Universidade de Lisboa

Faculdade de Ciências

Departamento de Informática

Construção Modular de Redes Neuronais Recorrentes Analógicas

João Pedro Guerreiro Neto

Orientado pelos Professores Doutores
José Félix Costa e Helder Coelho

Lisboa, 2001

Sumário

Este trabalho encontra-se entre as áreas científicas da Teoria da Computação e das redes neuronais artificiais. Ele procura possíveis pontos de contacto e explora as melhores formas de integrar conceitos destas duas áreas para aproximá-las numa estrutura comum e mais geral de computação.

Este esforço será centrado, primeiro, na definição de uma arquitectura computacional baseada num determinado modelo neuronal (e consequente demonstração que o seu poder computacional é equivalente ao das máquinas de Turing), e em seguida, no desenvolvimento de um conjunto de ferramentas que permitam aproveitar as potencialidades desse mesmo modelo neuronal. Isso será obtido através da construção de uma linguagem de alto nível e de um processo de compilação capaz de traduzir a representação sintáctica numa rede neuronal modular e paralela.

Estas ferramentas centram-se em dois aspectos da computação, o controle e a aprendizagem. Neste contexto, controle diz respeito à capacidade de executar todo o algoritmo que utiliza informação simbólica, ou seja, informação com um significado directo e bem definido. Já a aprendizagem consiste no conjunto de algoritmos de manipulação sub-simbólica de informação, onde cada entidade básica de informação não possui um significado individual, mas faz parte de uma representação distribuída.

Será apresentada uma arquitectura de máquina capaz de executar o modelo neuronal apresentado abordada, e ainda uma estrutura neuronal para um agente computacional num ambiente distribuído.

Abstract

This work lies within the scientific areas of Theory of Computation and artificial neural networks. It researches some possible knowledge bridges between both areas, and tries to integrate concepts in order to achieve a common and broader computational framework.

This effort concentrates, firstly, on a computational architecture definition, based on a certain neural model (e subsequent demonstration that its power is equivalent to Turing Machines). Secondly, it will be developed a set of working tools to maximize and take advantage of this computational model. This will be achieved by using a high level programming language, and an automatic compilation process, able to translate the algorithmic representation of a certain problem into a parallel and modular neural network.

These tools focus on two computational concepts: control and learning. In this context, control means all algorithms that use symbolic information, i.e., information with a well-defined context and meaning. Learning, on the other side, consists in a set of sub-symbolic algorithms, where there is no individual meaning for each basic piece of information, and all knowledge is distributed.

It will also be presented a proposal for a specific hardware to execute the neural model neural, and a structure of a multiagent entity based on the previous concepts.

Agradecimentos

Enquanto trabalho essencialmente individual, uma Tese apoia-se sempre numa comunidade que envolve quem se propõe a concluí-la. Esta comunidade, constituída por elementos humanos, por inspirações obtidas na literatura, em conselhos pessoais e na experiência que se acumula ao longo do tempo, revela-se sempre decisiva na estrutura e no valor da obra final. Gostaria por isso de agradecer:

- Ao Professor Doutor José Félix Costa, como orientador competente e pelos muitos e excelentes trabalhos realizados conjuntamente que tanto contribuíram para a estrutura final desta Tese;
- Ao Professor Doutor Helder Coelho, como co-orientador perspicaz e dedicado, sem o qual todo o trabalho desenvolvido nesta Tese não teria sido possível;
- À Professora Doutora Hava Siegelmann, por todos os trabalhos efectuados em conjunto;
- Ao Professor Doutor Ademar Ferreira pelo constante apoio, colaboração e amizade;
- Ao Professor Doutor Jaime Sichman da Universidade de São Paulo, pelo apoio logístico e cientifico;
- Aos alunos Miguel Rosa e Paulo Carreira pela construção do compilador neuronal de funções parciais recursivas;
- Ao Pedro Rodrigues pelas leituras e importantes comentários referentes à Tese;
- Ao Jobi Hübner, pelo suporte dado no teste efectuado na plataforma de desenvolvimento multiagente SACI;
- À Martha Nader, minha mulher, por toda a paciência e amor que ela tão bem sabe dar;
- E a toda a minha família por todo o apoio e compreensão que sempre me deram.

Em relação às instituições:

- À Faculdade de Ciências da Universidade de Lisboa por todo o apoio financeiro e logístico essencial para a conclusão deste trabalho;
- À Universidade de São Paulo, nomeadamente à Politécnica Eléctrica pelo apoio logístico dado.

Índice

Índice das Figuras

Índice de Simbolos

$\wedge$	Conjunção lógica
$\vee$	Disjunção lógica
$\mathbb{N}$	Conjunto dos números naturais
$\mathbb{Z}$	Conjunto dos números inteiros
$\mathbb{Q}$	Conjunto dos números racionais
$\mathbb{R}$	Conjunto dos números reais
$\varnothing$	Conjunto com zero elementos, conjunto vazio
$\|A\|$	Cardinalidade de A, i.e., o número de elementos que pertencem ao conjunto A
$x \in A$	O elemento *x* pertence ao conjunto A
$A \subseteq B$	O conjunto A está incluído no conjunto B, i.e., $\underset{x \in A}{\forall} x \in B$
$A = B$	O conjunto A é igual ao conjunto B, i.e., $A \subseteq B \wedge B \subseteq A$
$A \cup B$	União dos conjuntos A e B, i.e., $\{ x : x \in A \vee x \in B \}$
$A \cap B$	Intersecção dos conjuntos A e B, i.e., $\{ x : x \in A \wedge x \in B \}$
$A - B$	Diferença dos conjuntos A e B, i.e., $\{ x : x \in A \wedge x \notin B \}$
$A \backslash \{a\}$	Conjunto A excepto elemento *a*, i.e., $\{ x : x \in A \wedge x \neq a \}$
$A \times B$	Produto cartesiano de A por B, i.e., $\{ (x,y) : x \in A \wedge y \in B \}$
A^n	Potência de um conjunto, i.e., $A \times A^{n-1}$, sendo que $A^0 = \varnothing$
2^A	Conjunto potência de A, i.e., $\{ B : B \subseteq A \}$
$f : A \rightarrow B$	Função *f* de domínio A e contradomínio B
$< t_1, \ldots t_n >$	n-tuplo pertencente a $A_1 \times A_2 \times \ldots \times A_n$, $t_i \in A_i$
R^{-1}	Inverso de R, i.e., $\{ <b, a> : <a, b> \in R \}$
$S \circ R$	Composição de S em R, i.e., $\{ <a, c> : <a, b> \in R \wedge <b, c> \in S \}$
$(A_i)_{i \in I}$	Sequência de valores $A_{i1}, A_{i2}, \ldots, A_{in}$, com $i_k \in I$
$f = \Theta(g)$	A função *f* tem a mesma ordem de grandeza da função *g*, i.e., $\underset{n_0, c_1, c_2 > 0}{\exists} x \geq n_0 \Rightarrow c_1.g(x) \leq f(x) \leq c_2.g(x)$
$f = \Omega(g)$	A função *f* é limitada inferiormente pela função *g*, i.e., $\underset{n_0, c > 0}{\exists} x \geq n_0 \Rightarrow c.g(x) \leq f(x)$
$f = O(g)$	A função *f* é limitada superiormente pela função *g*, i.e., $\underset{n_0, c > 0}{\exists} x \geq n_0 \Rightarrow f(x) \leq c.g(x)$

