Analysis and implementation of the paper - On the Computation Power of Neural Nets

Omkar Zade
D-INFK, ETH Zürich
omzade@student.ethz.ch

Abstract

As a part of the course project we analyze and implement the paper - On the Computational Power of Neural Nets (Siegelmann and Sontag, 1995). The paper is a famous result on the universality of *first-order* recursive nets, and has important consequences regarding decidability of such nets. We complete a constructive proof, by providing an implementation of the main theorem of the paper. In doing so, we model the equations in the proof as efficient matrix-vector computations.

1 Introduction

The paper provides an existential and semi-constructive proof that any Turing machine can be simulated by a recurrent neural network of finite size with only rational weights. This result has the implication that any partial recursive function can be computed on RNNs and that RNNs, as a model of computation, are as powerful as (deterministic) Turing machines.

2 Our contributions

We implement Theorem 2 (a) from the paper using `numpy` linear algebra routines, and hence complete a full constructive proof of the theorem. This is to our knowledge a first attempt at implementing the paper. The computations in the original proof are specified in terms of single neurons. We model these computations as efficient matrix-vector computations for an entire layer of neurons, as is conventional in current neural network formulations.

Secondly, Lemma 5.1 claims the existence of vectors $v_1, v_2, \ldots, v_{2^t} \in \mathbb{Z}^{t+2}$ and constants $c_1, c_2, \ldots, c_{2^t} \in \mathbb{Z}$ such that equations (10) and (11) hold. We show how to efficiently compute these parameters which are ultimately used as weights in the neural network, by providing an algorithm.

The rest of the paper is organized as follows: we briefly explain the setting and formalisms in section 3. In section 4 we discuss the intermediate step of modeling a Turing machine with a dynamical system over rationals, and explain our implementation. In section 5, we see the final step of modelling it as a sigma-processor net (i.e. a composition of saturated-affine maps).

We only implement the linear time simulation - one step of Turing machine by four steps (layers) of the neural network. The paper also proves a real time simulation (Theorem 2 (b)), with more involved encodings, which we do not cover.

3 Setting and formalisms

3.1 σ-processor nets

A recurrent first-order neural network, or σ-processor net is a network of processors such that every processor's state is updated by the equation of type:

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

i.e. a saturated affine combination of all neurons $j = 1, \ldots, N$, M inputs u_j and a bias. σ is the saturated linear function i.e. $\sigma(x) = 0, x, 1$ depending on whether $x < 0, 0 \leq x \leq 1, x > 1$, respectively. We will consider nets without input and hence the terms u_j vanish. Hence, we have in matrix-vector notation:

$$\mathcal{F}(x) = \sigma(Ax + c) \tag{2}$$

where $A \in \mathbb{Q}^{N\times N}$ and $x, c \in \mathbb{Q}^N$

3.2 p-stack Turing machine

The paper considers simulation of p-stack Turing machines, as they are equivalent to $p/2$-tape Turing machines (Hopcroft and Ullman, 1979). It consists of a finite control and p binary stacks of unbounded length. Input is written on $stack_1$ at the beginning of computation, and there are $p-1$ working stacks. At the end of computation, when the machine reaches a special halting state, output is written to $stack_1$. We give the description of the machine which is relevant for the implementation, and refer the reader to Section 3.1 and 4.1 of the original paper for details.

A p-stack machine $\mathcal{M}$ is a $(p+4)$ tuple

$$(S, s_I, s_H, \theta_0, \theta_1, \dots, \theta_p)$$

where $S = \{0, 1, ..., s\}$ is set of finite states (control), $s_I = 0, s_H = 1 \in S$ are initial and halting states, θ_0 is a function that computes the next state defined as the map:

$$\theta_0 : S \times \{0,1\}^{2p} \to S$$

θ_i are functions that compute the next stack operation "no-op", "push0", "push1" or "pop"

$$\begin{aligned}\theta_i :& S \times \{0,1\}^{2p} \to \{(1,0,0), (1/4, 0, 1/4),\\ & (1/4, 0, 3/4), (4, -2, -1)\}\end{aligned}$$

for $i = 1 \dots p$

Instantaneous description of $\mathcal{M}$ is $\mathcal{X} := S \times \mathcal{C}^p$, where $\mathcal{C}$ is the "Cantor 4-set"

