Computation in Recurrent Neural Networks: From Counters to Iterated Function Systems

Yvonne Kalinke [1] * and Helko Lehmann [2]

[1] Queensland University of Technology
yvonne@fit.qut.edu.au
[2] University of Technology Dresden
eld@inf.tu-dresden.de

Abstract. In the paper we address the problem of computation in recurrent neural networks (RNN). In the first part we provide a formal analysis of the dynamical behavior of a RNN with a single self–recurrent unit in the hidden layer, show how such a RNN may be designed to perform an (unrestricted) counting task and describe a generalization of the counter network that performs binary stack operations.
In the second part of the paper we focus on the analysis of RNNs. We show how a layered RNN can be mapped to a corresponding iterated function system (IFS) and formulate conditions under which the behavior of the IFS and therefore the behavior of the corresponding RNN can be characterized as the performance of stack operations. This result enables us to analyze *any* layered RNN in terms of classical computation and, hence, improves our understanding of computation within a broad class of RNNs.
Moreover, we show how to use this knowledge as a *design principle* for RNNs which implement computational tasks that require stack operations. This principle is exemplified by presenting the design of particular RNNs for the recognition of words within the class of Dyck languages.

Introduction

So far, a lot of work has been done to find design principles for recurrent neural networks (RNN) such that cognitive tasks given in terms of classical computation can be translated into tasks which can be handled by the RNN (e.g., [3], [8], [11]). Other approaches are based on finding suitable RNN architectures which can be trained to perform these cognitive tasks (e.g., [14], [4]). Though, most of these RNNs were successfully used within certain applications, we still do not have a comprehension of computation in these networks and therefore can not answer questions like: Why does a RNN actually solves the task it was designed and trained for? Why are the computational capabilities of the RNN wrt. some tasks restricted? Does a RNN, i.e. its architecture, actually fit to the given task? So it turns out that, since our knowledge about how to interpret the behavior of RNNs in terms of computation is quite limited, we are not able to construct RNNs appropriate to an arbitrary computational task.

In contrast to that there are promising results coming from the investigations of *computational capabilities* of RNNs. In [13] it has been shown that there are no ultimate limitations imposed by the use of neural networks as computing devices and that we do not even need higher order neural networks to have enough computational power. It has been proven that a RNN made up of neurons using linear saturation activation functions can simulate an universal Turing machine[1] . However, this result does not provide the information we are interested

* The author acknowledges support from the German Academic Exchange Service (DAAD) under grant no. D/97/29570.

[1] The proof uses a stack construction that is similar to the one we shortly sketch within this paper.

in, since it does not answer the question on how to interpret computation within RNNs.

Following this question, the behavior of RNNs is commonly interpreted as behavior of Deterministic Finite State Machines (DFSM) (e.g., [5], [2]). However, in [9] it has been shown that this may be not adequate and that RNNs are closely related to iterated function systems (IFS). Anyway, we do not share the opinion that we need a new computational model to understand the behavior of RNNs, since the Turing machine and related models are not restricted to finite state sets. But, actually, IFSs are closely related to RNNs and in this paper we will show that layered RNNs can be mapped to IFSs and that IFSs provide a tool to analyze and understand computation in RNNs wrt. to certain computational tasks.

Some work has been done trying to examine dynamics within RNNs, e.g. in [4] and [12] simple RNNs were trained to learn context free languages up to a certain word length. It turned out that the RNNs behaved like a counter implementation. But in these papers no explanation for the restriction of the counter to a certain word length has been given. In [10] a partial analysis of the dynamics of sigmoidal activation functions (which is widely used in RNNs that have to be trained) is given and the possibility of bifurcation behavior is proposed as a model for *dynamics of attention*. In [7] we provided a complete[2] analysis of the dynamics of sigmoidal activation functions, where, in opposite to [10], we investigated also cases of non fixed point behavior. In [6] a higher order RNN (using sigmoidal activation functions) has been shown to recognize words in a context sensitive prediction task up to a certain length. To understand and prove the proposed behavior of the network an analysis of the dynamics has been given.

Since the application of continuous sigmoidal activation functions does not only complicate the analysis of the dynamical behavior, but moreover, does not allow the implementation of unrestricted stack operations (see Section 3 for some details), we decided to use piecewise linear activation functions instead. Analogously to the formal analysis of the dynamics of a RNN with a single recurrent unit in the hidden layer using a sigmoidal activation function in [7] within this paper we present a formal analysis for such a RNN using a linear saturation activation function. This analysis enables us to exploit the dynamical behavior of the activation function to construct a RNN that implements a counter which is not restricted to a certain word length. Such a counter moreover can be viewed as a specialization of a stack and we describe a RNN which is a generalization of the counter RNN and implements binary stack operations.

The paper is organized as follows. In the first section we introduce some notions and notations used throughout the paper. This is followed by the analysis of the dynamics of a RNN with a single recurrent unit in the hidden layer. In Section 3 we describe how to use the dynamical behavior of the RNN described in the second section to set up a RNN performing a simple counting task. Afterwards we extend the counter RNN to a RNN that is capable to perform binary stack operations. The main emphasis of the paper is treated in Section 5 where we address the relation between stack operations, IFS and RNN. We show that the behavior of IFSs can be interpreted in computational terms of stack operations and that layered RNNs can be mapped to corresponding IFSs. We moreover formulate conditions under which an IFS implements stack operations what finally leads us to a procedure allowing to analyze whether an *arbitrary* layered RNN performs stack operations, where arbitrary means that we allow arbitrary activation functions, number of layers, training algorithms or design principles etc. In Section 6 we show how the correspondence between IFSs implementing

[2] complete in the sense of investigating the whole parameter space

stack operations and RNNs can be used as a *design principle*. As an example we show how to construct RNNs that recognize Dyck languages $D(n)$ (for an arbitrary choice of n, respectively). Dyck languages are context free languages which cannot be recognized by DFSMs or one–counter–automata (if $n \geq 2$). We conclude with a discussion of the results.

