convergence.

To prove convergence, consider a training set of the form $\{(I(k), T(k)\}$ for $k = 1, \ldots, K$. To say that it is linearly separable means that there exists a weight vector w^*, with $||w^*|| = 1$, such that $T(k)w^*\cdot I(k) > 0$ for every k, where "$\cdot$" denotes the dot product. Since the training set is finite, we can let γ correspond to the worst case margin, i.e. $\gamma = \min_k T(k)w^*\cdot I(k)$. Let also R be the radius of the data, i.e. $R = \max_k ||I(k)||$.

Theorem 4. *With separable data, the perceptron learning algorithm converges in at most R^2/γ^2 steps to a weight vector that separates the data (zero training error).*

Proof: The idea of the proof is simply to look at the angle between the current value of the weight vector $w(t)$ and w^* and how it evolves in time. Let us suppose that example $I(k) = I$ with target $T(k) = T$ is selected at step t. Then: $w(t+1) = w(t) + TI$ and thus:

$$w(t+1)\cdot w^* - w(t)\cdot w^* = (w(t)+TI)\cdot w^* - w(t)\cdot w^* = TI\cdot w^* \geq \gamma \tag{3.37}$$

Thus, since $w(0) = 0$, after n learning steps we have:

$$w(n)\cdot w^* \geq n\gamma \tag{3.38}$$

To estimate the angle, we also need an estimate of the size of w. Since $w(t+1) = w(t) + TI$, we have:

$$||w(t+1)||^2 = (w(t)+TI)^2 \leq ||w(t)||^2 + ||TI||^2 \leq ||w(t)||^2 + R^2 \tag{3.39}$$

since $wTI < 0$. Thus after n steps, we have:

$$||w(n)||^2 \leq nR^2 \tag{3.40}$$

Thus the cosine between $w(n)$ and w^* satisfies:

$$\cos(w(n), w^*) = \frac{w(n) \cdot w^*}{||w(n)||||w^*||} \geq \frac{n\gamma}{\sqrt{n}R} \tag{3.41}$$

and thus by the time n reaches the value R^2/γ^2 (or its floor) the cosine of the angle is equal or close to 1, thus the angle is equal or close to 0, and it is easy to check that all the training examples are correctly classified. It is also easy to see that the result remains true even if $w(0) \neq 0$, although in general the convergence may be slower. If the data is not linearly separable, one can show the so-called perceptron cycling theorem which states that with the perceptron learning algorithm the weight vector w remains bounded and does not diverge to infinity [127, 273], and likewise the number of errors is bounded [481].

3.3.5 Data Normalization, Weight Initializations, Learning Rates, and Noise

Even in the simple case of gradual learning in a single unit there are a number of important practical issues.

First, everything else being equal, there is no reason to favor a particular component of the input vector. Doing so may actually slow down learning. Thus it is useful to preprocess the training data by normalizing each component in the same way through some affine transformation. Typically this is done by subtracting the mean and rescaling the range of each component to some fixed interval, or normalizing the standard deviations to one.

Second, everything else being equal, there is no reason to favor any synaptic weight, and doing so may actually slow down learning. A large bias, for instance, is associated with a hyperplane that is likely to be far from the cloud of normalized data points and thus likely to perform poorly, requiring larger modifications during learning. Thus all the weights, including the bias, should be initialized to small values. This can be done efficiently by initializing each weight independently using a uniform or normal distribution with small standard deviation.

Third, there is the issue of the learning rate, or step size, which must be decreased as learning progresses in order to converge to the optimum. In simple convex cases, it is possible to estimate the time of convergence to the global minimum with a given rate schedule and derive effective rate schedules. In more complex and realistic deep learning cases, the schedules must be set in advance or adjusted somewhat empirically at training time as a function of the data.

And fourth, in tandem with the third point, gradient descent changes are often applied using mini-batches. The size of the mini-batches is one way to control the degree of stochasticity of the learning process. In the case of a single unit, or a shallow layer, we have seen that the error functions are typically convex thus the use of stochastic gradient descent is less critical than for deep networks. A classical result due to Robbins and Monro [541] is that, in the convex case, following noisy gradients with a decreasing step size provably reaches the optimum.

