Neural Dynamics With Stochasticity

Hava T. Siegelmann

Information Systems Engineering, Faculty of Industrial Engineering and Management
Technion, Haifa 32000 Israel. E-mail: iehava@ie.technion.ac.il

1 Introduction

Our interest is in computers called *artificial neural networks*. These consist of assemblies of simple processors, or "neurons", each of which computes a scalar activation function of its input. The scalar value produced by a neuron is, in turn, broadcast to the successive neurons involved in a given computation. Some of the signals originate from outside the network and act as inputs to the entire system, while some of the output signals are communicated back to the environment and are thus used to encode the end result of the computation. These networks can be thought of either as functional units or as reactive systems, as they are able to both approximate input-output mappings and adapt to new environments. The networks are thus of great use as automatic learning tools and as adaptive and optimal controllers, e.g. in applications to vision, speech processing, robotics, signal processing, and many other fields (see [16],[45],[27],[38]). Herein we perceive neural nets as abstract functional devices able to perform exact computations rather than approximations only.

To achieve a rich and uniform computational model one must allow the system to evolve for a flexible amount of time by incorporating memory into the computation, this is not always the case in the neural architectures. In broad terms, one may classify neural networks according to their architecture, into *feedforward* and *recurrent* (or *feedback*) nets. The former are arranged in multiple layers, in such a manner that the input of each layer is provided by the output of the preceding one. Thus, the interconnection graph is acyclic, and the response time to an external input cannot be greater than the number of layers, independent of the length of the input. Feedforward networks are useful for representation, interpolation, and approximation of functions and stationary time series (see, e.g., [4, 5, 7, 11, 19, 20, 26, 48, 47, 46]), but because their computation ends in a fixed number of steps, one such network cannot perform general computations for inputs of varying lengths. Feedback networks, in contrast, allow loops in their graphs, and thus memory of past events is possible; this property facilitates a more compact and general representation of time series (see, e.g. [46]). These networks can be considered a rich computational model.

Most of traditional work on the computational power and capabilities of recurrent neural networks has focused on networks of infinite size (e.g., [15, 12, 13, 17, 52, 30]). Because these models contain an unbounded number of neurons, however, they cannot really explain the true power of their networks.

C. Lee Giles and Marco Gori (Eds.): Adaptive Processing, LNAI 1387, pp. 346-369, 1998.

They provide not only infinite memory (a fair resource), but they become infinite automata. Perhaps it is more appropriate to consider computational models which include a finite number of neurons, but still allow for growing memory by allowing, for example, for growing precision in the neurons; this is the basis of the current work. The first computational model that combined a finite number of neurons and infinite precision was introduced by Pollack [37]. Each neuron in his model computed a second-order polynomial or the threshold function; his model did not allow for first-order polynomials, as is common in neural networks, nor did it allow for any continuous activation function. We rather view continuity as an important requirement for the modeling of analog physical systems with neural networks. (It is hard to imagine many natural systems where any two arbitrary close numbers can be well discerned, as they are by the threshold function.) We thus choose to build on the model described in the following subsection.

1.1 Deterministic Analog Recurrent Networks

Several recent papers have considered the nature of deterministic analog (continuous) recurrent neural networks (e.g., [41, 43, 42, 25, 22, 28, 14, 44]). These consist of a finite number of neurons. The *activation value*, or *state*, of each neuron is updated at times $t = 1, 2, 3, \ldots$, according to a function of the activations (x_j) and inputs (u_j) at time $t-1$, and a set of real coefficients —also called *weights*— (a_{ij}, b_{ij}, c_i). More precisely, each neuron's state is updated by an equation of the type

$$x_i(t+1) = \sigma\left(\sum_{j=1}^{N} a_{ij}\, x_j(t) + \sum_{j=1}^{M} b_{ij}\, u_j(t) + c_i\right) \quad , \quad i = 1, \ldots, N \qquad (1)$$

where N is the number of neurons, M is the number of external input signals, and σ is a "sigmoid-like" function. In the basic model, σ is a very simple sigmoid, called the saturated-linear function:

$$\sigma(x) := \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \le x \le 1 \\ 1 & \text{if } x > 1\ . \end{cases} \qquad (2)$$

A subset of p out of the N neurons (the *output neurons*) is used to communicate the outputs of the network to the environment. Thus a network is specified by the data (a_{ij}, b_{ij}, c_i), together with a subset of its neurons.

Note that this model is highly homogeneous: each neuron computes the composition of two simple functions: affine combination and the simple saturated-linear activation function. The networks are built up of a finite number of neurons, whose number *does not increase* with the length of the input. The structure of the network, including the values of the interconnection weights, does not change in time, but rather remains constant. What changes in time are the

activation values, or outputs of each neuron. In this sense our model is "uniform" in contrast with certain models used in the past (e.g., [17]) that allow the number of units to increase over time and often even allow the structure to change, depending on the length of inputs being presented.

The network is a *parametric model of computation*: altering its constitutive parameters allows the model to coincide with previous models which are computationally and conceptually different. In particular the computational power depends on the type of numbers utilized as weights. When the weights are integers, the network is a finite state machine only. When the weights are rational numbers, the network is equivalent (in power and time) to the Turing Machine model ([41, 43]). Finally, when the weights require infinite precision, the finite networks are proved to be stronger than the Turing model, and to compute under polynomial time constraints the class P/poly, to be described in the next section. (The class P/poly includes all P and, moreover, some of EXP and even a few non-recursive functions, e.g. a unary encoding of the halting problem. However, this class of functions consists of a very small fraction of all binary functions.). This nonuniform class is thus associated with the uniform networks having large precision.

It was proven in [42] that, although real weight neural networks are defined with unbounded precision, they demonstrate the feature, referred to as "linear precision suffices". That is, up to the qth step of the computation, only the first $O(q)$ bits, in both weights and activation values of the neurons, influence the result. This means that for time bounded computation, only bounded precision is required. This property can be viewed as time-dependent resistance ("weak resistance") of the networks to noise and implementation error. (It is interesting to note that the amount of information necessary for the neural networks is identical to the precision required by chaotic systems, therefore neural networks may constitute a framework for the modeling of physical dynamics.)

In the current work, we consider recurrent neural networks that are allowed to exhibit stochastic and random behavior.

1.2 Randomness

Randomness is a basic characteristic of large distributed systems. It may result from the activity of the individual agents, from unpredictable changes in the communication pattern among the agents, or even just from the different update paces. All previous work that examined stochasticity in networks, e.g. [50, 36, 1, 34, 35, 8, 9], studied only acyclic architectures of binary gates, while we study general architectures of analog components. Due to these two qualitative differences, our results are totally different from the previous ones, and require new proof techniques.

Our particular stochastic model can be seen as an incorporation of the von Neumann model of unreliable interconnections of components to the area of neural networks: the basic component has a fixed probability ϵ for malfunction

at any step [50]. In contrast to the von Neumann model, here it is natural to allow for real values in ϵ, rather than rational values only. Furthermore, ϵ can be either a constant, as in the von Neumann model, or alternatively, a function of the history and the neighboring neurons. The latter, referred to as "the Markovian model," provides a useful model for stochastic computation. The element of stochasticity, when joined with exact known parameters, has the potential to increase the computational power of the underlying deterministic process. We find that it indeed adds some power, but only if the weights are rationals. In the cases of real weights and integer weights, this type of stochasticity does not change the computational power of the underlying process.

The proof concerning rational weights includes the following result from the realm of theoretical computer science. It is well known that probabilistic Turing machines that use binary coins with rational probabilities compute the class BPP. Here we consider binary coins having *real* probabilities and prove that the resulting polynomial time computational class is BPP/$log*$, which is BPP augmented with prefix logarithmic advice.