1 Notions and Notations

A *metric space* $(\mathbf{X}, d)$ consists of a set $\mathbf{X}$ and a distance function $d : \mathbf{X} \times \mathbf{X} \to \mathbb{R}$. A *dynamical system* consists of a metric space $(\mathbf{X}, d)$ and a transformation $f : \mathbf{X} \to \mathbf{X}$ and will be denoted as $(\mathbf{X}, d, f)$. Let I be some set of input symbols. Then a *dynamical system with input* is a metric space $(\mathbf{X}, d)$ and a transformation $T : \mathbf{X} \times I \to \mathbf{X}$. We denote the domain and the range of a transformation $f : \mathbf{X} \to \mathbf{X}$ by Dom_f and Ran_f, respectively. An *iterated function system* (IFS) consists of a metric space $(\mathbf{X}, d)$ and a finite number N of transformations $f_n : \mathbf{X} \to \mathbf{X}$ and will be denoted as $(\mathbf{X}, d; \{f_n \mid 1 \leq n \leq N\})$.

To define the class of hyperbolic IFSs we need some more definitions about sequences of points of metric spaces:

A sequence $\{x_n\}_{n=1}^{\infty}$ of points of a metric space $(\mathbf{X}, d)$ *converges* to a point $x \in \mathbf{X}$, if for any $\epsilon > 0$ there exists a positive integer N such that $\forall n > N : d(x_n, x) < \epsilon$. The point $x \in \mathbf{X}$ is then called the *limit* of the sequence. Let $S \subset \mathbf{X}$ be a subset of a metric space $(\mathbf{X}, d)$, then S is *compact* if each infinite sequence $\{x_n\}_{n=1}^{\infty}$ in S contains a subsequence with a limit in S. A sequence $\{x_n\}_{n=1}^{\infty}$ of points of a metric space $(\mathbf{X}, d)$ is said to be a *Cauchy sequence* if for any $\epsilon > 0$ there exists a positive integer N such that $\forall n, m > N : d(x_n, x_m) < \epsilon$. A metric space $(\mathbf{X}, d)$ is *complete* if each Cauchy sequence $\{x_n\}_{n=1}^{\infty}$ in $\mathbf{X}$ has a limit $x \in \mathbf{X}$.

A transformation $f : \mathbf{X} \to \mathbf{X}$ on a metric space $(\mathbf{X}, d)$ is a *contractive transformation* or *contraction* if there exists a constant $0 < s < 1$ such that

$$\forall x, y \in \mathbf{X} : d(f(x), f(y)) \leq s \cdot d(x, y). \tag{1}$$

Definition 1. A *hyperbolic* IFS consists of a complete metric space $(\mathbf{X}, d)$ and a finite set of contractive transformations $\{f_n : \mathbf{X} \to \mathbf{X} \mid 1 \leq n \leq N\}$.

A transformation $f : \mathbb{R}^n \to \mathbb{R}^n$ such that $f(x_1, x_2, \ldots, x_n) = (a_{11}x_1 + \ldots + a_{1n}x_n + a_1, \ldots, a_{n1}x_1 + \ldots a_{nn}x_n + a_n)$, where the $a_i, a_{ji} \in \mathbb{R}$ $(i, j = 1, \ldots, n)$, is said to be an (n–dimensional) *affine transformation*.

Let $S \subset \mathbf{X}$ be a subset of the metric space $(\mathbf{X}, d)$, then S is *bounded* if there exists a point $a \in \mathbf{X}$ and a real number $r > 0$ such that $\forall x \in S : d(a, x) < r$. Let d be some positive integer. The set $[0, 1]^d$ denotes the bounded subset $\{(x_1, x_2, \ldots, x_d) \in [0, 1]^d \mid 0 \leq x_i \leq 1, i = 1, 2 \ldots, d\}$.

Throughout the paper we will use the class of dynamical systems $([0, 1]^d, |x - y|, f)$, with the metric space $([0, 1]^d, |x - y|)$ where $|x - y|$ denotes the Euclidean distance between two points $x, y \in [0, 1]^d$ and the transformation $f : [0, 1]^d \to [0, 1]^d$. Note that each bounded subspace of the metric space $(\mathbb{R}^d, |x - y|)$ is a complete metric space, so is $([0, 1]^d, |x - y|)$ (see for instance [1]). Two subsets $S \subset [0, 1]^d$ and $R \subset [0, 1]^d$ of the metric space $([0, 1]^d, |x - y|)$ are *disjoint* if $\forall x \in S : \forall y \in R : |x - y| > 0$.

Examining the dynamical behavior of the transformation f we are interested in periodic points, i.e. points which are invariant under (possibly repeated) applications of f. The *orbit* of a point $x \in [0, 1]^d$ is the sequence $\{f^n(x)\}_{n=0}^{\infty}$.

Definition 2. $x \in [0, 1]^d$ is a *n–periodic point* of f iff $\exists n \in \mathbb{N} : f^n(x) = x \wedge (\neg \exists\, m \in \mathbb{N} : m < n \wedge f^m(x) = x)$.

The orbit of a n–periodic point of f is called a *cycle* with *period* n. A *fixed point* is a 1–periodic point. Periodic points may be attractive or repulsive. Attractive ones are stable with respect to minor perturbations, i.e., the system returns to the periodic point, whereas repulsive ones are unstable, i.e., perturbations may send the system to another periodic point. The n–periodic attractive points form a class of *attractors*.