1. Introdução

A forma de computação mais comum encontrada nas aplicações reais e mesmo no interesse da investigação científica na área da Informática é a denominada computação simbólica, ou seja, computação sobre representação. Nesta visão, pode afirmar-se que as primitivas computacionais são igualmente primitivas representacionais [Chalmers 97]. As entidades básicas entre mudanças de estado da máquina simbólica, são elas também portadoras de significado sendo, portanto, símbolos.

Este tipo de computação é muito popular e tem fornecido excelentes resultados nas diversas áreas da Informática e da Engenharia, mas de modo algum esgota todos os recursos da Ciência da Computação. Qualquer sistema construído tem a necessidade de possuir um valor semântico associado, mas nada se diz se esse valor tem de estar obrigatoriamente ligado às primitivas computacionais. Nos sistemas conexionistas, por exemplo, o significado é distribuído pelo sistema em padrões de actividade de várias unidades de processamento, que podem por si mesmas não ter um valor simbólico agregado. Para usar o termo de Smolensky (cf. [Smolensky 88]), estes sistemas executam computação sub-simbólica. O nível de computação é anterior ao nível de representação.

Mas também estes sistemas são computacionais[1]. O termo sub-simbólico reflecte igualmente a possibilidade de ser usado para sustentar um sistema simbólico de mais alto nível (ou seja, a utilização do primeiro não só não exclui o segundo, como pode, em certos casos, facilitar o seu uso posterior).

Allen Newell, em [Newell 90], desenvolve a definição de arquitectura simbólica. Para isso, ele define um *sistema simbólico*, como um sistema computacional (eventualmente universal, ver [Hopcroft 79] e [Hein 96] para mais informações) com *memória*, capaz de guardar a informação necessária, um conjunto de *símbolos* para permitir o reconhecimento da informação disponível, *operações* para manipulação simbólica, *interpretação* para a especificação das operações e das suas combinações admissíveis, *composicionalidade* para que seja possível compor quaisquer estruturas simbólicas válidas. Finalmente, Newell, define uma arquitectura simbólica como uma estrutura fixa que realiza um sistema simbólico. Sendo fixo, implica que o comportamento de algoritmos sobre a arquitectura depende dos detalhes dos símbolos, operações e interpretações do sistema simbólico, e não da forma que este (e as suas componentes) estão implementadas.

Uma arquitectura sub-simbólica é uma estrutura que não utiliza símbolos no seu processamento interno. Uma forma típica nos sistemas sub-simbólicos é o uso de representações analógicas para representação de conhecimento. Dois exemplos de arquitecturas sub-simbólicas, são as redes neuronais (uma recente obra de introdução ao tema encontra-se em [Haykin 99]) e as arquitecturas de subsumpção de Brooks (cf. [Brooks 91]).

Estas arquitecturas, com as suas vantagens e desvantagens intrínsecas, formam o núcleo de qualquer sistema de controle computacional, seja ele centralizado ou distribuído. A integração dessas arquitecturas resulta nos chamados sistemas inteligentes híbridos (cf. [Kandell 92] e [Wilson 93]).

---

[1] De notar, que a distinção entre computação simbólica e subsimbólica não coincide com a distinção entre os diversos paradigmas computacionais, como máquinas de Turing ou redes

1.1. Motivação

Quando em 1943, McCulloch e Pitts apresentaram o seu artigo seminal sobre redes neuronais, foi definido um novo paradigma computacional, usando-o nesse artigo para a representação da lógica proposicional (cf. [McCulloch e Pitts 43]). No entanto, a comunidade cientifica desde então, tem visto as redes neuronais quase exclusivamente dirigidas para tarefas de optimização e aprendizagem, desde a Lei de Hebb (cf. [Hebb 49]) e o Perceptrão (cf. [Rosenblatt 58]), até às modernas variantes da Retropropagação (cf. [Rumelhart *et al.* 86a]). A vantagem, em relação aos métodos mais clássicos, reside na sua capacidade de representação sub-simbólica onde o conhecimento adquirido é disperso pela estrutura neuronal de forma homogénea.

Por outro lado, a computação simbólica foi tratada por outros tipos de paradigmas de computação, com especial foco nas máquinas seriais com arquitectura de von Neumann (um dos primeiros textos onde se refere a técnica de guardar programas de igual modo como se guardam dados, pode ser lido em [von Neumann 45/82]), cuja evolução produziu os computadores actuais. Nestas máquinas, a informação é guardada em zonas de memória específicas e processada por um sistema fixo de processamento definido por um algoritmo previamente codificado. Para facilitar a comunicação com o programador, foram criadas linguagens de alto nível para o desenvolvimento de algoritmos complexos. Estes algoritmos são traduzidos por compiladores na representação binária da máquina em questão.