It is perhaps surprising that the real probabilities strengthens the Turing machine, because the machine still reads only the binary values of the coin flips. However, a long sequence of coin flips allows indirect access to the real valued probability, or more accurately, it facilitates its approximation with high probability. This is in contrast to the case of real weight networks, where access to the real values is direct and immediate. Thus, the resulting computation class (BPP/$log*$) is of intermediate computational power. It contains some nonrecursive functions, but is strictly weaker than P/poly.

Because real probabilities do not provide the same power as real weights, this work can be seen as suggesting a model of computation which is stronger than a Turing machine, but still is not as strong as real weight neural networks. Complementary to the feature of "linear precision suffices" for real weights we prove that for stochastic networks "logarithmic precision suffices" for the real probabilities; that is, for up to the qth step of the computation, only the first $O(\log q)$ bits in the probabilities of the neurons influence the result. We note that the same precision characterizes the quantum computer.

This report is organized as follows: Section 2 provides the required preliminaries of computational classes. Section 3 focuses on our stochastic networks, distinguishing them from a variety of stochastic models. Section 4 states the main results. Sections 5-7 include the proofs of the main theorems. In Section 8 we restate the model in various forms and in Section 9 we briefly describe a particular form of nondeterministic stochastic networks.

2 Preliminaries: Computational Classes

Let us shortly describe the computational classes relevant for this work. Turing machines are the basic computational model that is believed to describe all digital computers. Turing machine can be thought of computing input-output

maps of binary strings: the machine receives a binary string as an input and halts with another binary string on its tape, which is considered as the output. The functions computed by the various Turing machines constitute the *recursive* computational class. When the Turing machines are constrained to compute *efficiently* only, (i.e., to require not more time that polynomials in the length of the binary input string), the computational class is P. A particular subclass of P is the class of *regular* functions, which describes the functions computed by finite automata, see e.g. [18].

2.1 Probabilistic Turing Machines

The basis of the operation of the probabilistic Turing machine, as well as of our stochastic neural networks, is the use of random coins. In contrast to the deterministic machine, which acts on every input in a specified manner and responds in one possible way, the probabilistic machine may produce different responses for the same input.

Definition 1. ([2], volume I): A *probabilistic Turing machine* is a machine that computes as follows:

1. Every step of the computation can have two outcomes, one chosen with probability p and the other with probability $1 - p$.
2. All computations on the same input require the same number of steps.
3. Every computation ends with *reject* or *accept*.

All possible computations of a probabilistic Turing machine can be described by a full (all leaves at the same depth) binary tree whose edges are directed from the root to the leaves. Each computation is a path from the root to a leaf, which represents the final decision, or equivalently, the classification of the input word by the associated computation. A coin, characterized by the parameter p, chooses one of the two children of a node. In the standard definition of probabilistic computation, p takes the value $\frac{1}{2}$.

The *error probability* of a probabilistic Turing machine $\mathcal{M}$ is the function $e_{\mathcal{M}}(\omega)$ defined by the ratio of computations on input ω resulting in the wrong answer to the total number of computations on ω (which is equal to $2^{T(|\omega|)}$. The decision is defined as right or wrong with respect to a language L. Probabilistic computational classes are defined relative to the error probability. PP is the class of languages accepted by polynomial time probabilistic Turing machines with $e_{\mathcal{M}} < \frac{1}{2}$. A weaker class defined by the same machine model is BPP, which stands for *bounded error probabilistic polynomial time*. BPP is the class of languages recognized by polynomial time probabilistic Turing machines whose error probability is bounded above by some positive constant $\epsilon < \frac{1}{2}$. The latter class is recursive but it is unknown whether it is strictly stronger than P. There are other probabilistic classes such as R and ZPP, which are beyond the scope of this book, see e.g. [2].

2.2 Nonuniform Turing Machines

Nonuniform complexity classes are based on the model of advice Turing machines [21]; these, in addition to their input, receive also another sequence that assists in the computation. For all possible inputs of the same length n, the machine receives the same advice sequence, but different advice is provided for input sequences of different lengths. When the different advice strings cannot be generated from a finite rule (e.g. Turing machine) the resulting computational classes are called *nonuniform*. The nonuniformity of the advice translates into noncomputability of the corresponding class. The length of the advice is bounded as a function of the input, and can be used to quantify the amount of noncomputability.

Let Σ be an alphabet and \$ a distinguished symbol not in Σ; $\Sigma_\$$ denotes $\Sigma \bigcup \{\$\}$. We use homomorphisms h^* between monoids like $\Sigma_\$^*$ and Σ^* to encode words. Generally these homomorphisms are extensions of mappings h from $\Sigma_\$$ to Σ^*, inductively defined as follows: $h^*(\epsilon) = \epsilon$ and $h^*(a\omega) = h(a)h^*(\omega)$ for all $a \in \Sigma_\$$ and $\omega \in \Sigma_\$^*$. For example, when working with binary sequences, we usually encode "0" by "00", "1" by "11", and \$ by "01".

Let $A \subseteq \Sigma^*$ and $\nu : \mathbb{N} \to \Sigma^*$. Define $A_\nu = \{\omega\$\nu(|\omega|) \mid \omega \in A\}$, $A_\nu \subset \Sigma_\$$. Note that all words of A_ν that have the same length also receive the same suffix $\nu(|\omega|)$. This suffix is called the *advice*. We next encode A_ν back to Σ^* using a one to one homomorphism h^* as described above. We denote the resulting words by $\langle \omega, \nu(|\omega|) \rangle \in \Sigma^*$.

Definition 2. nonuniformity: Given a class of languages C and a class of bounding functions H. We say that $A \in C/H$ if and only if there is a function $\nu \in H$ such that $h^*(A_\nu) \in C$.

Some frequent H's are the space classes poly and log.

We next concentrate on the special case of prefix nonuniform classes [3]. In these classes, $\nu(n)$ must be useful for all strings of length up to n, not only those of length n. This is like in the definitions of "strong" [24] or "full" [3] nonuniform classes. Furthermore, $\nu(n_1)$ is the prefix of $\nu(n_2)$ for all lengths $n_1 < n_2$. Formally we define $\tilde{A}_\nu$ to be the set $\{\omega_1\$\nu(|\omega|) \mid \omega, \omega_1 \in A \text{ and } |\omega_1| < |\omega|\}$.

Definition 3. Prefix nonuniformity: Given a class of languages C and a class of bounding functions H. We say that $A \in \text{Pref-}C/H$ if and only if there is a prefix function $\nu \in H$ such that $h^*(\tilde{A}_\nu) \in C$. For the sake of brevity, we use the notation of C/H^* for the prefix advice class.

A special case is that of a Turing machine that receives a polynomially long advice and computes in polynomial time. The class obtained in this fashion is called P/poly, and in this case $\text{P/poly} = \text{P/poly}^*$. When exponential time and advice are allowed, *any* language on $\{0, 1\}$ is computable. Every such language can be recognized in the following form: just prepare a table of length 2^n whose

entries are associated with all the binary sequences of length n, in the lexicographic order. In each entry (sequence) write the bit "1" if the sequence is in the language, and '0" if it is not. Concatenate all 2^n bits into a sequence and use it as the advice for inputs of length n. This sequence encodes all the required information for accepting or rejecting any input sequence of length n.

Recall the probabilistic class BPP from subsection 2.1. We will later focus on the class BPP/ $\log *$. It is not hard to see that BPP/ $\log *$ is a strict subset of BPP/ log. Any tally set is in P/1 (and thus is clearly also in P/log and BPP/ log). Let S be a tally set, whose characteristic sequence is completely random (say, Kolmogorov random). It can not be in any class RECURSIVE/ $\log *$ because there is not enough information in $O(\log n)$ bits to get the n^{th} bit of S. As a special case S is not in BPP/ $\log *$.