2 Dynamics of Simple Recurrent Neural Networks

The simplest RNN consists of a single unit, whose output is propagated back to itself. Figure 1 shows such a unit with threshold θ, a recurrent connection with weight w, and the activation function $\sigma\colon \mathbb{R} \to [0,1]$:

$$\sigma(x) = \mathrm{sat}(wx + \theta + i), \tag{2}$$

where $\mathrm{sat}(x)$ represents the linear saturation function:

$$\mathrm{sat}(x) = \begin{cases} 0 \text{ if } & x < 0 \\ x \text{ if } & 0 \le x \le 1 \\ 1 \text{ if } & x > 1 \end{cases}$$

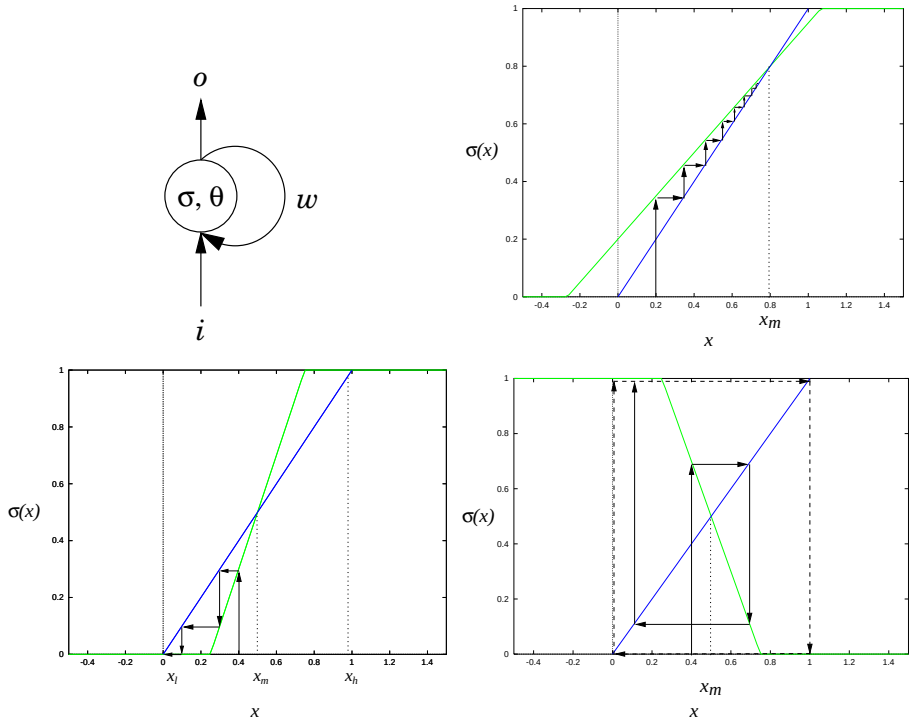


Fig. 1. A simple RNN with threshold θ, activation function σ, input i, output o, and weight w on the recurrent connection. Depending on the parameters p_1 and p_2 (which are determined by w and θ) the behavior of the dynamical system can be classified according to the three examples given here: The case of a single attractor of σ at x_m for the parameter values $p_1 = 3/4$ and $p_2 = -1/5$. The case of a repulsive fixed point at x_m and two attractors at x_l and x_h of σ for the parameter values $p_1 = 2$ and $p_2 = 1/2$. The case of an attractive cycle with period 2 for the parameter values $p_1 = -2$ and $p_2 = 3/4$.

Let $p_1 = w$, $p_2 = -(\theta + i)/w$, and assume that the input $i \in \mathbb{R}$ is clamped[3]. Then the output behavior of such a simple RNN corresponds to a dynamical system $(\mathbb{R}, |x - y|, \sigma)$, where

$$\sigma(x) = \mathrm{sat}(p_1(x - p_2)). \tag{3}$$

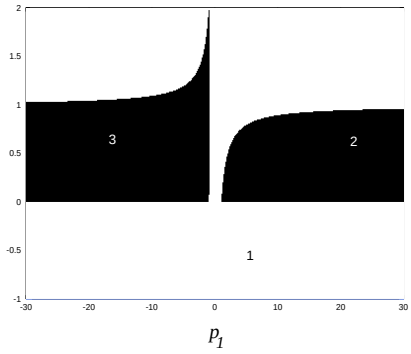
The activation function σ has at most three fixed points which are obtained as the intersections of $y = \sigma(x)$ with the line $y = x$. Let x_m denote the fixed point of σ, whose absolute value of the gradient is the largest compared to the gradients of all other fixed points of σ. We can distinguish three cases wrt. $\sigma'(x_m)$, where σ' denotes the first derivative of σ.

Case 1: If $-1 < \sigma'(x_m) < 1$ then σ admits one attractive fixed point x_m (see Figure 1). Whatever initial value x_0 is chosen the system eventually converges to x_m.

Case 2: If $\sigma'(x_m) > 1$ then σ has a repulsive fixed point at x_m and two attractive fixed points x_l and x_h, where $x_l < x_h$. The behavior of the system depends on the initial value x_0. If $x_0 < x_m$ then the system converges to the attractor x_l, whereas if $x_m < x_0$ then the system converges to the attractor x_h (see Figure 1).

Case 3: If $\sigma'(x_m) < -1$ then the system has a repulsive fixed point at x_m and an attractive cycle with period 2. In other words, the system has a 2–periodic attractor. Whatever initial value x_0 is chosen the system eventually converges to this cycle (see Figure 1).