Temos assim, novamente, os dois aspectos distintos de computação: o simbólico e o sub-simbólico. Porém, apenas as máquinas de von Neumann possuem um *hardware* barato, generalizado e com eficientes ferramentas de trabalho. Isto implicou que o paradigma neuronal da computação foi principalmente estudado e simulado nestas mesmas máquinas e não em

---

neuronais. Qualquer um destes sistemas é capaz de realizar ambos os tipos de computação, como veremos nesta Tese.

hardware específico. Ou seja, quando se refere um algoritmo neuronal, o que é apresentado é uma descrição algorítmica para computadores baseados na arquitectura de von Neumann, que executam a sua simulação.

Para melhorar a tarefa dos programadores de redes neuronais, foram criadas múltiplas ferramentas dirigidas para este assunto. Estas ferramentas abordam um ou mais dos seguintes três aspectos:

- Uso de Bibliotecas – No que diz respeito à criação de bibliotecas, a intenção central é que a partir de uma determinada linguagem de programação (Pascal, C, C++, Java, ...), é codificado um conjunto de variáveis, métodos e procedimentos, que criam um nível de abstracção complementar capaz de aumentar a produtividade e diminuir a percentagem de erros de programação. Existem múltiplas bibliotecas disponíveis com vários objectivos (cf. [Eberhart e Dobbins 90] para uma resenha de aplicações em computadores pessoais).
- Interpretação/Compilação – É apresentada uma linguagem específica que permite ao utilizador a definição do problema. Esta linguagem pode ser directamente executada pelo programa (linguagem interpretada), ou traduzida numa linguagem de mais baixo nível para ser executada por uma aplicação independente (compilação). Dois exemplos encontram-se nos trabalhos de Dongming Wang (cf. [Wang 90, 91, 92]) e no produto comercial *Synapse-3* da SIEMENS (cf. [SIEMENS 98]).
- Simulação – Os programas que realizam simulações baseiam-se num determinado modelo de rede, fornecendo ao utilizador procedimentos especializados para a construção arbitrária de redes.

Existem muitos programas disponíveis na comunidade científica (cf. [Schutter 92]). Todos estes programas fazem a tradução de uma linguagem próxima do paradigma neuronal de computação para a linguagem reconhecida pelos computadores de von Neumann. Suponhamos agora, que existe um computador neuronal. Um *hardware* capaz de simular eficientemente uma rede neuronal. Como traduzir os algoritmos existentes que foram feitos para máquinas seriais e não para este novo tipo de *hardware*?

A motivação central deste trabalho está relacionada com esta questão. Será possível automatizar a construção de redes neuronais arbitrariamente complexas num *hardware* capaz de executar o funcionamento paralelo de uma rede? Para responder a esta pergunta, outras questões se levantam. Que tipo de funções pode um determinado modelo de rede neuronal calcular? Neste momento, não falamos só de aproximação, mas sim de computação exacta no domínio numérico escolhido. E sendo possível, há perda ou ganho de eficiência em relação às máquinas de von Neumann? Que tipos de processos se tornam mais ou menos complexos com a mudança de paradigma da computação? O nosso esforço de investigação vai ao encontro deste conjunto de interrogações. Outra questão pertinente tem a ver com a crescente complexidade das aplicações actuais. Quando é necessário resolver uma tarefa complexa, é procedimento padrão separá-la em subtarefas relativamente independentes para facilitar a sua resolução e comprovar a sua correcção.

Tanto a experiência científica, como as aplicações comerciais provaram que o ramo da neurocomputação tem muita vitalidade, possuindo um amplo campo de aplicação e desenvolvimento futuro. No entanto, a tendência tem sido, e ainda é, olhar para uma rede neuronal como uma caixa preta de onde (depois de um dado processo de aprendizagem) se recebem resultados uma vez inseridos os dados iniciais. Esta visão terá dificuldades de se conciliar com o método lógico de desenvolvimento em subtarefas, na medida em que as exigências à investigação no campo forem crescendo em complexidade.

Não há nada que impeça que a função de uma rede não possa ser dividida em subfunções independentes, realizadas por redes mais simples e relacionadas entre si. Esta metodologia tem várias vantagens (dividindo o processo em partes, torna-se mais fácil detectar falhas e facilita o controle da complexidade associada) e mesmo algumas motivações biológicas (a neurociência indica que o cérebro humano é constituído por uma estrutura entrelaçada de módulos coadjuvantes) e psicológicas (acredita-se que o processo de aprendizagem humano é incremental, e que o sucesso num determinado estado da personalidade depende do sucesso do estado anterior).

O estudo da modularidade em redes neuronais e os seus potenciais mecanismos de comunicação e sincronização, será igualmente uma das motivações da Tese.

1.2. Objectivo

Com este trabalho, pretende mostrar-se que é possível conjugar os dois aspectos da computação, o controle e a aprendizagem, numa arquitectura neuronal maciçamente paralela. A construção desta arquitectura terá especial foco nas características modulares, para apresentar um método de limitação e controle da complexidade esperada em grandes redes neuronais. Estas questões serão discutidas e analisadas nos seguintes passos:

1) Uma demonstração formal de que o modelo de computação neuronal proposto é computacionalmente completo, i.e., equivalente à máquina de Turing. Características da rede: precisão ilimitada (interesse como modelo teórico), o número de neurónios e os pesos sinápticos são fixos. Com este resultado, é dada a garantia teórica do poder expressivo do sistema de computação neuronal a ser proposto.
2) Um processo de compilação que traduza qualquer estrutura de controle numa rede neuronal equivalente. Características: precisão limitada, o número de neurónios e os pesos sinápticos são fixos. A primeira diferença fundamental entre um compilador de C ou Pascal e este compilador, é que os primeiros traduzem o algoritmo para uma linguagem máquina reconhecida pela máquina serial onde esse algoritmo é executado, enquanto o segundo traduz o algoritmo numa descrição de uma rede neuronal independente do *hardware* utilizado[2]. A segunda diferença é que a linguagem, sendo dirigida para a construção directa de redes neuronais, pode introduzir construtores específicos para o tratamento de tarefas de aprendizagem.
3) A introdução de mecanismos de alteração da arquitectura da rede pela própria rede. Isto permitirá a representação de algoritmos de aprendizagem