3 Stochastic Networks

Four main questions are to be addressed when considering stochastic networks. How do we model stochasticity? What type of random behavior (or errors) should be allowed? How much randomness can be handled by the model? Finally, stochastic networks are not guaranteed to generate the same response in different runs of a given input; thus, how do we define the output of the network for a given input?

Modeling Stochasticity The first question, how to model stochasticity, was discussed by von-Neumann [50] and quoted by Pippenger [36]:

> The simplest assumption concerning errors is this: With every basic organ is associated a positive number ϵ such that in any operation, the organ will fail to function correctly with the (precise) probability ϵ. This malfunctioning is assumed to occur statistically independently of the general state of the network and of the occurrence of other malfunctions.

We first adopt von Neumann's statistical independence assumption. This assumption is also consistent with the works of Wiener [51] and Shannon [39], where the "noise" —which is the source of stochasticity— is modeled as a random process with known parameters. Note that in this model, the components are stochastic in precisely the amount ϵ, and "are being relied upon to behave unreliably in this exact amount" [36] (see also [8, 9, 35]). Similarly, in our work we assume either full knowledge of ϵ, or only knowledge of the first $O(\log T)$ bits of ϵ, where T is the computation time of the network. We show these two options to be equivalent. We then continue and expand to a Markovian model of stochasticity: here ϵ depends on the neighboring neurons and the recent history of the system. This richer model can be used to describe various natural phenomena.

Types of Randomness As for the second question, regarding the type of randomness, we consider any type of random behavior that can be modeled by augmenting the underlying deterministic process, either with independent identically distributed binary sequences (IID), or with Markovian sequences. We then abandon the stochastic coin model and substitute it with asynchronicity of update and with various nondeterministic reactions of the neurons themselves. This will be described in Section 8, where we discuss stochastically forgetting neurons (each neuron forgets its activation value with some probability; this forgetting is modeled as resetting the activation value to "0") and also the effect of probabilistic changes in the interconnection between neurons; various other types of randomness are proposed.

Amount of Randomness The next question we consider is the amount of stochasticity allowed in the model. Von Neumann assumed a *constant failure probability* ϵ in the various gates, independent of the size of the network. Furthermore, he allowed *all components* to behave randomly. Thus, larger networks suffer from more unreliability than smaller ones. In contrast, many later models allowed ϵ to decrease with the size of the net (see discussion in [36]), and others assumed the incorporation of fully deterministic/reliable components in critical parts of the network (see, e.g., [23, 29, 31, 49]). Because our network is recurrent, it is easy to verify that when ϵ is constant and all neurons are unreliable, no function requiring non-constant deterministic time $T(n)$ is computable by the network. We thus focus on networks which include both reliable/deterministic neurons and unreliable/stochastic neurons characterized by fixed ϵs. It is beyond the scope of this book, but it is worth noting that some biological modelings consider networks that combine deterministic and stochastic behavior. (An equivalent computational model can be achieved by allowing all neurons to behave randomly, while forcing the error to decrease polynomially with the parameter $T(n)$.)

Defining the Output Response As for the question of defining an output for the probabilistic process, we adopt the bounded error approach. Given $\epsilon < \frac{1}{2}$, we only consider networks that yield a wrong output on at most a fraction ϵ of the possible computations.

3.1 The Model

The underlying deterministic network is as introduced in Subsection 1.1. The following definition endows the network with stochasticity.

Definition 4. A stochastic network has additional input lines, called *stochastic lines*, that carry independent identically distributed (IID) binary sequences, one bit per line at each tick of the clock. The distributions may be different on the different lines. That is, for all time $t \geq 0$, the stochastic line l_i has the value 1 with probability p_i $(0 \leq p_i \leq 1)$, and 0 otherwise.

Equivalently, stochastic networks can be viewed as networks composed of two types of components: analog deterministic neurons and binary probabilistic gates/coins. A probabilistic gate is a binary gate which outputs "1" with probability $p \in [0,1]$.

Yet another way to view the same model is to consider it a network of neurons, part of which function deterministically, while others have "well-described faults"; that is, their faulty behavior can be described by a neural circuit. More on this and on other equivalent models appears in Section 8.

We define the recognition of a language by a stochastic network using the bounded error model as in BPP. We assume that the number of steps in all computations on an input ω is exactly the same. The classification of an input ω in a computation run is the reject or accept decision at the end of that computation. The final decision of ω considers the fraction of reject and accept classifications of the various computations.

Definition 5. Let $T : \mathbb{N} \to \mathbb{N}$ be a total function on natural numbers. We say that the language $L \subseteq \{0,1\}^+$ is *ϵ-recognized in time T* by a stochastic net $\mathcal{N}$ if every $\omega \in \{0,1\}^+$ is classified in time $T(|\omega|)$ by every computation path of $\mathcal{N}$ on ω, and the error probability in deciding ω relative to the language L is bounded: $e_{\mathcal{N}}(\omega) < \epsilon < \frac{1}{2}$. An equivalent definition is that for any given run on an input ω, if ω is not accepted in time $T(|\omega|)$, then it is rejected.

In the end notes (Section 10) we demonstrate how to arbitrarily reduce the recognition error in stochastic networks. This indicates that the following complexity class is well-defined.

Definition 6. S-NET is the class of languages that are ϵ-recognized by stochastic networks for any $\epsilon < \frac{1}{2}$.

4 The Main Results

Now that we have defined the model, we are ready to state the theorems about stochastic networks and compare them with deterministic networks. Proofs appear in the following sections.

4.1 Integer Networks

In the deterministic case, if the networks are restricted to integer weights, the neurons may assume only binary activations, and the networks become computationally equivalent to finite automata. Similar behavior occurs for stochastic networks.

Theorem 7. *The class* S-NET$_Z$ *of languages that are ϵ-recognized by networks with integer weights is the set of regular languages.*

The case of integer weights is considered only for the sake of completeness. Rational and real stochastic networks are of greater interest.

4.2 Rational Networks

In deterministic computation, if the weights are rational numbers, the network is equivalent in power to the Turing machine model. Two different I/O conventions are suggested in [43]; in the first, input lines and output neurons are used, and in the second, the discrete input and output are encoded as the state of two pre-fixed neurons.

In Section 2.2 we introduced nonuniform computational classes; in particular Definition 3 described the special case of prefix nonuniformity. We will next focus on the class BPP/$log*$. Because it includes nonrecursive functions, just like other nonuniform classes, the following theorem is of special interest:

The following theorem states the equivalence between rational stochastic networks and the class BPP/$log*$:

Theorem 8. *The class* S-NET$_Q$ $[p]$ *of languages ϵ-recognized by rational stochastic networks in polynomial time is equal to BPP/log*.*

Remark. As a special case of this theorem we note that if the probabilities are all rationals, then the resulting polynomial time computational class is constrained to BPP.

Recall that BPP is recursive; it is included both in P/poly ([2] pg. 144, cor. 6.3) and in $\Sigma_2 \bigwedge \Pi_2$ ([2] pg 172, Theorem 8.6). It is still unknown whether the inclusion P $\subseteq$ BPP is strict or not.

4.3 Real Networks

Deterministic real networks compute the class P/poly in polynomial time [42]. The addition of stochasticity does not yield a further increase in the computational power.