As the transformation σ in equation (3) depends on the parameters p_1 and p_2, the three cases may be graphically depict wrt. these parameters. The figure on the right hand side shows the three regions in the space opened up by p_1 p_2 and p_2 which correspond to the three cases defined above. The various regions can be determined analytically by the different values of the derivative of σ at x_m, that is



$$\sigma'(x) = \begin{cases} 0 & \text{if} \quad p_1(x - p_2) < 0 \\ p_1 & \text{if} \quad 0 \le p_1(x - p_2) \le 1 \\ 0 & \text{if} \quad p_1(x - p_2) > 1 \end{cases}$$

Regions 1 and 2 are bounded by values p_1 and p_2 where $p_1(x - p_2) = 0$ and $p_1(x - p_2) = 1$ and $\sigma'(x) = p_1 > 1$. Regions 1 and 3 are bounded by values p_1 and p_2 where $p_1(x - p_2) = 0$ and $p_1(x - p_2) = 1$ but $\sigma'(x) = p_1 < -1$. With the fixed point condition $\sigma(x) = x$ we find the borderlines

$$p_2 = 0, \quad p_2 = 1 - \frac{1}{p_1} \quad \text{for } p_1 > 1 \text{ and} \quad p_2 = 0, \quad p_2 = 1 - \frac{1}{p_1} \quad \text{for } p_1 < -1$$

in the parameter space, respectively.

According to equation (2) and equation (3) the input value i to a self–recurrent neuron computing the linear saturation function can be understood as changing the value of p_2. Hence, the input cannot steer the neurons behavior arbitrarily, e.g. it is not possible to steer the neuron from region 2 (one repulsive fixed point, two attractive fixed points) to region 3 (one repulsive fixed point, one attractive cycle with period 2). Using additional neurons (without self–

[3] This parameter transformation relates our results concerning the dynamics of iterated linear saturation functions to the results for iterated continuous sigmoidal functions described in [7].

recurrent connections) however, will enable the input i to choose a value for the parameter p_1 from a determined set of values.

3 Setting up a Counter

By using linear saturation instead of sigmoidal activation functions it is possible to avoid some difficulties occurring in the design of a RNN that implements a counter as described and discussed in [4] and [7]. This is due to the fact that the inverse mapping of an affine transformation (if it exists) is again an affine transformation[4].

To implement a counter the functions f_{inc} which implements the incrementing and f_{dec} which implements the decrementing of the counter, respectively, must fulfill the conditions

$$\forall x \in \mathrm{Dom}_{f_{inc}} : \forall n \in \mathbb{N} : f_{dec}(f_{inc}^{n+1}(x)) = f_{inc}^{n}(x) \quad \text{and} \tag{4}$$

$$\forall x \in \mathrm{Dom}_{f_{inc}} : \forall m, n \in \mathbb{N} : m \neq n \rightarrow f_{inc}^{m}(x) \neq f_{inc}^{n}(x). \tag{5}$$

These conditions can easily be met by the affine transformations

$$f_{inc}(x) = a_{inc}x + b_{inc} \quad \text{and} \quad f_{dec}(x) = a_{dec}x + b_{dec}$$

which range over a bounded subset $[l, r]$ of $\mathbb{R}$. A neuron computing $\mathrm{sat}(wx+\theta)$ behaves over the subset

$$\left[-\frac{\theta}{w}, \frac{1-\theta}{w}\right] \quad \text{if} \quad w > 0 \quad \text{and} \quad \left[\frac{1-\theta}{w}, -\frac{\theta}{w}\right] \quad \text{if} \quad w < 0$$

as an element computing the affine transformation $wx+\theta$. Hence, if the bounded subset $[l, r]$ is chosen accordingly, a neuron can be designed which implements incrementing and decrementing of a counter, respectively.

To ensure that $f_{inc}^{n}(x) \in [l, r]$ for all $n \in \mathbb{N}$ and for all $x \in [l, r]$, the parameters a_{inc} and b_{inc} must fulfill, if $a_{inc} > 0$:

$$l(1 - a_{inc}) \leq b_{inc} \leq r(1 - a_{inc}) \quad \text{and} \quad 0 \leq a_{inc} \leq 1, \tag{6}$$

and if $a_{inc} < 0$:

$$l - a_{inc}r \leq b_{inc} \leq r - a_{inc}l \quad \text{and} \quad -1 \leq a_{inc} \leq 0.$$

Together with condition (5) follows $|a_{inc}| < 1$. Hence, the transformation f_{inc} is contractive on $[l, r]$ according to equation (1) and the neuron implementing f_{inc} shows region–1–behavior ($-1 < p_1 = w = a_{inc} < 1$).

Moreover, from condition (4) follows that f_{dec} is given as the inverse transformation of f_{inc}. Hence, the parameters a_{dec} and b_{dec} are determined as follows:

$$a_{dec} = \frac{1}{a_{inc}}, \qquad b_{dec} = -\frac{b_{inc}}{a_{inc}}.$$

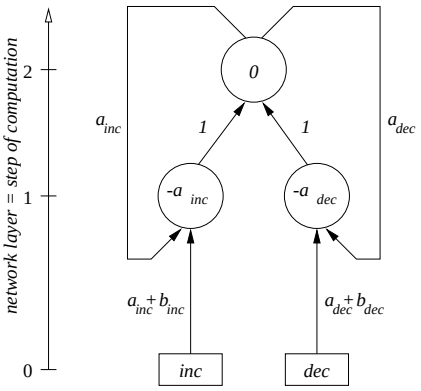
If a_{inc} is considered to be positive ($0 < a_{inc} < 1$), from condition (6) with $-\frac{\theta}{w} = -\frac{b_{inc}}{a_{inc}} \leq l$ and $r \leq \frac{1-\theta}{w} = \frac{1-b_{inc}}{a_{inc}}$ follows

$$0 \leq b_{inc} \leq 1 - a_{inc}$$

With the substitutions $p_1 = w = a_{dec} = \frac{1}{a_{inc}}$ and $p_2 = -\frac{\theta}{w} = -\frac{b_{dec}}{a_{dec}} = b_{inc}$ for the neuron implementing f_{dec} holds $0 \leq p_2 \leq 1 - \frac{1}{p_1}$, i.e. the neuron shows region–2–behavior.