[2] Tendo um hardware neuronal disponível, basta fazer a tradução da descrição da rede produzida pelo processo de compilação para a descrição apropriada desse hardware.

nesse mesmo modelo, tendo em conta os aspectos importantes para a sua integração com o processo de compilação. Características: precisão limitada, o número de neurónios é fixo, os pesos sinápticos não. Serão estudadas as possibilidades computacionais que possam aproveitar esta capacidade de modificação da topologia da rede por si mesma.

As respostas para as questões referidas abrem caminho para os seguintes temas:

Computação – o conceito de computação apresentado neste trabalho é diferente daquele usualmente reconhecido pelas máquinas de Turing. Aqui não se trata de um autómato de estados que utiliza uma fita de memória para ler e guardar informação. Este conceito de computação é melhor descrito pela evolução temporal de um vector de activações que descreve a actividade de uma rede neuronal.

Universalidade – provando que o modelo neuronal é computacionalmente completo, tem-se a garantia teórica da capacidade destas redes para computarem qualquer algoritmo.

Controlo – Antes desta Tese, o aspecto do controlo de execução de programas a partir de uma rede neuronal estava longe de ser simples. Os trabalhos existentes consideravam estruturas neuronais muito mais complexas que as apresentadas ao longo deste trabalho. A simplificação conseguida pelo passo conceptual dado aqui, reduziu as várias redes necessárias para um dimensão tão reduzida, que a sua análise torna-se quase trivial.

Sincronização – ao construir sistemas neuronais complexos, será criado um sistema de sincronização extremamente simples, mas capaz de controlar eficazmente redes de dimensão arbitrária.

Modularidade – juntamente com a definição de arquitecturas neuronais baseadas em descrições algorítmicas, como as funções parciais recursivas ou as linguagens de programação de alto nível, foi possível, como veremos ao longo da Tese, definir estruturas modulares de processamento. Esta modularização melhora o nível de controlo e correcção das redes neuronais produzidas.

Integração simbólica/sub-simbólica – os módulos de controle neuronal abrem a possibilidade de serem usados para controlar processos de aprendizagem executados por outros módulos pertencentes à mesma rede. Isto significa que é possível, em princípio, ter uma rede neuronal homogénea, capaz de executar computação simbólica ao mesmo tempo que executa computação sub-simbólica.

De notar que o modelo neuronal apresentado nesta Tese não é realista do ponto de vista biológico, não sendo essa uma preocupação do nosso trabalho. No entanto, é importante referir que actualmente não se conhecem simulações eficientes de sistemas simbólicos (como a máquina de Turing) em redes neuronais que utilizem funções de activação mais realistas (como a função sigmoidal). Por exemplo, em [Siegelmann 99] é apresentado uma simulação usando a sigmóide que possui atraso exponencial.

Outra restrição séria diz respeito à presença de ruído. No nosso trabalho, considera-se que não existe ruído de qualquer espécie. Porém, está provado que a existência de ruído limita seriamente o conjunto de linguagens reconhecidos pelas redes neuronais. Em [Maass e Sontag 99] é demonstrado que se o nível de ruído for Gaussiano, ou de qualquer outra distribuição que seja não nula num intervalo suficientemente grande, o poder computacional das redes neuronais recorrentes não consegue reconhecer sequer as linguagens regulares, ou seja, as linguagens reconhecidas pelos autómatos finitos.

Em relação às linguagens de programação de alto nível, a linguagem a ser apresentada nos capítulos 4 e 5 possui várias restrições, como o facto de não ter mecanismos de recursão ou a sintaxe da linguagem não ser muito elaborada, mas isso é compensado por um conjunto de ferramentas mais próximas do conceito de computação neuronal destes sistemas.

1.3. Sumário

Esta Tese é composta por um conjunto de capítulos com a seguinte estrutura:

Depois desta introdução, o segundo capítulo apresenta alguns elementos de Teoria da Computação e de Neurocomputação essenciais para se entender as estruturas e raciocínios utilizados nos capítulos seguintes. No terceiro capítulo é apresentado um sistema baseado nas funções parciais recursivas, que servirá juntamente com um conjunto de definições e teoremas, para demonstrar o resultado da universalidade. Será dada especial atenção à questão da modularidade, ponto central nas construções neuronais posteriores.

No capítulo quatro, é mostrado como se constrói um sistema automático de tradução de um algoritmo numa rede neuronal. Será descrito todo um conjunto de redes que definem os vários procedimentos constituintes de uma descrição algorítmica que possibilitará o processo de compilação. Este sistema é apresentado a partir da descrição de uma linguagem de programação paralela de alto nível capaz de representar os algoritmos necessários, linguagem esta que foca algumas das particularidades da neurocomputação (como o paralelismo massivo). No capítulo seguinte será feita a integração da aprendizagem no sistema proposto anteriormente. Esta integração mostrará que é possível a execução simultânea de computação simbólica e sub-simbólica numa arquitectura neuronal homogénea. Será apresentado um novo modelo para a rede neuronal, e com ele, uma extensão à linguagem descrita no capítulo anterior capaz de efectivar essa integração.

O sexto capítulo apresenta uma proposta de *hardware* capaz de executar eficientemente, no que diz respeito a velocidade, a memória usada, e a aspectos de comunicação e sincronização da computação paralela, o modelo das redes neuronais proposto. O capítulo sete mostra um caso de estudo no domínio da Inteligência Artificial Distribuída e Sistemas de Multiagentes, onde será apresentado um algoritmo distribuído de transmissão de informação por uma rede arbitrariamente complexa de agentes roteadores. Uma vez obtido o algoritmo, o processo de compilação permitirá a construção de um ‘agente neuronal’ capaz de executar esse conjunto complexo de tarefas associadas. Finalmente, a Tese termina com as conclusões obtidas ao longo deste trabalho, direcções futuras e as considerações finais.