Theorem 9. *Denote by* NET$_R$ $(T(n))$ *the class of languages recognized by real networks in time $T(n)$, and by* S-NET$_R$ $(T(n))$ *the class of languages ϵ-recognized by real stochastic networks in time $T(n)$. Then*

$$\begin{aligned} \text{NET}_R\,(T(n)) &\subseteq \text{S-NET}_R\,(T(n)) \\ \text{S-NET}_R\,(T(n)) &\subseteq \text{NET}_R\,(n^2 + nT(n))\ . \end{aligned}$$

The results for polynomial stochastic networks are summarized in the following table:

Weights	Deterministic	Stochastic
$\mathbb{Z}$	regular	regular
$\mathbb{Q}$	P	BPP/log*
$\mathbb{R}$	P/poly	P/poly

Table 1. The computational power of recurrent neural networks

5 Integer Stochastic Networks

In this section we prove Theorem 7, which states the correspondence between probabilistic automata and integer stochastic networks. The classical definition of a probabilistic automaton (see [33] for example) is as follows:

Definition 10. A probabilistic automaton $\mathcal{A}$ is the 5-tuple $\mathcal{A} = (Q, \Sigma, p, q_0, F)$ where Q is a finite set of states, Σ is the finite input alphabet, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ are the accepting states. The probabilistic transition function $p(m, q, a), where m, q \in Q, and a \in \Sigma$, specifies the probability of getting into state m from a state q and the input symbol a. This transition function satisfies that for all q and a,

$$\sum_{m \in Q} p(m, q, a) = 1 \ .$$

We consider the double-transition probabilistic automaton which is a special case of probabilistic automata: the transition function may transfer each state-input pair into exactly two states. That is, for all $q \in Q$ and $a \in \Sigma$, there are exactly two states m_{1qa}, m_{2qa} such that $p(m_{1qa}, q, a), p(m_{2qa}, q, a) > 0$ and $p(m_{1qa}, q, a) + p(m_{2qa}, q, a) = 1$. It is easy to verify that this automaton is indeed equivalent to the general probabilistic automaton (up to a reasonable slowdown in the computation). The proof is left to the reader.

A double-transition probabilistic automaton can be viewed as a finite automaton with $|Q||\Sigma|$ additional input lines of IID binary sequences; these lines imply the next choice of the transition. From the equivalence of deterministic integer networks and finite automata, we conclude the equivalence between probabilistic automata and integer stochastic networks.

Rabin showed that bounded error probabilistic automata are computationally equivalent to deterministic ones ([33], p. 160). Thus, stochastic networks with integer weights are computationally equivalent to bounded error probabilistic automata, and S-NET$_Z$ is the class of regular languages.

6 Rational Stochastic Networks

This section is devoted to the proof of Theorem 8 which places rational stochastic networks in the hierarchy of super Turing computation.

We use a generalization of the classical probabilistic Turing machine [2] that substitutes the random coin of probability $\frac{1}{2}$ by a finite set of real probabilities. (A set of probabilities is required in order to describe neurons with different stochasticity.)

Definition 11. Let $S = \{p_1, p_2, \ldots p_s\}$ be a finite set of probabilities. A *Probabilistic Turing machine over the set* S is a nondeterministic machine that computes as follows:

1. Every step of the computation can have two outcomes, one chosen with probability p and the other with probability $1 - p$.
2. All computations on the same input require the same number of steps.
3. Every computation ends with *reject* or *accept*.

We denote by $\mathrm{BP}[S, T]$ the class of languages recognized in time T by bounded error probabilistic Turing machines with probabilities over S. We use the shorthand notation $\mathrm{BPP}[S] = \mathrm{BP}[S, poly]$. Similarly, we denote $\text{S-NET}_Q\,[S, T]$ as the class of languages recognized by rational stochastic nets in time T, with probabilities from S.

Lemma 12. *Let S be a finite set of probabilities, then $BP[S, T]$ and* $\text{S-NET}_Q\,[S, T]$ *are polynomially time related.*

Proof. 1. $\mathrm{BP}[S, T] \subseteq \text{S-NET}_Q\,[S, O(T)]$:
Let $\mathcal{M}$ be a probabilistic Turing machine over the set S that computes in time T. We simulate $\mathcal{M}$ by a rational stochastic network $\mathcal{N}$ having stochastic streams l_i with probabilities $p_i \in S$. Consider the program:

Repeat
  If (l_i=0) **then** NextStep(0, cur-state, cur-tapes)
    else NextStep(1, cur-state, cur-tapes)
Until (final state)

where NextStep is a procedure that given the current state of a Turing machine, the current tapes, and which of the two random choices to take, changes deterministically to the next configuration of the machine. This program can be compiled into a network that computes the same function, having no more than a linear slowdown [40].

2. $\text{S-NET}_Q\,[S, T] \subseteq \mathrm{BP}[S, \mathrm{poly}(T)]$:
It is easy to verify that if a rational stochastic network $\mathcal{N}$ has s IID input channels, then it can be simulated by a probabilistic Turing machine over the same s probabilities.

Next, we differentiate the case in which all probabilities of the set S are rational numbers from the case where S contains at least one real element.

6.1 Rational Set of Choices

Consider probabilistic Turing machines with probabilities over the set S, where S consists of rational probabilities only. Zachos showed that in the error bounded model, if the transition function decides its next state uniformly over k choices (k is finite but can be larger than 2), this model is polynomially equivalent to the classical probabilistic Turing machine with $k = 2$ [53]. When the probabilities are rationals, we can substitute them all by a common divisor which is written as $\frac{1}{k'}$ for an integer k'. This process increases the number of uniform choices, and implies polynomial equivalence between probabilistic Turing machines with one fair coin, and probabilistic machines over a set S. We conclude Remark 4.2 stating the computational equality between the class S-NET$_Q$ [poly] and the class BPP. Thus, rational stochasticity adds power to deterministic rational networks if and only if the class BPP is strictly stronger than P. Note that S-NET$_Q$ [poly] must be computationally strictly included in NET$_R$ [poly], because BPP is included in P/poly ([2] pg. 144, cor 6.3).

6.2 Real Set of Choices

Lemma 12 relates probabilistic Turing machines to stochastic neural networks. The lemma below completes the proof of Theorem 8 by showing the equivalence between real probabilities and log prefix advice in the probabilistic Turing model. We define $\mathrm{BP}_R(T) = \cup_{p\in[0,1]}\mathrm{BPP}[\{p\}, T]$ as the class of languages recognized by probabilistic bounded error Turing machines that use coins of real probability and compute in time T. $\mathrm{BP}_Q(T)/log*$ is similarly defined, with the addition of prefix advice. Note that $\mathrm{BP}_Q(poly)/log* = \mathrm{BPP}/log*$.

Lemma 13. *The classes $BP_R(T)$ and $BP_Q(T)/log*$ are polynomially related.*

Proof.

1. $\mathrm{BP}_R(T) \subseteq \mathrm{BP}_Q(O(T\log T))/\log*$:
Let $\mathcal{M}$ be a probabilistic Turing machine over the probability $p \in [0,1]$ that ϵ-recognizes the language L in time $T(n)$. We show that a probabilistic Turing machine $\mathcal{M}'$ having a fair coin, that upon receiving prefix advice of length $c\log(T(n))$ for a constant c, ϵ'-recognizes L in time $O(T(n)\log(T(n)))$. In *italics* we describe the algorithm for the simulation and then we bound its error probability.