[4] Whereas the inverse mapping of a sigmoidal transformation is not a sigmoidal transformation.

Whereas, if a_{inc} is considered to be negative ($-1 < a_{inc} < 0$), very similar arguments lead to the conclusion that the neuron implementing f_{dec} shows region–3–behavior. The resulting RNN is shown in the figure on the right hand side. For switching between region–1–behavior and region–2–behavior (or region–3–behavior if $-1 < a_{inc} < 0$) two additional neurons are used. The output of the first unit is 0 in case the counter has to be decremented ($inc = 1$, $dec = 0$) and the output of the second unit is 0 in case the counter has to be incremented ($inc = 0$, $dec = 1$).



4 Extending Counters to Binary Stacks

The RNN described in the previous section can easily be extended to implement another important computational task: operations on a binary stack. To imple-

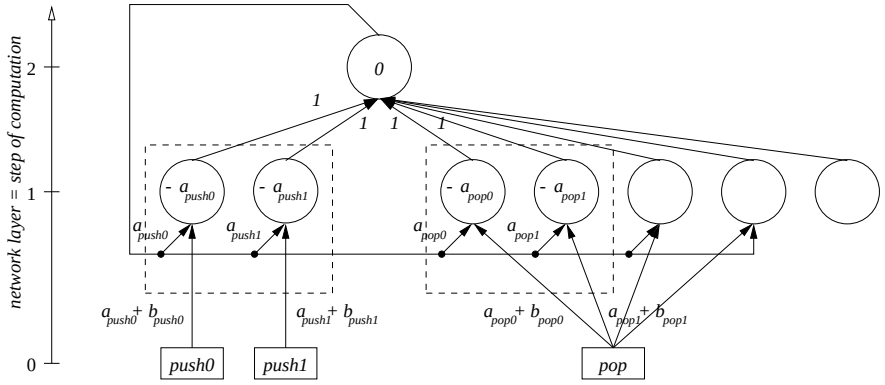


Fig. 2. A simple RNN implementing a binary stack.

ment the *push* operations the contractive affine transformations $f_{push0}(x) = a_{push0}x + b_{push0}$ for pushing the value 0 and $f_{push1}(x) = a_{push1}x + b_{push1}$ for pushing value 1, respectively, are used. To implement the *pop* operation an inverse transformation f_{pop} is used which must additionally fulfill:

$$\forall x \in \mathrm{Dom}_{f_{push0}} : f_{pop}(f_{push0}(x)) = x \quad \text{and}$$
$$\forall x \in \mathrm{Dom}_{f_{push1}} : f_{pop}(f_{push1}(x)) = x.$$

Thus, the ranges of f_{push0} and f_{push1} must be disjoint:

$$\mathrm{Ran}_{f_{push0}} \cap \mathrm{Ran}_{f_{push1}} = \emptyset.$$

Figure 2 sketches the structure of a RNN implementing a binary stack. Here, we do not want to give a particular instantiation of the parameters a_{push0}, b_{push0}, etc. fulfilling the above conditions. See Section 6 for details on how to compute appropriate values for these parameters[5].

[5] We added some additional neurons which are necessary for implementing the particular kind of *pop* operation which is the inverse operation of each of the *push* operations. In this paper, we do not discuss this construction any further.

5 Iterated Function Systems and Stacks

In this section we show how the dynamics of a wide class of RNNs can be understood in terms of IFSs. Furthermore, we show that a RNN performs stack operations if the corresponding IFS meets certain conditions.

A layered RNN corresponds to a dynamical system with input $([0,1]^d, |x-y|; T)$, where $T : [0,1]^d \times I \to [0,1]^d$. If we consider the set I of input values to be finite, this RNN can be transformed into an equivalent IFS $([0,1]^d, |x-y|; \{f_i \mid f_i = T(i) \wedge i \in I\})$ where $T(i)$ represents the mapping $T(x_1, x_2, \ldots, x_d, i)$ for some constant i (as also presented in [9]).

Theorem 3. *Let $(\mathbf{X}, d; F^c)$ denote an IFS, where $F^c = \{f_i \mid f_i : \mathbf{X} \to \mathbf{X} \wedge i \in I\}$ and I is a finite set of input symbols. The IFS simulates the set of operations $\{push(s,i) \mid i \in I\}$ on a stack s if*

(i) $\forall i \in I : f_i \in F^c$ is a bijective contraction on $\mathbf{X}$ and
(ii) the subsets $Ran_{f_i} \subset \mathbf{X}$ are disjoint.