The true gradient is a sum of many terms, the gradients computed on each example. Thus stochastic gradient can be viewed as a drawing from a Gaussian random variable with mean equal to the true gradient, and standard deviation inversely proportional to the square root of the batch size. The larger the mini-batches, the closer the gradient estimate is to the true gradient. In this way, stochastic gradient descent can be modeled as a Ornstein-Uhlenbeck Brownian motion process [636] leading to a stochastic differential equation [488, 434].

3.4 Extensions of Shallow Learning

In this chapter we have studied the design and learning algorithms for shallow $A(n, 1)$ or $A(n, m)$ architectures. However, the same ideas can be applied immediately to a number of other settings. In a later chapter, we will consider an example of shallow recurrent network called the Hopfield model, consisting of n symmetrically connected linear threshold gates, which are all visible. This enables the implementation of shallow learning using the simple Hebb's rule. Here we consider other examples within the class of layered feedforward networks.

3.4.1 Top Layer of Deep Architectures

In this chapter, we have seen how to design a shallow network for regression or classification and derived a gradient descent learning rule for it. However the design solution is far more general and can be applied to the design of the top layer of *any* deep feedforward architecture for regression or classification purposes. Thus the general design problem for deep forward architectures is basically solved for the output layer. Furthermore, if all the weights in the lower layers are frozen, the same learning algorithm can be applied. Indeed, freezing the weights of the lower layers simply provides a transformation of the original input vectors into a new set of input vectors for the top layer. Thus logically the next level of architectural complexity to be explored is provided by feedforward architectures $A(n, m, p)$ with two layers of weights, but where the lower layer weights are fixed and only the upper layer is adaptive. This actually happens in two well-known cases: extreme machines where the lower weights are random, and support vector machines where the lower weights are equal to the training examples.

3.4.2 Extreme Machines

In the so-called extreme machines [330, 158], the weights in the lower layer are chosen at random and thus provide a random transformation of the input data. In the case where the hidden layer consists of linear units and is smaller in size than the input layer ($n > m$), this can be justified by the the Johnson-Lindenstrauss Lemma and other related theorems [348, 258, 215]. The basic idea implemented by this first layer is the idea of dimensionality reduction with little distortion. The Johnson-Lindenstrauss Lemma states that a properly-scaled random projection from a high-dimensional space to a lower-dimensional space tends to maintain pairwise distances between data points. More precisely:

Lemma 1. *(Johnson-Lindenstrauss) Given $0 < \epsilon < 1$, a set X of K points in $\mathbb{R}^n$, and a number $m > 8\ln(K)/\epsilon^2$, there is a linear map $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ such that:*

$$(1-\epsilon)||u-v||^2 \leq ||f(u)-f(v)||^2 \leq (1+\epsilon)||u-v||^2 \tag{3.42}$$

for all $u, v \in X$. f can be a suitably scaled orthogonal projection onto a random subspace of dimension m.

Thus the first random layer reduces the dimensionality of the data without introducing too much distortion. The following layer can address classification or regression problems based on the more compact representation generated in the hidden layer. The theory derived in this chapter (e.g. design, learning algorithm, and learning convergence) applies immediately to the top layer of this case without any changes. Note that this remains true even if non-linear transfer functions are used in the hidden layer, as long as the random projection weights remain fixed. In some cases, expansive transformations where $m > n$ can also be considered, for instance in order to encode input vectors into sparse hidden vectors (see [57] for related work).

3.4.3 Support Vector Machines

From an architectural standpoint, in support vector machines (SVMs) [202, 208], the units in the hidden layer are also linear as in extreme machines. The weights of these hidden units are identical to the vectors in the training set. Thus each hidden unit is associated with one training example and its activation is equal to the dot product between the input vector and the corresponding training example. More precisely, consider the binary classification problem with input vectors $I(1), \dots, I(K)$ with corresponding binary targets $T(1), \dots, T(K)$. It can be shown that the separating hyperplane with maximal margin can be written in the activation form:

$$S(I) = \sum_{i=1}^{K} w_i T(i)(I \cdot I_i) + w_0 = \sum_{i=1}^{K} w_i T(i)(I^t I_i) + w_0 \tag{3.43}$$

While gradient descent can be applied to learning the weight vector w, SVMs come with their own optimization learning algorithm. In the optimal solution, the vectors associated with non-zero weights are the support vectors. This approach can be generalized to kernel machines where a similarity kernel $K(I, I_j)$ is used to compute the similarity between I and I_j in lieu of the dot product [577].