> *Let p' be the rational number which is obtained from the $c\log(T(n))$ most significant bits of the binary expansion of p. The advice of $\mathcal{M}'$ consists of the bits of p' starting from the most significant bits.*
> *One coin flip by $\mathcal{M}$ can be simulated by a binary conjecture of $\mathcal{M}'$, which is based on $c\log(T(n))$ coin flips of its fair coin. $\mathcal{M}'$ tosses $c\log(T(n))$ times and compares the resulting guessed string with the advice to make a binary conjecture. If the guessed string precedes the advice in the lexicographic order, $\mathcal{M}'$ conjectures "0", otherwise $\mathcal{M}'$ conjectures "1".*

The error probability of $\mathcal{M}'$ is the probability that it generates a sequence of conjectures which yields a wrong decision. Denote by R the sequence of $T(n)$ binary conjectures that $\mathcal{M}'$ generates during its computation, and by B the set of $T(n)$-long binary sequences which are "bad", i.e. misleading conjectures. As the error of $\mathcal{M}$ is bounded by ϵ, the cardinality of B is bounded by $2^{T(n)}\epsilon$. We conclude that

$$\begin{aligned}\Pr\ (R \in B) &= \sum_{b\in B} \Pr\ (R = b) \leq |B| \max_{b\in B} \Pr\ (R = b) \\ &\leq\ 2^{T(n)}\epsilon \max_{b\in B} \Pr\ (R = b).\end{aligned}$$

We assume that $p \leq \frac{1}{2}$. Then the most probable sequence for $\mathcal{M}'$ to guess is $0^{T(n)}$. Because p and p' share the same $O(\log n)$ first bits,

$$|p' - p| \leq \frac{1}{T(n)^c}\ ,$$

and the total error of $\mathcal{M}'$ is bounded by

$$2^{T(n)}\epsilon(p + \frac{1}{T(n)^c})^{T(n)} = (2p)^{T(n)}\epsilon(1 + \frac{1}{pT(n)^c})^{T(n)}\ .$$

Using the formula

$$(1 + x_n)^n \approx e^{nx_n} \tag{3}$$

for small x_n, we approximate the above as

$$\approx (2p)^{T(n)}\epsilon e^{T(n)(pT(n))^{-c}}$$

which is bounded by an error $\epsilon' < \frac{1}{2}$ for $c > 1$.

2. $\mathrm{BP}_Q(T)/\log * \subseteq \mathrm{BP}_R(O(T^2))$:
Given a probabilistic Turing machine $\mathcal{M}$, having a fair coin and a logarithmically long prefix advice A that ϵ-recognizes a language L in time $T(n)$, we describe a probabilistic Turing machine $\mathcal{M}'$ with an associated real probability p that ϵ'-recognizes L in time $O(T^2(n))$.

The probability p is constructed as follows. The binary expansion of p starts with ".01", i.e. $\frac{1}{4} \leq p \leq \frac{1}{2}$; the following bits are the advice of $\mathcal{M}$.

$\mathcal{M}'$ computes in two phases:

Phase I — Preprocessing: *$\mathcal{M}'$ guesses the advice sequence A of $\mathcal{M}$ by tossing its unfair coin $z = O(T^2(n))$ times.*

Phase II — Simulating the computation of $\mathcal{M}$: *$\mathcal{M}'$ simulates each flip of the fair coin of $\mathcal{M}$ by up to $2n$ tosses using the following algorithm:*

(a) Toss the unfair coin twice.
(b) If the results are "01" conjecture "0", if they are "10" conjecture "1".
(c) If the results are either "00" or "11" and (a) was called less than $T(n)$, goto (a).
(d) Conjecture "0". (when (a) was called $T(n)$ times and the decision was not yet made.)

We first bound the error of guessing the advice. Let #1 be the number of "1"'s found in z flips, and define

$$p' = \frac{\#1}{z} .$$

We next show that p' is a good estimate for p by bounding $|p-p'|$. The Chebychev formula states that for any random variable x with expectation μ and variance ν, and $\forall \epsilon > 0$, $\Pr\ (|x-\mu|) > \epsilon) \leq \frac{v}{\epsilon^2}$. Here x is the sum of iid random variables; The expectation of such z independent Bernoulli trials is $\mu = zp$, and the variance is $\nu = zp(1-p)$. We conclude that for all $\epsilon > 0$,

$$\Pr\ (|zp' - zp| > \epsilon) \leq \frac{zp(1-p)}{\epsilon^2} .$$

Because $p(1-p) \leq \frac{1}{4}$ (when $p \in [\frac{1}{4}, \frac{1}{2}]$), by choosing $\epsilon = \sqrt{25z}$, we get

$$\Pr\ \left(|p' - p| > \sqrt{\frac{25}{z}} \right) \leq \frac{1}{100} .$$

Thus, if in the first phase $\mathcal{M}'$ tosses its coin $z = 25T^2(n)$ times, then the advice is reconstructed with logarithmically many bits and with an error probability bounded by $\frac{1}{100}$ (the first two bits "01" are omitted from the guessed advice).

We next prove the correctness of phase II. We compute the probability of $\mathcal{M}'$ to guess a bad sequence of coin flips. As above, we denote the set of misleading guesses by B, and by R the sequence of binary conjectures of length $T(n)$ generated by $\mathcal{M}'$ during the algorithm.

$$\Pr\ (R \in B) = \sum_{b \in B} \Pr\ (R = b) \leq |B| \max_{b \in B} \Pr\ (R = b)$$

- The probability of getting the values "00" or "11" in two successive coin flips is $p' = p^2 + (1-p)^2$. Thus, the probability of ending a coin flip simulation in step (d) of the algorithm is bounded by $p'' = p'^{T(n)}$. Since $\frac{1}{4} \leq p \leq \frac{1}{2}$, we conclude that $p' \leq \frac{5}{8}$ and $p'' \leq \frac{5}{8}^{T(n)}$.
- The probability of ending one coin flip simulation with the conjecture "0" is: $\frac{1}{2}(1-p'') + p'' = \frac{1}{2} + \frac{p''}{2}$.
- $\max_b$ Pr $(R = b) \leq$ Pr $(R = 0^{T(n)}) = (\frac{1}{2} + \frac{p''}{2})^{T(n)}$.

Thus,

$$\begin{aligned} \Pr\ (R \in B) &\leq 2^{T(n)} \epsilon (\frac{1}{2} + \frac{p''}{2})^{T(n)} \leq \epsilon (1 + p'')^{T(n)} \\ &\approx \epsilon e^{T(n)\frac{5}{8}^{T(n)}} < \epsilon_2 \ . \end{aligned}$$

The error probability ϵ' of $\mathcal{M}'$ is bounded by

$$\Pr(\text{"wrong advice sequence"}) + \Pr(\text{"bad guess sequence"}) \leq \tfrac{1}{100} + \epsilon_2$$

which is also bounded by $\frac{1}{2}$. ∎

Remark. So far we have discussed stochastic networks, defined by adding choices to deterministic networks. We can similarly define the stochastic nondeterministic network by adding choices to nondeterministic networks. When weights are rationals, the latter class is similar to the framework of interactive proof systems ([2] volume I, chapter 11).

7 Real Stochastic Networks

In this section we prove Theorem 9 and show that stochasticity does not add power to real deterministic networks. It is trivial to show that stochasticity does not decrease the power of the model; we thus focus on the other direction and prove that S-NET$_R$ [poly] $\subseteq$ NET$_R$ [poly].

We prove this inclusion in two steps. Given a real stochastic network that ϵ-recognizes a language L, the first step describes a nonuniform family $\mathcal{F}$ of feedforward networks that recognizes L; this creates only a constant slowdown in the computation. The second step describes a deterministic recurrent network that simulates the family $\mathcal{F}$, with a polynomial slowdown of $n^2 + nT(n)$. (We could skip the first step with a more elaborate counting argument in the second step, but we prefer this method for its simplicity of representation.)