The complete dynamics map of $\mathcal{M}$:

$$\mathcal{P} : \mathcal{X} \to \mathcal{X}$$

is defined as:

$$\begin{aligned}\mathcal{P} :=&[\theta_0(s, a_1, .., a_p, b_1, .., b_p),\\ &\theta_1(s, a_1, .., a_p, b_1, .., b_p) \cdot (q_1, a_1, 1),\\ &\vdots\\ &\theta_p(s, a_1, .., a_p, b_1, .., b_p) \cdot (q_p, a_p, 1)]\end{aligned}$$

3.3 Discrete time dynamical system

A discrete time dynamical system (over rationals) is a specified by a dynamics map

$$\mathcal{F} : \mathbb{Q}^N \to \mathbb{Q}^N$$

such that for each integer $t \geq 1$, one defines the *state at time t*, as the value obtained by recursively solving the equations:

$$x(1) := x^{\text{init}}$$

$$x(t+1) = \mathcal{F}(x(t))$$

4 Simulate a p-stack machine as a dynamical system over $\mathbb{Q}^{s+p}$

As an intermediate step of construction, we simulate $\mathcal{M}$ (as described in 3.2) by a dynamical system over $\mathbb{Q}^{s+p}$. A vector in this space is: $(x_1, \dots, x_s, q_1, \dots, q_p)$. The state vectors are unit vectors in $\mathbb{Q}^s$, with $\boldsymbol{e_0} = (0, 0, .., 0)$ indicating the initial state, $\boldsymbol{e_1} = (1, 0, .., 0)$ the halting state and $\boldsymbol{e_i}$, $i = 2..s$ indicating state $i \in S$. The $q_i's \in \mathbb{Q}$ encode content of the stacks, where the encoding function δ is used to map binary contents of the stack $a_1 \dots a_k$ to rationals in $[0,1]$ as

$$\delta[a_1 \dots a_k] = \sum_{i=1}^{k} \frac{2a_i + 1}{4^i}$$

. The motivation for this choice of encoding function is detailed in section 3.1 of the original paper.

Given the description of the Turing machine, we want to define a mapping

$$\mathcal{P} : \mathbb{Q}^{s+p} \to \mathbb{Q}^{s+p} \tag{3}$$

from the current state and stacks to the next state and stacks:

$$(x_1, .., x_s, q_1, .., q_p) \to (x_1^+, .., x_s^+, q_1^+, .., q_p^+)$$

Intuition. Assume the Turing machine $\mathcal{M}$ implements a partial recursive function

$$\phi : \{0,1\}^m \to \{0,1\}^n$$

for some $m, n \in \mathbb{N}$. If given input $\omega \in \{0,1\}^m$, if $\mathcal{M}$ computes $\phi(\omega) \in \{0,1\}^n$ in $\mathcal{T}$ time steps, the dynamical system $\mathcal{P}$, starting with $x^{\text{init}} = (0, 0..., 0, \delta[\omega], 0, 0, ..., 0)$ as the initial state, attains in $\mathcal{T}$ time steps the state $(1, 0..., 0, \delta[\phi(\omega)], 0, 0, ..., 0)$.

4.1 Construction

We define maps

$$\beta_{ij} : \{0,1\}^{2p} \to \{0,1\}$$

which intuitively is the "next state" map $(\beta_{ij}(a_1, .., a_p, b_1, ..b_p) = 1$ iff $\theta_0(j, a_1, .., a_p, b_1, ..b_p) = i)$ and

$$\gamma_{ij}^k : \{0,1\}^{2p} \to \{0,1\}$$

which intuitively are the next stack action maps (and hence derived from θ_i's). So for each of $i = 1..p$ stacks and $j = 0..s$ we have four such maps for $k = 1..4$ corresponding to "no-op", "push0", "push1" and "pop" on stack i when the machine is in state j.

Now, the next state is computed as

$$x_i^+ = \sum_{j=0}^{s} \beta_{ij}(a_1, ..., a_p, b_1, ..., b_p)x_j$$

.

Letting $x_0 := 1 - \sum_{j=1}^{s} x_j$, we implement this as a single matrix computation which updates all of $\boldsymbol{x} = (x_0, .., x_s)$ as

$$\boldsymbol{x}^+ = \boldsymbol{\beta}(a_1, .., a_p, b_1, ..b_p)\boldsymbol{x} \tag{4}$$

For the sake of representation, we interpret $(a_1, .., a_p, b_1, ..b_p)_{\text{base-2}}$ as an integer in $\{0, ..., 2^{2p} - 1\}$, and use it as the first index into $\boldsymbol{\beta}$. Hence, $\boldsymbol{\beta}$ is a tensor with shape $(2^{2p}, s, s+1)$ and $\boldsymbol{x}^+ = (x_1^+, .., x_s^+)$.