Proof. Each hyperbolic IFS has an unique attractor (for a proof see [1], theorem 7.1 on page 81). For each finite sequence $w = w_1 w_2 \ldots w_n$ of inputs $w_j \in I$ $(1 \leq j \leq n)$ there is a mapping ϕ from w onto $f_{w_n}(\ldots(f_{w_2}(f_{w_1}(x)))\ldots)$ for an arbitrary but constant choice of $x \in \mathbf{X}$ (page 123, theorem 2.1 in [1]). Additionally, the mapping ϕ is one–to–one if the subsets $\mathrm{Ran}_{f_i} \subset \mathbf{X}$ are disjoint (for a proof see [1], theorem 2.2 on page 125). Finally, consider a stack s with content w. The operation $push(s,u)$ for some $u \in I$ is mapped to the application of f_u to $f_{w_n}(\ldots(f_{w_2}(f_{w_1}(x)))\ldots)$. □

Theorem 4. *Let $(\mathbf{X}, d; F^e)$ denote an IFS, where $F^e = \{g_i \mid g_i : \mathbf{X} \to \mathbf{X} \wedge i \in I\}$ and I is a finite set of input symbols. The IFS simulates the set of operations $\{pop(s,i) \mid i \in I\}$ on a stack s in $\mathbf{X}$ if for each $g_i \in F^e$ there exists a $g_i^{-1} : \mathbf{X} \to \mathbf{X}$ such that*

(i) $\forall x \in Ran_{g_i^{-1}} : g_i(g_i^{-1}(x)) = x$,
(ii) $\forall i \in I : g_i^{-1}$ is a bijective contraction on $\mathbf{X}$ and
(iii) the subsets $Ran_{g_i^{-1}} \subset \mathbf{X}$ are disjoint.

Proof. From *(ii)* and *(iii)* it follows that $(\mathbf{X}, d; F^c)$ where $F^c = \{g_i^{-1} \mid g^i \in F^e\}$ simulates the operations $push(s,i)$ on a stack s. Additionally, it follows *(i)* from that for all $w = w_1 w_2 \ldots w_n$ $(w_j \in I)$ and $u \in I$:
$g_u(g_u^{-1}(g_{w_n}^{-1}(\ldots(g_{w_2}^{-1}(g_{w_1}^{-1}(x)))\ldots))) = g_{w_n}^{-1}(\ldots(g_{w_2}^{-1}(g_{w_1}^{-1}(x)))\ldots)$. □

Theorem 3 and Theorem 4 allow us to derive a procedure for analyzing arbitrary layered RNN and checking whether they perform stack operations wrt. some input symbols.

Procedure 1.

step1. Determine the (finite) set I of input symbols and transform the layered RNN into the corresponding IFS.
step2. Check the domains of the transformations for contractive and expansive sub–domains.
step3. In case of a contractive sub–domain check whether the transformation is a one–to–one mapping over this domain.
step3. Generate new IFSs for each corresponding sub–domain.
step4. Analyze the transformations that are contractive in a certain sub–domain: check if they range over disjoint subsets (of the sub–domain).

step5. Analyze the transformations which are expansive in a certain sub–domain: check whether the corresponding inverse contractive one–to–one transformations ranging over disjoint subspaces (of the sub–domain) exist.

If we consider RNN using linear saturation as activation functions – as the RNNs described in Section 2 up to 4 – some tests can be carried out to compute parts of the given procedure:

one–to–one: Since the linear saturation function is one–to–one iff the activation is in $[0,1]$ and within this activation interval the neuron computes an affine transformation, the bounds for this behavior can be calculated from the net parameters, easily.

contractivity: If the absolute value of the largest eigenvalue of the operator matrix is smaller than 1, then the affine transformation is a contraction.

expansion: If the absolute value of the largest eigenvalue of the operator matrix is greater than 1, then the affine transformation is an expansion.

disjoint subspaces: Check the bounding surfaces of the corresponding hypercubes. If there are no points of intersection and none of the hypercubes contain another one completely, then the subspaces are disjoint.

As a result of Procedure 1 we may moreover realize that, since the inverse function of a sigmoidal function is not computable by a neuron, RNNs using sigmoidal activation functions can not implement stack operations as described above. By a similar observation the result provided in [13], stating that RNNs using linear saturation activation functions can simulate an universal Turing machine, can not be adapted to RNNs using sigmoidal activation functions, since the proof given in [13] is based on a similar stack construction as described here, and thus based on the fact that a neuron can compute the inverse function of another neuron.

6 Recognizing Dyck Languages by Recurrent Neural Networks

The results from the previous section can also be used to *design* networks which perform computational tasks. To illustrate this we design a simple RNN that recognizes a context free language. More precisely, we construct for any Dyck language $\mathcal{D}(n)$ a corresponding RNN $\mathcal{N}_{\mathcal{D}(n)}$ that recognizes $\mathcal{D}(n)$.

Consider the class of Dyck languages $\mathcal{D}(n)$ (also known as the bracket languages). Let, for $n \geq 1$, $\Sigma_n = \{a_1, b_1, a_2, b_2, \ldots, a_n, b_n\}$, $V = \{S\}$ and $P_n = \{S \to Sa_iSb_iS \mid i = 1, \ldots, n\} \cup \{S \to \epsilon\}$ be the set of terminal symbols, nonterminal symbols and productions, respectively.

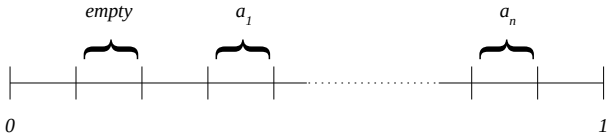
Obviously, in terms of operations on an n–ary stack s_n, a word $c = c_1c_2 \ldots c_m$ ($c_j \in \Sigma_n$) of $\mathcal{D}(n)$ can be recognized by performing the following algorithm on c:

Algorithm 1.

step1. $j := 0$, $s_n := \emptyset$
step2. if $j = m$ stop
step3. $j := j + 1$
step4. if $c_j = a_i$ for some i ($1 \leq i \leq n$) then $push(s_n, a_i)$, goto step2
step5. if $c_j = b_i$ and $top(s_n) = a_i$ for some i ($1 \leq i \leq n$) then $pop(s_n)$, goto step2
step6. $s_n := push(s_n, a_i)$ (for some $1 \leq i \leq n$), stop

The string is accepted iff the algorithm has stopped and the stack is empty. Note that, whenever the algorithm stops in step6 the stack is not empty, i.e., the string is not accepted.