2. Elementos de Computação e de Neurocomputação

2.1. Teoria da Computação

Em 1900, no Congresso Internacional de Paris, o eminente matemático alemão David Hilbert apresentou o que ele considerava serem os 23 problemas mais relevantes na Matemática para o novo Século que surgia. O último problema dessa lista, perguntava se existiria um processo mecânico pelo qual a verdade ou falsidade de qualquer conjectura matemática pudesse ser decidida. Este problema ficou conhecido por *Entscheidungsproblem*.

entscheidung palavra alemã para decisão

Do ponto de vista matemático, isso implicava a existência de um sistema axiomático bem definido onde todas as consequências decorrentes da aplicação do sistema seriam teoremas válidos, e todas as verdades matemáticas seriam por ele demonstráveis. Hilbert pretendia um sistema completo e coerente que fosse capaz de induzir a Matemática!

Em 1931, o lógico austríaco Kurt Gödel criou uma formalização desse sistema. Porém, o presente estava envenenado, limitando definitivamente as esperanças de Hilbert sobre o assunto. Gödel concluía que existiam preposições indecidíveis, i.e., as quais não se poderia demonstrar a sua validade ou

falsidade, em qualquer sistema axiomático do tipo que Hilbert propusera. Ou seja, para um sistema que fosse capaz de representar a aritmética, o facto de ele ser coerente implicava que ele tinha obrigatoriamente de ser incompleto. Num segundo resultado, Gödel demonstrou também que a consistência de um sistema não se pode provar dentro do próprio sistema. Isto implicava que para além de ser incompleta, a Matemática tinha irremediavelmente de "confiar" na coerência dos seus próprios resultados.

No entanto, Gödel tinha deixado por resolver se seria ou não possível responder à seguinte questão: Dado um sistema axiomático e uma qualquer preposição, é possível saber *algoritmicamente* se a preposição é indecidível nesse sistema? Se esta resposta fosse positiva, poder-se-ia eliminar as preposições indecidíveis e trabalhar somente com as restantes dentro do contexto do sistema. Caso contrário, a matemática ficaria para sempre "infectada" de proposições cujo valor lógico não poderia ser demonstrado. Ainda no âmbito desta questão, estava em aberto o problema de se saber exactamente o que *algoritmo* significava. O que era um algoritmo? Interessava associar à ideia intuitiva,

> *Um algoritmo (ou procedimento efectivo), é um conjunto de regras bem definidas, que nos informa, de momento a momento, o que se deve fazer.*

uma precisa definição matemática[3]. O problema com esta explicação, é que está baseada na capacidade do executor de interpretar correctamente as regras. Um observador incapaz poderia não entender na globalidade a explicação, ou por outro lado, um observador demasiado capaz (ou totalmente fora do contexto) poderia induzir efeitos secundários não previstos por quem as determinou. Uma melhor forma para definir um algoritmo, para além de dar o conjunto das regras, seria dar também os detalhes do *mecanismo de interpretação* dessas mesmas regras. Assim, um algoritmo passaria a ser,

a) uma linguagem que introduz a sintaxe das regras,

[3] Apesar de não ser óbvio que fosse possível descobrir um conceito formal equivalente a este conceito intuitivo complexo.

b) um mecanismo de interpretação de regras nessa linguagem,

c) um conjunto de regras respeitando as restrições estabelecidas em a).

O próprio Hilbert havia introduzido o conceito de funções primitivamente recursivas e conjecturara que era equivalente ao conjunto das funções que poderiam exprimir-se de forma algorítmica. No entanto, em 1928, William Ackermann encontrou uma função total intuitiva que não era derivável por esse conjunto de funções, a conhecida função de Ackermann.

função de Ackermann

$\psi(0,y) = y+1$
$\psi(x+1,0) = \psi(x,1)$
$\psi(x+1,y+1) = \psi(x, \psi(x+1,y))$

Depois do trabalho de Gödel, surgiram outras formalizações para o conceito de algoritmo. Church (em 1933 e 36) introduziu o cálculo λ, e usou-o como nova formalização de algoritmo. Em 1936, Kleene derivou o conceito de funções parcialmente recursivas, mostrando a diferença relativamente às funções primitivamente recursivas e ao mesmo tempo, provando a equivalência com o cálculo λ. Uma função dir-se-ia computável se e só se fosse uma função parcialmente recursiva.

Durante um curso em Cambridge, Alan Turing ouviu falar do *entscheidungsproblem,* e ao pensar nele, considerou que era necessário modelar o processo pelo qual o matemático atinge a prova de uma dada conjectura. Assim, tomou a expressão 'processo mecânico' usada por Hilbert figurativamente, no seu sentido literal. Em 1936, apresentou o que é hoje conhecido por Máquina de Turing, um formalismo que não tendo um cariz essencialmente matemático, era uma máquina no sentido intuitivo do termo. Possuía um sistema de controle que determinava o comportamento da máquina, uma fita onde guardava informação e uma cabeça de leitura/escrita. Turing também provou a equivalência da sua máquina com o cálculo λ.

Mas mais importante que isso, Turing respondeu à questão deixada aberta por Gödel: Seria possível saber algoritmicamente se uma preposição arbitrária é indecidível num dado sistema? A resposta de Turing deu o golpe final no programa de Hilbert: Não!

Para responder a esta pergunta, Turing mostrou inicialmente que a intuição da época ao dizer que problemas progressivamente mais complexos necessitavam de máquinas cada vez mais complexas, estava errada. Turing construiu uma máquina, seja U, que dada uma codificação adequada da máquina que resolvia um dado problema, U era capaz de simular a execução dessa máquina e, portanto, de resolver o problema em questão. E isto para qualquer máquina! Por isso, Turing chamou à maquina U, a máquina universal.