The next stack contents are computed by equations (9.1-9.4) in the original paper,

$$\begin{aligned} q_i^+ &:= (\sum_{j=0}^{s} \gamma_{ij}^1(a_1, .., a_p, b_1, ..b_p)x_j)q_i \\ &+ (\sum_{j=0}^{s} \gamma_{ij}^2(a_1, .., a_p, b_1, ..b_p)x_j)) \times (q_i/4 + 1/4) \\ &\vdots \end{aligned}$$

for $i = 1..p$. We also model these as a single matrix-vector update:

$$\begin{aligned} \boldsymbol{q}^+ = &(\boldsymbol{\gamma}^1(a_1, .., a_p, b_1, ..b_p)\boldsymbol{x}) \odot \boldsymbol{q} \\ &+ \boldsymbol{\gamma}^2(a_1, .., a_p, b_1, ..b_p)\boldsymbol{x}) \odot ((1/4)\boldsymbol{q} + 1/4) \\ &+ \dots \end{aligned}$$

where $\odot$ is the Hadamard product (element-wise multiplication) and $\boldsymbol{q} + 1/4$ performs *broadcasting* (addition of scalar to each element of vector $\boldsymbol{q}$) in `numpy`. The shape of $\boldsymbol{\gamma}$ is $(2^{2p}, p, s+1)$.

Plugging these two vector equations in a loop with a termination condition that $\boldsymbol{x} = \boldsymbol{e_1}$ (halting state) finishes the simulation. q_1 then contains the encoding of the output if the original Turing machine also terminates, otherwise the simulation runs forever. We note that each iteration of the loop is one step of the Turing machine, hence, this is a real time simulation.

5 Simulate $\mathcal{P}$ by a compositional σ-processor net

The final step of simulation is to simulate $\mathcal{P}$ [as defined in (3)] by a net. Precisely, we will model $\mathcal{P}$ as the composition of saturated affine maps [as defined in (2)]

$$\mathcal{P} = \mathcal{F}_1 \circ \mathcal{F}_2 \circ \mathcal{F}_3 \circ \mathcal{F}_4$$

The authors show in Lemma 5.1 that for any function over binary strings such as $\beta : \{0,1\}^t \to \{0,1\}$ (such as β_{ij} from section 4), there exist vectors $v_1, v_2, \dots, v_{2^t} \in \mathbb{Z}^{t+2}$ and constants $c_1, c_2, \dots, c_{2^t} \in \mathbb{Z}$ such that for each $d_1, d_2, \dots, d_t, x \in \{0,1\}$

$$\beta_{ij}(d_1, \dots, d_t)x = \sum_{r=1}^{2^t} c_r \sigma(v_r \cdot \mu) \tag{5}$$

where $\mu = [1, d_1, \dots, d_t, x] \in \mathbb{Z}^{t+2}$ and $\cdot$ is the dot product.

However, to actually implement the maps, we need a systematic way to find these constants, which is omitted from the original paper.

5.1 Solving for parameters

We provide an algorithm to compute v_r's and c_r's, given description of any binary function β_{ij} such that (5) holds.

The crux of the algorithm is solving for c_r's assuming a polynomial representation of β as in equation 12 in the paper:

$$\begin{aligned} \beta_{ij}(a_1, ..., a_p, b_1, ..., b_p)x = &c_1 + c_2 a_1 + \dots + c_{2p+1} b_p \\ &+ c_{2p+2} a_1 a_2 \cdots \\ &+ c_{2^{2p}} a_1 a_2 ... b_1 .. b_p \end{aligned}$$

Now, given the maps β_{ij}, we can set up 2^{2p} linear equations in c_r's by substituting $a_1, ..., a_p, b_1, ..., b_p$ on the right hand side and the value $\beta_{ij}(a_1, ..., a_p, b_1, ..., b_p)$ on the left hand side. Hence we have the linear system:

$$Ax = b$$

where $x = [c_1, c_2, \dots, c_{2^{2p}}] \in \mathbb{Z}^{2^{2p}}$, $b = [\beta_{ij}[00..00], \beta_{ij}[00..01], \dots, \beta_{ij}[11..11]] \in \{0,1\}^{2^{2p}}$, and $A \in \{0,1\}^{(2^{2p}) \times (2^{2p})}$ is the such that rows of A represents substitutions of $a_1, ..., a_p, b_1, ..., b_p$ with $00..00, 00..01, \dots, 11..11$ respectively on the right hand side of (1).