To construct a RNN that recognizes the language $\mathcal{D}(n)$, we define a metric space containing representations of all possible stack values and an IFS that performs the corresponding operations on the metric space. To keep the resulting network small, consider the metric space $([0,1],|x-y|)$. The space $[0,1]$ is partitioned in compact subspaces as illustrated here:



A subspace named a_i (for $1 \leq i \leq n$) represents the set of stack values where the top element is the symbol a_i. These subspaces are considered to have the same structure as the interval $[0,1]$, i.e., the subspace named a_j of the subspace named a_i represents the set of stack values where the first element is the symbol a_i and the second element is the symbol a_j, etc. The subspace named *empty* represents the empty stack. To keep the construction simple, we consider all intervals to be of equal length[6]. Hence, the $[0,1]$ interval is partitioned into $2n+3$ subintervals of length $\frac{1}{2n+3}$. According to this partitioning of the state space the (contractive) transformations f_i mapping the interval $[0,1]$ to $[\frac{2i+1}{2n+3}, \frac{2i+2}{2n+3}]$, respectively, are defined as

$$f_i(x) = \mathrm{sat}(\omega_1 x + \theta_i) = \mathrm{sat}(\frac{1}{2n+3}x + \frac{2i+1}{2n+3}).$$

The corresponding inverse transformations mapping the interval $[\frac{2i+1}{2n+3}, \frac{2i+2}{2n+3}]$ to $[0,1]$, respectively, are given by

$$g_i(x) = \mathrm{sat}(\omega_2 x + \theta_{n+i}) = \mathrm{sat}((2n+3)x - (2i+1)).$$

Note that, for all $x < \frac{2i+1}{2n+3}$ $g_i(x) = 0$ and for all $x > \frac{2i+2}{2n+3}$ $g_i(x) = 1$. Now, the metric space given above together with these set of functions define the IFS $([0,1],|x-y|;\{f_i \mid 1 \leq i \leq n\} \cup \{g_i \mid 1 \leq i \leq n\})$. Algorithm 1. can be reformulated in terms of this IFS as follows:

Algorithm 2.

step1. $j := 0$, $s \in [\frac{1}{2n+3}, \frac{2}{2n+3}]$
step2. if $j = m$ stop
step3. $j := j + 1$
step4. if $c_j = a_i$ for some i $(1 \leq i \leq n)$ then $s := f_i(s)$, goto step2
step5. if $c_j = b_i$ and $s \in [\frac{2i+1}{2n+3}, \frac{2i+2}{2n+3}]$ for some i $(1 \leq i \leq n)$ then $s := g_i(s)$, goto step2
step6. $s := g_i(s)$, goto step2

The string is accepted iff the algorithm has stopped and $s \in [\frac{1}{2n+3}, \frac{2}{2n+3}]$. Note, that although the algorithm does not stop in step6 the value of s can never be an element of $[\frac{1}{2n+3}, \frac{2}{2n+3}]$ again.

Consider the word $c = c_1 c_2 \ldots c_m$ being sequentially fed into the system such that the symbol c_j $(1 \leq j \leq m)$ is input at time j. Then the four layered RNN $\mathcal{N}_{\mathcal{D}(n)}$ implements Algorithm 2. Note, that 4 computation steps are done within one time step. Hence, the RNN recognizes the Dyck language within linear time of the length of the input word c.

[6] The interval between two named intervals is left unnamed to avoid identification conflicts. E.g., the representation of an infinite sequence $a_i a_n a_n \ldots$ would be undistinguishable from the representation of a_{i+1} (for $1 \leq i \leq n-1$), otherwise.

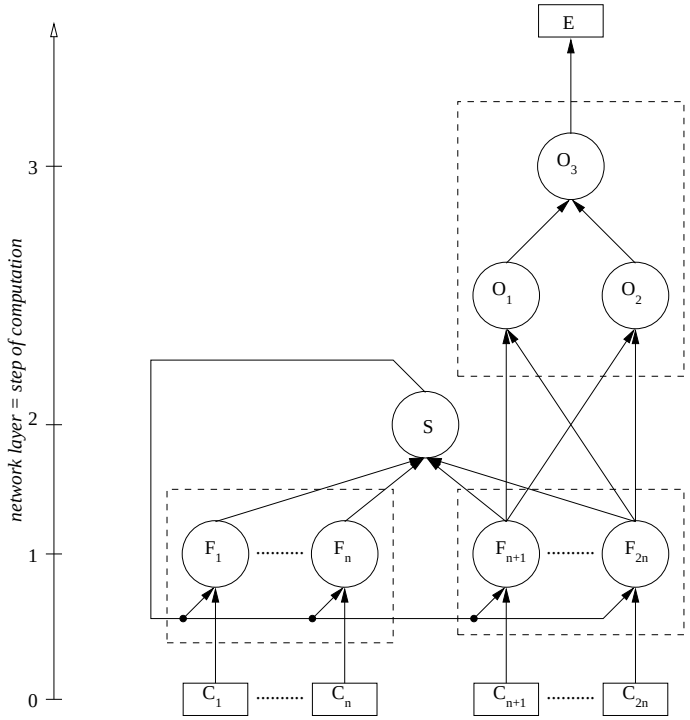


Fig. 3. A first–order RNN that recognizes the Dyck language $\mathcal{D}(n)$.

The RNN $\mathcal{N}_{\mathcal{D}(n)}$ consists of $2n+4$ neurons which are connected according to Figure 3. All neurons compute the linear saturation function $\mathrm{sat}(x)$ where x represents the weighted sum of its input plus some threshold.