Turing provou então que o *entscheidungsproblem* poderia ser traduzido num outro problema, denominado por problema da paragem:

The Halting Problem

> *Seja uma máquina de Turing M e um conjunto de dados iniciais I.*
> *Será que a execução de M sobre I termina?*

Ele demonstrou, por redução ao absurdo, que se existisse uma máquina capaz de garantir o problema da paragem de qualquer máquina (que seria uma espécie de máquina universal), a execução desta máquina, para certos casos, levava-nos a situações contraditórias. Isto implicava que não havia nenhuma máquina capaz de resolver o problema da paragem nem o *entscheidungsproblem*. O caso negava a questão deixada em aberto por Gödel.

A matemática revela-se incompletamente mecanizável, i.e., por mais complexo que seja o formalismo apresentado, haverá sempre preposições indecidíveis que nem sequer podem ser identificadas como tal nessa formalização. Entretanto, o número de formalismos apresentados na literatura científica aumentava. Todos se exprimiam uns nos outros, e quando isso não acontecia, era devido ao sistema proposto ter falhas que lhe reduziam a capacidade. As mais variadas formas de expressão da computabilidade, baseadas em diferentes abordagens, eram análogas. Kleene, em 1952, apelidou como Tese de Church, a conjectura vigente, que a máquina de Turing e os seus equivalentes, exprimiam o conceito intuitivo de algoritmo[4].

[4] Uma tese mais forte, conhecida por Tese de Church Física, apresentada posteriormente por David Deutsch, afirma que qualquer sistema físico pode exprimir, no melhor caso, a

Pode classificar-se os formalismos existentes nos seguintes grupos:

- Abordagens matemáticas: cálculo λ, funções parcialmente recursivas,
- Abordagens operacionais: máquina de Turing, URM, RAM,
- Autómatos celulares,
- Cadeias de Markov, sistema de Post,
- Redes Neuronais.

Nas abordagens matemáticas, dada uma linguagem e o seu significado, cada palavra dessa linguagem corresponde a uma função computável. Veremos o exemplo das funções parcialmente recursivas no capítulo 3. O foco nestas abordagens está assim, na definibilidade da função a que corresponde um dado algoritmo.

Já nas abordagens operacionais, o foco é na mecanização do algoritmo. É apresentada uma forma de construir programas, ou seja, uma forma de descrever algoritmos e o funcionamento da máquina que executa esses programas. A execução do programa pela máquina faz corresponder ao resultado obtido o valor da função associada. Para além da máquina de Turing, muitas outras máquinas foram sugeridas. Falaremos nesta secção da máquina URM (Universal Register Machine) concebida por Shepherdson e Sturgis em 1963 (cf. [Shepherdson e Sturgis 63]). Uma outra máquina, proposta por John von Neumann, a máquina RAM (Random Access Machine), é a verdadeira precursora da arquitectura dos computadores modernos.

Nos autómatos celulares, cuja teoria foi desenvolvida por Codd, em 1963 (cf. [Codd 68]) e por von Neumann, em 1966 (cf. [von Neumann 66]), a computação opera sobre um espaço de dados em que todas as células de memória são processadas simultaneamente e possuem um sistema de controle uniforme. O sistema de Post (cf. [Post 43]) e as cadeias de Markov (1954)

---

computabilidade da máquina de Turing. No entanto, esta tese foi rejeitada por diversos formalismos que mostram capacidades para além da máquina de Turing, se considerarmos o uso de números reais [Siegelmann 99]. Este facto, motivador de muito debate, passa pela constatação que as teorias vigentes na Física usam números reais nos seus modelos descritivos. Ou seja, estas máquinas super-Turing limitam-se a utilizar os meios fornecidos pelos próprios sistemas físicos referenciados na Tese de Church Física.

definem algoritmo como um conjunto de regras de produção/rescrita que operam sobre os dados, i.e., sobre palavras de uma determinada linguagem. Quanto à computação neuronal, assunto motivador da nossa Tese, será o tema central dos capítulos seguintes. Apresentaremos nas secções seguintes alguns formalismos baseados na abordagem operacional, que nos serão úteis posteriormente.

Máquina URM

URM - Universal Register Machine

Na máquina URM existe um conjunto infinito (mas enumerável, ou seja, com a mesma cardinalidade dos números naturais) de registos de memória. Cada registo de memória guarda um número natural.

memória

Definição 2.1.1: Seja a função $r:\mathbb{N}\setminus\{0\}\rightarrow\mathbb{N}$. A *memória* de uma máquina URM é denotada pela sequência $(r_i)_{i\in \mathbb{N}\setminus\{0\}}$. R_i denota o i-ésimo registo da máquina, r_i denota o conteúdo de R_i. ◁

O conteúdo dos registos pode ser modificado pela máquina em consequência de certas instruções que ela reconhece. Uma sequência finita de instruções constitui um programa. A execução de uma máquina URM é simplesmente a execução de um programa sobre uma determinada memória.

instruções URM

Definição 2.1.2: As *instruções* URM são de quatro tipos distintos:

- Instruções Zero: Para cada $n \in \mathbb{N}$, a instrução Z(n) aplica o valor zero no n-ésimo registo de memória, R_n.
- Instruções Sucessor: Para cada $n \in \mathbb{N}$, a instrução S(n) aplica o valor r_n+1 no n-ésimo registo de memória, R_n.
- Instruções Transferência: Para cada $n, m \in \mathbb{N}$, a instrução T(m, n) aplica o valor r_m+1 no n-ésimo registo de memória, R_n.
- Instruções Salto: Para cada $n, m \in \mathbb{N}$, e $q \in \mathbb{N}\setminus\{0\}$, a instrução J(m, n, q) altera a ordem de execução da máquina URM. Se $R_m = R_n$ então a próxima instrução a ser executada é a q-ésima instrução do programa. Caso contrário passa a execução para a instrução a seguir. Em qualquer dos casos, a memória não é alterada. ◁

Observação 2.1.3: As instruções transferência podem ser definidas à custa dos outros três tipos de instruções. Porém, utilizam-se como funções base dado facilitarem a construção de programas.