There is some ingenuity required in computing A efficiently (i.e. in constant time, instead of actually performing the substitutions) as shown in the code.

We can now use `np.linalg.solve` to find c_r's, for each of β_{ij} for $i = \{1..s\}, j = \{0..s\}$ and store them as vectors $\boldsymbol{c}_{ij} \in \mathbb{Z}^{2^{2p}}$.

Intuitively, v_r's are "projection vectors" which select the appropriate terms from $(a_1, ..., a_p, b_1, ..., b_p)$ for the corresponding coefficient c_r. We refer the reader to the code for the detailed algorithm.

5.2 4-Layer construction

(This section references objects and figures from the original paper, to preserve brevity of the report). [See FIG. 1. The universal network] The main challenge in implementing the 4-Layer construction proposed by the authors is to arrange computations of the form:

$$x_i^+ = \sum_{j=0}^{s} \beta_{ij}(a_1, ..., a_p, b_1, ..., b_p)x_j$$
$$= \sum_{j=0}^{s} \sum_{r=1}^{2^{2p}} c_r \sigma(v_r \cdot \mu)$$

as matrix-vector computations.

To demonstrate the necesarry transformations, we show how to compute the next state $\boldsymbol{x}^+$ and note that next stack contents $\boldsymbol{q}^+$ can be computed analogously.

F_4-layer neurons contain x_i's. Then, x_i^+ is computed in two steps as follows:
F_3-layer neurons compute each of the terms in the inner summation i.e. $\sigma(v_r \cdot \mu)$. F_2-layer neurons then compute the linear combination with c_r and the outer summation, and hence the new states x_i^+.

Let's look at the map from F_4 to F_3, i.e. the map $\mathcal{F}_4$, restricting our attention to $x_0, \ldots, x_s, a_1, ..., a_p, b_1, ..., b_p$ in layer F_4. Layer F_3 has for each x_j, 2^{2p} neurons. Hence, the shape of this map is $\mathcal{F} : \mathbb{Q}^{((s+1)2^{2p}) \times ((s+1)+2p+1)}$.

The $(s+1) + 2p + 1$ columns of this map correspond to $x_0, \ldots, x_s, a_1, ..., a_p, b_1, ..., b_p$. The rows of this map are divided into $s + 1$ groups of 2^{2p} rows in each group. The first group of rows selects x_0, the second group selects x_1 and so on and the $(s+1)$th group selecting x_s in the dot product with μ. In each group, the last $2p + 1$ elements are that of v_r, where the $+1$ th element is the $-k$ from the equation:

$$l_1 l_2 ... l_k = \sigma(l_1 + l_2 + \cdots + l_k - k + 1)$$

$\mathcal{F}_4$ also applies saturation σ to the computed values.

Now, we look at the map $\mathcal{F}_3 : \mathbb{Q}^{(s+1)2^{2p}} \to \mathbb{Q}^s$ which computes the new states x_i^+ by performing linear combination with c_r's. This map has the shape $\mathbb{Q}^{s \times (s+1)2^{2p}}$. Each row i is simply the coeffecient vectors $\boldsymbol{c}_i$ stacked horizontally (for $j = 1...(s+1)$).

We omit the computation of next stack contents $\boldsymbol{q}^+$ from the implementation, but observe that the matrices corresponding to maps $\mathcal{F}_4, \ldots, \mathcal{F}_1$ can be appended with more rows and columns to include other "neurons" of the universal network.

6 Conclusion

Given a description of a Turing machine, we have implemented an algorithm to construct a RNN which simulates it in linear time overhead. This shows that a RNN with only ratioinal weights, are capable of *accurately* modeling any decidable function. We have not explored several other interesting directions. It would be interesting to see, for example, how effective learning algorithms e.g. backpropogation are to learn these parameters, given a subset of input/output pairs of the partial recursive function, and to analyse the identifiability of the parameters learned by the network.

[The code is submitted as a `tar` archive and will be published to Github post evaluation]

References

John E. Hopcroft and Jeff D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company.

H.T. Siegelmann and E.D. Sontag. 1995. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150.