Lemma 14. *Step 1: Let $0 < \epsilon < \frac{1}{2}$, and let L be a language that is ϵ-recognized by a real stochastic network $\mathcal{N}$ of size N in time T. Then, L can also be recognized by a family of nonuniform feedforward real networks $\mathcal{F} = \{\mathcal{N}_n\}_{i=1}^{\infty}$ of depth $T(n)+1$ and size $cNT(n)+1$, where*

$$c = \lceil \frac{8\epsilon \ln 2}{(1-2\epsilon)^2} \rceil \ .$$

Proof. The technique used in this proof is similar to the one used in the proof that BPP $\subseteq$ P/poly [2].

Let $\mathcal{N}$ be a real network that ϵ-recognizes a language L as above. We show the existence of a family of networks that recognizes L. Let r be the number of probabilistic gates g_k, $1 \leq k \leq r$; each outputs "1" with probability p_k. For a given input of length n, by unfolding the network to $T(n)$ layers, each a copy of $\mathcal{N}$, we get a feedforward network with $r'_n = T(n)r$ probabilistic gates. Denote this feedforward stochastic net by $\mathcal{N}'_n$.

We pick a string

$$\rho^{i,n} = \rho^i_1 \rho^i_2 \cdots, \rho^i_{r'_n} \ \in \{0,1\}^{r'_n}$$

at random, with probability $p_{(j \mod r)}$ that ρ^i_j is "1". Let $\mathcal{N}'_n\{\rho^{i,n}\}$ be a deterministic feedforward net similar to $\mathcal{N}'_n$, but with the string $\rho^{i,n}$ substituting the probabilistic gates (i.e., ρ^i_j substitutes $g_{j \bmod r}$ in level (j div r). We now pick cn such strings

$$\rho[n] = (\rho^{1,n}, \rho^{2,n}, \ldots \rho^{cn,n})$$

at random. The feedforward net $\mathcal{N}_n$ consists of the cn subnetworks $\mathcal{N}'_n\{\rho^{i,n}\}$ ($i = 1 \ldots cn$) and one "majority gate" in the final level. The majority gate takes the output of the cn subnetworks as its input. That is, for each n, the network $\mathcal{N}_n$ computes the majority over cn random runs of the stochastic net $\mathcal{N}$.

We compute the probability that $\mathcal{N}_n$ outputs incorrectly on an input ω of length n. By the definition of $\mathcal{N}$, each $\mathcal{N}'_n\{\rho^{i,n}\}$ has the probability ϵ of being wrong. Thus, picking $\rho[n]$ at random, the probability of $\mathcal{N}_n$ to fail is bounded by $B(\frac{cn}{2}, cn, \epsilon)$; this is the probability of being wrong in at least $cn/2$ out of cn independent Bernoulli trials, each having the failure probability ϵ.

We use the bound $B(\frac{cn}{2}, cn, \epsilon) \leq (4\epsilon(1-\epsilon))^{cn/2}$ from [32], and choose $c = \lceil \frac{8\epsilon \ln 2}{(1-2\epsilon)^2} \rceil$ to get

$$B(\frac{cn}{2}, cn, \epsilon) < 2^{-n} \ .$$

This is a bound on the error probability for any input of length n. Thus, the sum of failures for all the inputs of length n is less than 1. Hence, there must be at least one choice of random strings $\rho[n]$ that makes $\mathcal{N}_n$ correctly recognize any input of length n.

The above lemma of the 2-step algorithm introduces a family $\mathcal{F}$ of deterministic feedforward networks that decides L. This specially structured family will be shown in the next step to be included in P/poly. More specifically, in the following lemma we complete the proof of Theorem 9 with a construction of a real deterministic network that simulates $\mathcal{F}$.

Lemma 15. *Step 2: Any language that is recognized by the real family $\mathcal{F} = \{\mathcal{N}_n\}_{n=1}^{\infty}$ described above can also be recognized by a real deterministic recurrent network $\mathcal{N}_r$. Furthermore, an input of length n that was recognized by the network $\mathcal{N}_n$ having depth $T(n)$, can be recognized by $\mathcal{N}_r$ in time $O(n^2 + nT(n))$.*

Proof. We remind the reader that the whole family $\mathcal{F}$ was constructed from a single recurrent neural network; call it $\mathcal{N}_2$. Each member $\mathcal{N}_n$ of $\mathcal{F}$ can thus be described by the tuple

$$(\mathcal{N}_2, n, \rho[n])$$

where $\mathcal{N}_2$ is the underlying deterministic recurrent network, n is the index of the network, and $\rho[n] \in \{0,1\}^{r'_n cn}$.

Let $\tilde{\mathcal{N}_2}$ be any binary encoding of $\mathcal{N}_2$ and $\tilde{\rho}$ be the infinite string

$$\tilde{\rho} = \rho[1]\ \ 2\ \ \rho[2]\ \ 2\ \ \rho[3]\ \ 2\ \cdots\ \ .$$

Let $\alpha = \alpha_1\alpha_2\cdots \in \{0,1,2\}^{\#}$ and denote by $\alpha|_6$ the value $\sum_{i=1}^{|\alpha|} \frac{2\alpha_i+1}{6^i}$. This encoding is Cantor-like, and a network can read weights of this form letter by letter in $O(1)$ time each (see [43]). We next construct the recurrent network $\mathcal{N}_r$ that has the weights $\tilde{\mathcal{N}_2}|_6$ and $\tilde{\rho}|_6$ and recognizes the language L. $\mathcal{N}_r$ operates as follows:

1. $\mathcal{N}_r$ reads the input ω and measures its length.
2. $\mathcal{N}_r$ retrieves the encoding $\rho[n]$ from the constant $\tilde{\rho}|_6$. (This takes $\sum_{j\le n} |\rho[j]| \le O(n^2)$ as proven in [42]).
3. $\mathcal{N}_r$ executes the code:

```
Func Net (ω, ρ[n], n, N̂2);
Var      Yes, No, z: Counter,
         A: Boolean,
         ρ: Real;
Begin
   z=1 ;
   Repeat
      ρ ← retrieve(ρ̃, i, n)        % retrieving ρ^{i,n}
      A ← simulate (N̂2, ρ, ω)
      If A = 1 then Increment(Yes) else Increment(No)
      Increment(z)
   Until (z > cn)
   Net ← Return(Yes > No)
End
```

We know that the command "simulate" is feasible from the constructive Turing machine simulation in [43]. Furthermore, it was shown in [40] how to construct a net from this type of high-level language. This program, as well as the associated network, takes $nT(n)$ steps. Thus $\mathcal{N}_r$ fulfills the requirements of Lemma 15, and Theorem 9 is proven.

8 Unreliable Networks

In this section we provide a different formulation of stochastic networks. Our von-Neumann like modeling captures many types of random behavior in networks. It can describe the probabilistic "forgetting" neuron:

$$x_i^+ = \begin{cases} \text{regular update} & \text{with probability } 1-p_i \\ 0 & \text{with probability } p_i \ ; \end{cases} \tag{4}$$

as well as the probabilistic "persistent" neuron:

$$x_i^+ = \begin{cases} \text{regular update} & \text{with probability } 1-p_i \\ x_i & \text{with probability } p_i \ ; \end{cases} \tag{5}$$

and the probabilistic "weakly connected" neuron x_i, which is defined by the update equation: $x_i(t+1) = \sigma\left(\sum_{j=1}^{N} \tilde{a}_{ij} x_j(t) + \sum_{j=1}^{M} b_{ij} u_j(t) + c_i\right)$, where

$$\tilde{a}_{ij}^k = \begin{cases} a_{ij}^k & \text{with probability } 1-p_{ij} \\ 0 & \text{with probability } p_{ij} \ . \end{cases} \tag{6}$$

We generally allow not only for these three types of errors but for more general faults. Each stochastic neuron computes one of d functions which may be more complicated than the above examples.