3.5 Exercises

3.1 Derive the prior distribution associated with L1 regularization. Extend the analysis to Lp regularization.

3.2 What happens in linear regression when the matrix Σ_{II} is not invertible?

3.3 Estimate the expectation of the error that a trained linear unit makes on a new data point, i.e. a data point not in the training set (generalization error)? How does this expectation depend on the size of the training set. Clarify all your assumptions.

3.4 (1) Prove that the quadratic error function in linear regression is convex with respect to the weights. (2) Consider k-classification with softmax output units satisfying $O_i = e^{S_i} / \sum_j e^{S_j}$. Is the relative entropy error term associated with O_i convex with respect to the weights w_{ij} associated with S_i? Is the relative entropy error term associated with O_i convex with respect to the other weights w_{kl}, i.e. the weights associated with the other activations S_k, $k \neq i$)?

3.5 Study the evolution of the weight vector during learning in a single linear unit trained using the simple Hebbian learning rule $\Delta w_i = \eta O I_i$.

3.6 Prove that any Boolean functions of n variables can be written as a polynomial threshold function of degree n.

3.7 Let $C_d(n,1)$ be the logarithm base 2 of the number of Boolean polynomial threshold functions of n variables of degree d, and $C_d^*(n,1)$ be the number of Boolean homogeneous (over $H^n = \{-1,1\}^n$) polynomial threshold functions of n variables of degree d. Prove the following relationships (the degree d is omitted when $d = 1$):

$$C(n,1) = C^*(n+1,1) \tag{3.44}$$

$$C^*_{d-1}(n,1) < C^*_d(n,1) < C^*(\binom{n}{d},1) \quad \text{for} \quad d \geq 2 \tag{3.45}$$

$$C_{d-1}(n,1) < C_d(n,1) < C([\binom{n}{0} + \binom{n}{1} + \ldots \binom{n}{d},1) \quad \text{for} \quad d \geq 2 \tag{3.46}$$

3.8 Hyperplane arrangements. 1) Prove that the number $K(m,n)$ of connected regions created by m hyperplanes in $\mathbb{R}^n$ (passing through the origin) satisfies:

$$K(m,n) \leq 2\sum_{k=0}^{n-1} \binom{m-1}{k} = 2\binom{m-1}{\leq n-1}$$

2) Prove that this bound becomes an equality if the normal vectors to the hyperplanes are in general position. 3) Prove that the same bound and equality condition remain true for m central hyperplanes (i.e. affine hyperplanes passing through the same point). 4) Prove that the number $L(m,n)$ of connected regions created by m affine hyperplanes in $\mathbb{R}^n$ satisfies:

$$L(m,n) \leq \sum_{k=0}^{n} \binom{m}{k} = \binom{m}{\leq n}$$

5) Prove that $L(m,n)$ is bounded below by the number of all intersection subspaces defined by the hyperplanes. [An intersection subspace refers to the intersection of

any subfamily of the original set of hyperplanes. The dimensions of an intersection subspace may range from zero (a single point) to n (intersecting an empty set of hyperplanes gives the entire space $\mathbb{R}^n$)]. 6) Prove that if the hyperplanes are in general position, then the upper bound and the lower bound are the same and each bound becomes and equality. These results are needed for the next few exercises.

3.9 To see the relevance of hyperplane arrangements, let us fix a finite subset $S \subset \mathbb{R}^n \setminus \{0\}$ and consider all homogeneous linear threshold functions on S, i.e. functions $f : S \to \mathbb{R}$ of the form

$$f_a(x) = \mathrm{sgn}(a \cdot x)$$

where $a \in \mathbb{R}^n$ is a fixed vector. Consider the collection ("arrangement") of hyperplanes

$$\{x^\perp : \ x \in S\},$$

where $x^\perp = \{z \in \mathbb{R}^n : \ z \cdot x = 0\}$ is the hyperplane through the origin with normal vector x. Two vectors a and b define the same homogeneous linear threshold function $f_a = f_b$ if and only if a and b lie on the same side of each of these hyperplanes. In other words, $f_a = f_b$ if and only if a and b lie in the same open component of the partition of $\mathbb{R}^n$ created by the hyperplanes $x^\perp$, with $x \in S$. Such open components are called the *regions* of the hyperplane arrangement. Prove that: the number of homogeneous linear threshold functions on a given finite set $S \subset \mathbb{R}^n \setminus \{0\}$ equals the number of regions of the hyperplane arrangement $\{x^\perp : \ x \in S\}$.