T(m, n) = Z(n), J(m,n,5), S(n), J(1,1,2)

Definição 2.1.4: Um *programa* URM P é uma sequência finita não vazia de instruções URM, denotadas por $I_1, I_2, \ldots, I_{|P|}$. ◁

programa URM

Observação 2.1.5: Por convenção, considera-se que os n argumentos iniciais para a execução de um programa URM, estão colocados nos n primeiros registos de memória, $R_1, \ldots, R_n$. Os restantes registos guardam o valor zero. Também por convenção, cada programa URM deve guardar o resultado final da sua execução no primeiro registo, R_1.

Definição 2.1.6: A função $\rho : <I_1,\ldots,I_n> \rightarrow \mathbb{N}$, devolve o maior registo referenciado no programa URM $<I_1, \ldots, I_n>$. ◁

ρ

Definição 2.1.7: Uma *máquina* URM é o par $<(r_i)_{i\in \mathbb{N}\setminus\{0\}}, P>$, onde:

máquina URM

- $(r_i)_{i\in \mathbb{N}\setminus\{0\}}$ é uma memória
- P é um programa URM ◁

Definição 2.1.8: Um *configuração* de uma máquina URM é o par $<(r_i)_{i\in 1..k}, q>$ onde R_k é o registo de maior índice diferente de zero, e $q \in \mathbb{N}\setminus\{0\}$, representa a próxima instrução do programa URM a ser executada. ◁

configuração

O funcionamento de uma máquina URM com um programa P, define-se da seguinte forma. Seja a configuração actual $<(r_i)_{i\in 1..k}, q>$:

- se $q > \rho(P)$, a máquina pára; caso contrário
- se $I_q = Z(n)$, a próxima configuração actual será $<(r_i)_{i\in 1..max(k,n)}, q+1>$, com $r_n = 0$; caso contrário
- se $I_q = S(n)$, a próxima configuração actual será $<(r_i)_{i\in 1..max(k,n)}, q+1>$, com $r_n = r_n+1$; caso contrário
- se $I_q = T(m, n)$, a próxima configuração actual será $<(r_i)_{i\in 1..max(k,n)}, q+1>$, com $r_n = r_m$; caso contrário
- se $I_q = J(m, n, q')$ e $r_m = r_n$, a próxima configuração actual será $<(r_i)_{i\in 1..k}, q'>$; caso contrário
- se $I_q = J(m, n, q')$ e $r_m \neq r_n$, a próxima configuração actual será $<(r_i)_{i\in 1..k}, q+1>$.

Se o programa necessita de n argumentos para funcionar, a configuração inicial da máquina será $<(r_i)_{i\in 1..n}, 1>$, com r_i a guardar o valor do i-ésimo argumento.

Definição 2.1.9: Seja uma máquina URM M. Uma *computação* de M sobre o tuplo $t \in \mathbb{N}^n$, é a sequência de configurações obtidas a partir da configuração inicial $<(r_i)_{i\in 1..n}, 1>$ onde $r_i = t_i$, i=1..n. ◁

computação

Se a computação for uma sequência infinita de configurações diz-se que a máquina *diverge* para essa configuração inicial.

Exemplo 2.1.10: Seja o seguinte programa URM,

- < J(2,3,5), S(1), S(3), J(1,1,1) >

Este programa emula a função soma binária, onde por convenção, os dois argumentos estão guardados nos registos R_1 e R_2. O que o programa faz é acumular em R_1 tantas unidades quantas aquelas que existem em R_2, usando R_3 para comparar quantos sucessores já foram realizados. Quando R_2=R_3, em R_1 estará o valor da soma. A máquina pára ao saltar para uma instrução não existente.

2.2. Neurocomputação

Pode indicar-se com relativa segurança, o trio mais influente do início da neurocomputação como sendo constituída pelos artigos seminais escritos por Warren McCulloch e Walter Pitts em 1943, o livro de Donald Hebb em 1949 e o artigo de Frank Rosenblatt em 1958 (cf. respectivamente [McCulloch e Pitts 43], [Hebb 49] e [Rosenblatt 58]). No conjunto destes artigos foi mostrado que um paradigma computacional baseado e inspirado na actividade do cérebro humano tinha poder computacional não trivial e que poderia ser estudado matematicamente (McCulloch e Pitts demonstraram que até redes neuronais simples eram capazes de computar funções lógicas e aritméticas). Mostrou-se também que o condicionamento clássico observado nos animais era uma propriedade de neurónios individuais (Hebb apresentou uma lei de aprendizagem específica para as ligações sinápticas entre neurónios). Finalmente, com a apresentação do perceptrão e do neurocomputador MARK I

nos fins da década de 50, Rosenblatt mostrou que a aprendizagem automática e o reconhecimento de padrões eram exequíveis neste contexto computacional.

Na década de 60, a investigação expandiu-se tanto na análise matemática como na simulação computacional de redes neuronais artificiais. Block, colega de Rosenblatt, demonstrou em [Block 62] que o perceptrão constituído por um único neurónio era capaz classificar problemas de um determinado contexto, se essa classificação fosse matematicamente possível. Antes, em 1960, Bernard Widrow e Marcian Hoff, apresentaram um novo algoritmo adaptativo (cf. [Widrow e Hoff 60]) capaz de aprender mais depressa e acuradamente que o perceptrão. A regra apresentada era simples e elegante, e ela ou uma das suas variações, ainda são usadas em alguns sistemas actuais. Eles assumiram que o algoritmo era capaz de avaliar o erro entre a saída desejada e aquela que de facto era calculada pela rede num dado instante. Com esse dado, a rede adaptar-se-ia para minimizar o erro e aproximar-se da classificação óptima para o problema. Este algoritmo de correcção ficou conhecido por LMS, e foi aplicado pelos autores no conhecido modelo ADALINE.