Let $F_1, \ldots, F_{2n}$, S and O_1, O_2, O_3 denote the neurons of $\mathcal{N}_{\mathcal{D}(n)}$ as depicted in Figure 3. Let W_{KL} denote the weight of the edge connecting an object K to some object L , and T_N the threshold of neuron N . Furthermore, let C_j denote the input vector $(C_1^j, \ldots, C_n^j, C_{n+1}^j, \ldots, C_{2n}^j)$ at step j , where

$$C_i^j = \begin{cases} 1 & \text{if} \quad c_j = a_i \quad \text{and} \quad (1 \leq i \leq n) \\ 1 & \text{if} \quad c_j = b_{i-n} \quad \text{and} \quad (n+1 \leq i \leq 2n) \\ 0 & \text{else} \end{cases}$$

The weights and thresholds of the neurons realizing the *push* functions are given by $W_{C_k F_k} = \omega_1 + \theta_k$, $W_{SF_k} = \omega_1$, $W_{F_k S} = 1$, $T_{F_k} = -\omega_1$ (for $1 \leq k \leq n$), and the weights and thresholds of the neurons realizing the *pop* functions are given by: $W_{C_k F_k} = \omega_2 + \theta_k$, $W_{SF_k} = \omega_2$, $W_{F_k S} = 1$, $T_{F_k} = -\omega_2$ (for $n+1 \leq k \leq 2n$). The neuron whose activation represents the stack value has threshold 1 and for the neurons computing the output the weights $W_{F_k O_1} = \omega_2$, $W_{F_k O_2} = -\omega_2$, $W_{O_1 O_3} = W_{O_1 O_3} = 1$ and thresholds $T_{O_1} = 0$, $T_{O_2} = 3$, $T_{O_3} = -1$ were chosen to output 1 iff the stack value is element of the interval representing the empty stack.

7 Discussion

Within the first part of the paper we gave a formal analysis of the dynamical behavior of a RNN with a single self–recurrent unit in the hidden layer that com-

putes a linear saturation activation function. This analysis provided the basis for the construction of a RNN implementing a counting task as discussed in [4] and [7]. In contrast to the networks described in these papers the computational depth of the RNN presented here is not restricted. Moreover, we described a generalization of the counter network to a RNN that implements a binary stack and its *push* and *pop* operations. The basic idea behind this generalization is a mapping between a layered RNN and a corresponding IFS. The correspondence is explicitly stated in the second part of the paper, where we gave conditions for an IFS such that the IFS and the corresponding RNN implement stack operations wrt. some input symbols. We derived a procedure to analyze any layered RNN independent of the particular used activation function. Such an analysis may provide good comprehension on how behavior of RNNs can be characterized in terms of classical computation no matter whether the structure of the RNN is result of a training process, a particular design method or even the modeling of parts in real brains. Moreover, the IFS–RNN correspondence can also be used as a design principle for RNNs which implement computational tasks that are given in terms of stack operations. As an example we showed how to design a class of RNN $\mathcal{N}_{\mathcal{D}(n)}$ such that for each n, $\mathcal{N}_{\mathcal{D}(n)}$ recognizes the Dyck language $\mathcal{D}(n)$.

References

1. M. Barnsley: *Fractals Everywhere.* CA: Academic Press, San Diego, 1988.
2. M. Casey. The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction. *Neural Computation*, 8(6):1135–1178, 1996.
3. J.L. Elman: *Finding Structure in Time.* Cognitive Science, 14, pp. 179–211, 1990.
4. J. Wiles and J. Elman: *Learning to Count without a Counter: A Case Study of Dynamics and Activation Landscapes in Recurrent Networks.* Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society, Cambridge, MA, MIT Press, 1995.
5. C.W. Omlin and C.L. Giles. Constructing deterministic finite-state automata in recurrent neural networks. *Journal of the ACM*, 45(6):p. 937, 1996.
6. P. Grünwald and M. Steijvers: *A Recurrent Network that performs a context-sensitive prediction task.* Proceedings of the Eighteenth Annual Conference of the Cognitive Science Society, Morgan Kauffman, 1996.
7. S. Hölldobler and Y. Kalinke and H. Lehmann: *Designing a Counter: Another Case Study of Dynamics and Activation Landscapes in Recurrent Networks.* LNAI 1303, Proceedings of the KI97: Advances in Artificial Intelligence, Springer, pp. 313–324, 1997.
8. M.I. Jordan: *Attractor Dynamics and Parallelism in a Connectionist Sequential Machine.* Proceedings of the Annual Conference of the Cognitive Science Society, pp. 531–546, 1986.
9. J.F. Kolen: *Exploring the Computational Capabilities of Recurrent Neural Networks.* PhD Thesis, Ohio State University, 1994.
10. H. Nakahara and K. Doya: *Dynamics of Attention as Near Saddle–Node Bifurcation Behavior.* In: D.S. Touretzky and M.C. Mozer and M.E. Hasselmo (eds.): *Advances in Neural Information Processing Systems*, Volume 8, Neural Information Processing Systems 1995, MIT Press, 1996.
11. J.B. Pollack: *Recursive Distributed Representations.* Artificial Intelligence, **46**, pp. 77–105, 1990.
12. P. Rodriguez and J. Wiles: *Recurrent Neural Networks Can Learn to Implement Symbol-Sensitive Counting.* Proceedings of the 11th Conference on Neural Information Processing Systems, Denver CA, USA, 1998 (to appear).
13. H. Siegelmann and E.D. Sontag. *Turing Computability with Neural Nets.* Applied Mathematics Letters, **4**(6), pp.77-80, 1991.
14. A. Stolcke and D. Wu: *Tree Matching with Recursive Distributed Representations.* International Computer Science Institute, Berkeley, Technical Report TR-92-025, 1992.