3.10 Prove the following theorem originally obtained by Wendel [678].

Theorem 5. *Let K points in $\mathbb{R}^n$ be drawn i.i.d from a centrosymmetric distribution such that the points are in general position. Then the probability that all the points fall in some half space is:*

$$P_{n,K} = 2^{-K+1}\binom{K-1}{\leq n-1} = 2^{-K+1}\sum_{i=0}^{n-1}\binom{K-1}{i} \tag{3.47}$$

3.11 Prove that for any $n > 1$, the following upper bound holds:

$$C(n,1) \leq n^2 \tag{3.48}$$

3.12 Prove that for any $n > 1$ and all degrees d such that $1 \leq d \leq n$, the following upper bound holds:

$$C_d(n,1) \leq n\binom{n}{\leq d} \tag{3.49}$$

3.13 Capacity of sets. The capacity $C(S)$ of a finite set $S \subset \mathbb{R}^n$ containing $|S|$ points is defined to be the logarithm base two of the number of possible ways the set can be split by linear threshold functions. Likewise, the capacity $C_d(S)$ of order d ($d > 1$) of S is defined to be the logarithm base two of the number of possible ways the set can be split by polynomial threshold functions of degree d. More generally, we can define the capacity $C(S, n_1, \ldots, n_L)$ to be the logarithm base 2 of the number of linear threshold functions that can be defined on the set S using a feedforward architecture $A(n_1, \ldots, n_L)$ of linear threshold function (with $n = n_1$). Obviously $C(S) = C(S, n, 1)$. Prove that this notion of set capacity satisifes the following properties:

1) Affine invariance: For any invertible affine transformation $F : \mathbb{R}^{n_1} \to \mathbb{R}^{n_1}$, we have:

$$C\big(F(S), n_1, n_2, \ldots, n_L\big) = C(S, n_1, n_2, \ldots, n_L)$$

2) Single layer: For any set $S \subset \mathbb{R}^n$:

$$C(S, n, m) = C(S)m$$

3) Replacement by image: For any set $S \subset \mathbb{R}^{n_1}$ and a threshold map from $\mathbb{R}^{n_1}$ to H^{n_2}, we have:

$$C\big(f(S), n_2, n_3, \ldots, n_L\big) \leq C(S, n_1, n_2, \ldots, n_L)$$

Derive lower and upper bounds on $C(S)$ and $C_d(S)$ when $S \in \mathbb{R}^n$ and when $S \in \{-1, +1\}^n$. In particular, show that for any such set in $\mathbb{R}^n$:

$$C(S) \leq 2\binom{|S|-1}{\leq n} \tag{3.50}$$

and:

$$1 + \log_2 |S| \leq C(S) \leq 1 + n \log_2 \left(\frac{e|S|}{n}\right)$$

where the lower bound is true as soon as $|S| > 2^{17}$. The upperbound can be simplified to $n \log_2 |S|$ as soon as $n > 4$. To prove this result you will need to prove the following inequality as an intermediate result:

$$\sum_{k=0}^{n} \binom{N}{k} \leq \left(\frac{eN}{n}\right)^n \tag{3.51}$$

which is valid for all integers $1 \leq n \leq N$. Show that the lower bound is attained by a particular choice of S. If S is a subset of the Boolean hypercube, the lower bound can be improved to:

$$\frac{1}{16}\log_2^2 |S| \leq C(S) \leq 1 + n\log_2\left(\frac{e|S|}{n}\right)$$

3.14 Prove a lower bound of the form $\alpha n^2 \leq C(n,1)$ for some positive constant α less than one. For example, use a recursive construction to show that the number of linear threshold functions of n variables is greater than $2^{n(n-1)/2}$.