Definition 16. A neuron g is said to have a *well-described fault* if it computes d different functions f_i each with a probability $p_{gi} (\sum_{i=1}^{d} p_{gi} = 1)$ such that all the f_i's are deterministically computable by a net in constant time.

It is not difficult to verify the following:

Lemma 17. *A stochastic network that consists of both well-described faulty and deterministic neurons can be described by our stochastic modeling.*

The proof is left to the reader.

Remark. Note that if all neurons are stochastic ("catastrophic nets"), the stochasticity can no longer be controlled, and no function requiring non-constant deterministic time T is computable by such a network. Introducing a varying error rate or a nonuniform architecture allows one to overcome the catastrophe.

Similarly, we can define the "Markovian" model of unreliability: unlike the model of independent erroneous neurons, let us consider devices whose unreliable behavior depends on the last c choices of all devices and the last c global states in the network. This better models biological phenomena such as dying neurons, toxication and the Korsakoff syndrome [10]. Note that this model is not strictly Markovian because transitions do not depend only on the global states, but on the choices as well.

Definition 18. Let g be a well-described faulty neuron with d choices. Let $x(i) \in [0,1]^N$ be the activation vector in time i, and let the vector

$$F_{c,t} = (x(t-c), \ldots, x(t-1)) \in [0,1]^{cN}$$

represent the activation values of the neurons in the previous c steps. Similarly, let $j(i) \in \{1,2,\ldots,d\}^N$ be the vector of choices made by all neurons in time i, and let the vector

$$J_{c,t} = (j(t-c), j(t-c+1), \ldots, j(t-1)) \in \{1,2,\ldots,d\}^{cN}$$

represent the choices made by all neurons in the previous c steps. Observe that only $x(t-c)$ in $F_{c,t}$ is necessary: the rest can be computed from $x(t-c)$ and the choices $J_{c,t}$; we use this redundancy for simplicity of presentation.

A *c-Markov network* is a network with some unreliable neurons, for which the probabilities p_{gi} are functions of $J_{c,t}$ and $F_{c,t}$; there exist dN stochastic subnetworks that upon receiving $p_{gi}(t)$ as input, output "1" with this probability.

We state without proof:

Theorem 19. *For any well-described fault and any constant $c \geq 1$, c-Markovian networks are computationally equivalent to networks of independent unreliability.*

One final equivalent model that we note is the *asynchronous model*. To characterize the behavior of the asynchronous neural networks, we adapt the classical assumption of asynchronous distributed systems: no two neurons ever update simultaneously. An *asynchronous network* is a network with an additional N-level probabilistic gate, g, where level l_i appears with probability p_i ($\sum_i p_i = 1$). At each time t, only processor $g(t)$ updates, and the output is interpreted probabilistically.

9 Nondeterministic Stochastic Networks

While digital computing nondeterminism has a single definition, it has two possible interpretations in analog models. Weak nondeterminism incorporates guesses of random bits into the computation. The strong nondeterministic model incorporates guesses of real numbers [6].

Definition 20. A *stochastic architecture* $\mathcal{A}$ is a network in which the probabilities are variables v_i. A *nondeterministic stochastic architecture* is an architecture $\mathcal{A}$ that when given an input string ω, guesses the values of the probabilities $v = v_1, v_2, \ldots v_N$ and outputs a probabilistic response $\mathcal{A}_v(\omega) \in \{0,1\}$. As $\mathcal{A}_v$ is a stochastic network, it $\epsilon(v)$ recognizes a language $L_{\mathcal{A}_v}$. The language accepted by the architecture is thus

$$L_{\mathcal{A}} = \{\omega \mid \exists v, \epsilon(v) \in (0, \frac{1}{2}) : \ \mathcal{A}_v(\omega) = 1 \ \text{ with probability } \ > 1 - \epsilon(v)\} \ .$$

As in the model of computation over the real numbers, we consider weak and strong nondeterminism. We say that L is accepted by $\mathcal{A}$ using a *strong model of nondeterminism* if $v \in \mathbb{R}^N$, and by a *weak model of nondeterminism* when v is a vector of N (non-periodic) rationals represented with $O(T)$ bits.

Lemma 21. *Let nondet-*S-NET$_U$ *[poly] be the nondeterministic counterpart of* S-NET$_U$ *[poly], where* $U \in \{Q, R\}$*, then* S-NET$_U$ *[poly] = nondet-*S-NET$_U$ *[poly]*$_{\text{weak}}$ *=* nondet-S-NET$_U$ *[poly]*$_{\text{strong}}$.

When considering rational weights, this says no more than that the class BPP/*log*$*$ is equivalent to its nondeterministic version; it is a simple corollary of the observation that only the first $O(\log T(n))$ bits of the probabilities v are significant in a probabilistic computation.

10 End Notes: Reducing the Error Probability

The following lemma is standard for probabilistic models of computation. It shows that the error probability can be reduced to any desired value. The particular form we use is analogous to the lemma concerning threshold circuits [32].

Lemma 22. *Let* $f(\lambda, \epsilon) = \lceil \frac{2 \log \lambda}{\log(4\epsilon(1-\epsilon))} \rceil$*. For every* $0 < \lambda < \epsilon < \frac{1}{2}$*, any language that is* ϵ*-recognized by a stochastic network* $\mathcal{N}$ *of size* N *and in computation time* T*, can also be* λ*-recognized by another stochastic network* $\mathcal{N}'$ *either of size* $Nf(\lambda, \epsilon)$ *and in time* $(T+1)$*, or of size* $(N+10)$ *and in time* $T(f(\lambda, \epsilon)+10)$.

Proof. (Sketch):
The main idea is to perform b independent computations of the network $\mathcal{N}$ on the same input ω, and output the majority result of these computations.

The probability of an error in this experiment is the probability of having an error in at least half of b independent Bernoulli trials, where each trial has a failure probability ϵ. Given b Bernoulli trials, the probability of failing up to m of them is

$$B(m, b, \epsilon) = \sum_{i=1}^{m} \binom{b}{i} \epsilon^i (1-\epsilon)^{b-i} .$$

For $m = \lceil \frac{b}{2} \rceil - 1$ this probability can be bounded by $(4\epsilon(1-\epsilon))^{\frac{b}{2}}$. Choosing $b \geq \frac{2 \log \lambda}{\log(4\epsilon(1-\epsilon))}$, the error is bounded by λ.

The independent trials can be implemented either serially or in parallel. In the serial case, the net computes b trials serially, using the same hardware, while keeping track of all of its history decisions, and decides upon the majority. Ten neurons suffice for the necessary bookkeeping. The parallel version yields a network that consists of b parallel copies of the original hardware and an additional neuron to majorize the results.

11 Acknowledgements

I wish to thank Joe Kilian, Ricard Gavaldà, Michael Saks, Bill Horne, Seppo Törmä and Felix Costa for the many helpful comments.