3.15 Capacity with binary weights. Let $C_B(n,1)$ be the capacity of a linear threshold gate with n inputs with binary weights restricted to $\{-1,1\}$. Let $C_B^*(n,1)$ be the capacity of a linear homogeneous threshold gate (i.e. with no bias) with n inputs, with binary weights restricted to $\{-1,1\}$. Prove that for any n:

$$C_B^*(n,1) = n \quad \text{if } \ n \ \text{ is odd}$$
$$C_B(n,1) = n+1 \quad \text{if } \ n \ \text{ is even}$$

Show that if one extends the definition of a threshold function by arbitrarily deciding that $\text{sgn}(0) = +1$, then each formula above is true for every n.

Show that the capacity of polynomial threshold gates of degree $d > 1$ with binary weights satisfies

$$C_{B,d}(n,1) \leq \sum_{k=1}^{d} \binom{n}{k} = \binom{n}{\leq d} - 1$$

Derive a similar upperbound in the homogeneous case, where homogeneous is defined over H^n. Derive a similar upperbound in the homogeneous case, where homogeneous is defined over $\mathbb{R}^n$.

3.16 Capacity with positive weights. Let $C_P(n,1)$ be the capacity of a linear threshold gate with n inputs with weights restricted to be positive (≥ 0), and similarly $C_P^*(n,1)$ for linear threshold gate with n inputs but no bias. Prove that:

$$C_P^*(n,1) = \frac{C^*(n,1)}{2^n}$$

and:

$$C^*_P(n,1) \leq C_P(n,1) \leq C^*_P(n+1,1)$$

As a result, using Zuev's result:

$$C_P(n,1) = n^2\,(1+o(1))$$

In short, for $d=1$, when the synaptic weights are forced to be positive the capacity is still quadratic. Write a conjecture for the case $d>1$ corresponding to polynomial threshold gates of degree d with positive weights.

3.17 Estimate the capacity of an architecture $A(n,1,1)$ where the hidden unit is a ReLU unit, and the output unit is a linear threshold unit.

3.18 Study the perceptron algorithm when $w(0) \neq 0$. Prove that, in the linearly separable case, the algorithm is still convergent, but the convergence in general may be slower. Study the perceptron algorithm in the case of training data that is not linearly separable. Prove that in this case, the weight vector remains bounded and does not diverge to infinity.

3.19 Satisfiability of threshold gates. Consider m linear threshold gates $f_1,\ldots,f_m$ of n binary variables, i.e. the common input is in $\{-1,+1\}^n$, with binary weights restricted to the set $\{-1,+1\}$. These threshold functions are said to be satisfiable if there exists a vector x in $\{-1,+1\}^n$ such that $f_i(x)=1$ for $i=1,\ldots,m$. Is the satisfiability of threshold gates problem NP-complete?

3.20 Study by simulations and analytically, whenever possible, the behavior of stochastic gradient descent in shallow learning, including: the effect of data normalization, the effect weight initialization, the effect of the learning rate, the effect of noise (e.g. online versus batch learning), and the speed and accuracy of convergence to the global minimum. Begin with the simplest case of a single linear unit with a single weight acting as a multiplier (i.e. $O=wI$) and generalize to a linear unit with n weights. Then proceed with a logistic unit with a single weight (i.e. $O=\sigma(wI)$, σ is the logistic function) and then generalize to n weights.

3.21 Study by simulations and analytically, whenever possible, the learning behavior of a single linear or logistic unit with probabilistic output, i.e. where the output O' is sampled from the normal distribution with mean $O=\sum_i w_i I_i$ in the linear regression case, or from the Bernouilli distribution with parameter $p=O=\sigma(\sum_i w_i I_i)$ (σ is the logistic function) in the binary classification case. Start with a single weight and generalize to n weights. For learning, consider the quadratic (linear case) or relative entropy (classification case) error functions based on O, or on O', in combination with two learning rules: $\Delta w_i = \eta(T - O)I_i$ and $\Delta w_i = \eta(T - O')I_i$, for a total of four error/rule combinations. For each combination, examine both on-line and batch learning.

3.22 Show that there exists a differentiable transfer function such that the Boolean XOR function can be implemented by a single unit with linear activation. More broadly, show every Boolean function of n variables that is symmetric, i.e. invariant under any permutation of its input entries, there exists a differentiable transfer function such that the function can be implemented by a single unit with linear activation and identical weights. Even more broadly, show that for every Boolean function of n variables, there exists a differentiable transfer function and a set of weights (with a corresponding linear activation) that can realize the function exactly.

3.23 Prove the Johnson-Lindenstrauss Lemma.