References

1. L. Adleman. Two theorems on random polynomial time. In *IEEE Sympos. on Foundations of Computer Science*, volume 19, pages 75–83, New-York, 1978.
2. J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity*, volume I and II. Springer-Verlag EATCS Monographs, Berlin, 1988-1990. Second Edition for Volume I in 1995.
3. J. L. Balcázar, M. Hermo, and E. Mayordomo. Characterizations of logarithmic advice complexity classes. *Information Processing 92, IFIP Transactions A-12*, 1:315–321, 1992.
4. A.R. Barron. Neural net approximation. In *Proc. Seventh Yale Workshop on Adaptive and Learning Systems*, pages 69–72, Yale University, 1992.
5. E.B. Baum and D. Haussler. What size net gives valid generalization? *Neural Computation*, 1:151–160, 1989.
6. L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: Np completeness, recursive functions, and universal machines. *Bull. A.M.S.*, 21:1–46, 1989.
7. G. Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control, Signals, and Systems*, 2:303–314, 1989.
8. R.L. Dobrushin and S.I. Ortyukov. Lower bound for the redundancy of self-correcting arrangement of unreliable functional elements. *Problems info, Transmission*, 13:59–65, 1977.
9. R.L. Dobrushin and S.I. Ortyukov. Upper bound for the redundancy of self-correcting arrangement of unreliable functional elements. *Problems info, Transmission*, 13:346–353, 1977.
10. Y. Finkelstein. *Cholinergic Mechanisms of Control and Adaptation in the Rat Septo-Hippocampus under Stress Conditions.* PhD thesis, Hebrew University in Jerusalem, Israel, 1994.
11. J.A. Franklin. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2:183–192, 1989.
12. S. Franklin and M. Garzon. Neural computability. In O. M. Omidvar, editor, *Progress In Neural Networks*, pages 128–144. Ablex, Norwood, NJ, 1990.
13. M. Garzon and S. Franklin. Neural computability. In *Proc. 3rd Int. Joint Conf. Neural Networks*, volume II, pages 631–637, 1989.
14. C.L. Giles, B.G. Horne, and T. Lin. Learning a class of large finite state machines with a recurrent neural network. *Neural Networks*, 1995. In press.
15. R. Hartley and H. Szu. A comparison of the computational power of neural network models. In *Proc. IEEE Conf. Neural Networks*, pages 17–22, 1987.
16. S. Haykin. *Neural Networks: A Comprehensive Foundation.* IEEE Press, New York, 1994.
17. J.W. Hong. On connectionist models. *On Pure and Applied Mathematics*, 41, 1988.

18. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
19. K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4:251–257, 1991.
20. K. Hornik, M. Stinchcombe, and H. White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3:551–560, 1990.
21. R.M. Karp and R.J. Lipton. Some connections between uniform and nonuniform complexity classes. In *Proceedings of 12th ACM Symp. on Theory of Computing*, pages 302–309, 1980.
22. J. Kilian and H.T. Siegelmann. On the power of sigmoid neural networks. In *Proc. Sixth ACM Workshop on Computational Learning Theory*, Santa Cruz, July 1993.
23. G.I. Kirienko. Sintez samokottektiruyshchikhsya skhem iz funktsionalnykh elementov dlya aluchava tastushchego chisla oshibok v skheme. *Diskret. Anal.*, 16:38–43, 1970.
24. K. Ko. On helping by robust oracle machines. *Theoretical Computer Science*, 52, 1987. 15-36.
25. P. Koiran, M. Cosnard, and M. Garzon. Computability with low-dimensional dynamical systems. *Theoretical Computer Science*, 132:113–128, 1994.
26. W. Maass, G. Schnitger, and E.D. Sontag. On the computational power of sigmoid versus boolean threshold circuits. In *Proc. 32nd IEEE Symp. Foundations of Comp. Sci*, pages 767–776, 1991.
27. M. Matthews. On the uniform approximation of nonlinear discrete-time fading-memory systems using neural network models. Technical Report Ph.D. Thesis, ETH No. 9635, E.T.H. Zurich, 1992.
28. C.B. Miller and C.L. Giles. Experimental comparison of the effect of order in recurrent neural networks. *International Journal of Pattern Recognition and Artificial Intelligence*, 7(4):849–872, 1993. Special Issue on Neural Networks and Pattern Recognition, editors: I. Guyon , P.S.P. Wang.
29. A. A. Muchnik and S. G. Gindikin. The completeness of a system made up of non-reliable elements realizing a function of algebraic logic. *Soviet Phys. Dokl*, 7:477–479, 1962.
30. P. Orponen. Neural networks and complexity theory. In *Proc. 17th Symposium on Mathematical Foundations of Computer Science*, pages 50–61, 1992.
31. S.I. Ortyukov. Synthesis of asymptotically nonredundant self-correcting arrangements of unreliable functional elements. *Problems Inform. Transmission*, 13:247–251, 1978.
32. I. Parberry. *Circuit Complexity and Neural Networks.* MIT Press, 1994.
33. A. Paz. *Introduction to Probabilistic Automata.* Academic Press, New York, 1971.
34. N. Pippenger. Reliable computation by formulae in the presence of noise. *IEEE Trans. Inform. Theory*, 34:194–197, 1988.
35. N. Pippenger. Invariance of complexity measure of networks with unreliable gates. *J. ACM*, 36:531–539, 1989.
36. N. Pippenger. Developments in: The synthesis of reliable organisms from unreliable components. In *Proc. of symposia in pure mathematics*, volume 5, pages 311–324, 1990.
37. J. B. Pollack. *On Connectionist Models of Natural Language Processing.* PhD thesis, Computer Science Dept, Univ. of Illinois, Urbana, 1987.

38. M. M. Polycarpou and P.A. Ioannou. Identification and control of nonlinear systems using neural network models: Design and stability analysis. Technical Report 91-09-01, Department of EE/Systems, USC, Los Angeles, Sept 1991.
39. C. E. Shannon. A mathematical theory of communication. *Bell System Tech J.*, pages 379–423, 623–656, 1948.
40. H. T. Siegelmann. On nil: The software constructor of neural networks. *Parallel Processing Letters*, 6(4):575–582, 1996.
41. H. T. Siegelmann and E. D. Sontag. Turing computability with neural nets. *Appl. Math. Lett.*, 4(6):77–80, 1991.
42. H. T. Siegelmann and E. D. Sontag. Analog computation via neural networks. *Theoretical Computer Science*, 131, 1994. 331-360.
43. H. T. Siegelmann and E. D. Sontag. On computational power of neural networks. *J. Comp. Syst. Sci*, 50(1):132–150, 1995. Previous version appeared in *Proc. Fifth ACM Workshop on Computational Learning Theory,* pages 440-449, Pittsburgh, July 1992.
44. H.T. Siegelmann, B.G. Horne, and C.L. Giles. Computational capabilities of recurrent narx neural networks. Technical Report UMIACS-TR-95-12 and CS-TR-3408, Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland, 1995.
45. E.D. Sontag. Neural nets as systems models and controllers. In *Proc. Seventh Yale Workshop on Adaptive and Learning Systems*, pages 73–79, Yale University, 1992.
46. E.D. Sontag. Neural networks for control. In H.L. Trentelman and J.C. Willems, editors, *Essays on Control: Perspectives in the Theory and its Applications*. Birkhauser, Boston, 1993.
47. M. Stinchcombe and H. White. Approximating and learning unknown mappings using multilayer feedforward networks with bounded weights. In *Proceedings of the International Joint Conference on Neural Networks, IEEE*, 1990.
48. H.J. Sussmann. Uniqueness of the weights for minimal feedforward nets with a given input-output map. *Neural Networks*, 5:589–593, 1992.
49. D. Ulig. On the synthesis of self-correcting schemes from functional elements with a small numer of reliable elements. *Math. Notes. Acad. Sci. USSR*, 15:558–562, 1974.
50. J. von Neumann. Probabilistic, logics and the synthesis of reliable organisms from unreliable components. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*. Princeton U. Press, Princeton, NJ, 1956.
51. N. Wiener. *Extrapolation, interpolation, and smoothing of stationary time series*. MIT Press, Cambridge, MA, 1949.
52. D. Wolpert. A computationally universal field computer which is purely linear. Technical Report LA-UR-91-2937, Los Alamos National Laboratory, 1991.
53. S. Zachos. Robustness of probabilistic computational complexity classes under definitional perturbations. *Information and Control*, 54:143–154, 1982.