LMS
Least Mean Square

Depois de um período de sucessos e crescentes expectativas, o campo começou a ter dificuldades de progressão sendo alvo de crescentes criticas e descréditos, culminando estas com o conhecido livro de Minsky e Papert em 1969, “Perceptrons” (cf. [Minsky e Papert 69]), onde os autores mostram matematicamente que o perceptrão não pode implementar funções tão triviais como a função lógica XOR, i.e., Ou Exclusivo. A tese implícita de Minsky e Papert era que todas as redes neuronais sofriam da mesma falha que o perceptrão. Esse argumento teve repercussão no meio científico, e a neurocomputação perdeu apoios e investigadores durante cerca de uma década.

$x \; XOR \; y \equiv (x \wedge \neg y) \vee (\neg x \wedge y)$

É costume pensar que a década de 70 foi uma travessia do deserto no que respeita às descobertas nesta área, mas, apesar de tudo, muito trabalho foi realizado e alguns dos conceitos mais importantes actualmente, surgiram nesse período. Teuvo Kohonen e James Anderson em 1972 (respectivamente [Kohonen 72] e [Anderson 72]) propuseram independentemente um modelo de memória associativa, onde o elemento básico neuronal era analógico e não

digital como nos modelos de McCulloch e de Rosenblatt. Tanto as entradas, os pesos sinápticos e as saídas possuíam valores contínuos. A relação entre a entrada e a saída era especificada por uma matriz de multiplicação que definia a estrutura do sistema. A regra de aprendizagem sináptica era uma generalização da lei de Hebb. Considerando as entradas e saídas como vectores de valores, foi Kohonen que chamou aos sistemas onde a entrada possui a mesma dimensão que a saída de autoassociativos, e heretoassociativos aos de dimensão distinta.

Stephen Grossberg investigou os mecanismos de inibição e saturação neuronais na neurocomputação (um neurónio biológico possuía comportamento fortemente não-linear em estados de saturação) e introduziu a conhecida função de activação sigmoidal, muito utilizada em aplicações recentes (cf. [Grossberg 76]).

$$f(u) = \frac{1}{1-e^{-u}}$$

Considera-se geralmente que a neurocomputação moderna se iniciou com os trabalhos do eminente físico John Hopfield (cf. [Hopfield 82, 84]). Para além do modelo conhecido por rede de Hopfield, uma das suas principais contribuições foi a introdução do conceito equivalente à energia física a partir da matriz que define a rede. A dinâmica da rede de Hopfield é construída (assincronamente e usando uma regra simples de adaptação sináptica) para minimizar a sua energia e só estabilizar quando chegar num mínimo local ou global. A passo que deu a partida definitiva para uma nova explosão criativa no campo, foi o trabalho iniciado por David Rumelhart e colegas (cf. [Rumelhart *et al*. 86]), com a apresentação de um algoritmo de aprendizagem para redes de múltiplas camadas, uma generalização do algoritmo de Widrow-Hoff, o conhecido algoritmo de *Retropropagação*[5]. Deste modo, caia a restrição conjecturada por Minsky, que qualquer rede neuronal seria incapaz de aprender certas funções, dado que as redes neuronais de múltiplas camadas são capazes de aproximar qualquer função, seja ela linearmente separável ou não.

ing., Back Propagation

[5] A descoberta do algoritmo é, de facto, mais antiga. Ele já tinha sido descrito em Agosto de 1974 em Harvard no Ph.D. de Paul Werbos (cf [Werbos 74]) e desenvolvido pelo próprio nos anos seguintes.

Redes neuronais

Nesta secção serão apresentados os conceitos referentes à neurocomputação necessários para a compreensão dos capítulos seguintes. Procurar-se-á aproximar as várias concepções propostas pela literatura cientifica, muitas vezes em formas distintas, numa unidade possível que facilitará a exposição do trabalho desenvolvido (para maiores detalhes cf. [Rojas 95], [Barbosa 93], [Garzon 95], [Haykin 99]).

Informalmente, uma rede neuronal típica é um modelo matemático de um sistema com: a) unidades de processamento denominadas por *neurónios*, b) conexões entre as unidades de processamento denominadas por *sinapses*, c) um conjunto de canais de comunicação com o exterior ou *ambiente*. Sobre este sistema é então apresentado um algoritmo que determina a forma como a rede se modifica consoante os dados fornecidos pelo ambiente. Desta forma, existem dois conceitos centrais: a parte física da rede, i.e., a sua arquitectura, e o procedimento algorítmico que determina o seu funcionamento. Apresentamos agora, alguns conceitos formais que serão centrais para a formulação do conceito de rede neuronal.

canal de dados

Definição 2.2.1: Um *canal de dados*, é uma função de domínio $\mathbb{N}$ e contradomínio $\mathbb{Q}$. $\lhd$

ambiente

Definição 2.2.2: *Ambiente* é um conjunto finito de canais de dados. $\lhd$

Observação 2.2.3: Dada a sua inspiração biológica, onde as entradas, saídas e os estados dos neurónios biológicos são variáveis físicas e bioquímicas, é considerado que valores contínuos são mais apropriados na modelação neuronal. Por isso é escolhido o conjunto dos racionais para os modelos neuronais (o conjunto dos Reais não pode ser representado por máquinas de Turing ou equivalentes).

Definição 2.2.4: A unidade de processamento, doravante chamada de *neurónio*, é um tuplo $x_k = <S_k, T_k, s, \delta, \phi>$ onde:

neurónio

- $S_k = \{s_{k1}, s_{k2}, ..., s_{kN}\}$, é um conjunto finito de conexões, ou *sinapses*. Cada sinapse s_{ki} é um trio $<x_i, w_{ki}, x_k>$, onde x_i é o neurónio *pré-sináptico*, w_{ki} é o respectivo *peso sináptico*, sendo o próprio neurónio x_k também designado por neurónio *pós-sináptico*.

sinapse
peso sináptico
neurónios pré/pós-sinápticos

- $T_k = \{t_{k1}, t_{k2}, ..., t_{kM}\}$, é um conjunto finito de conexões de entrada. Cada t_{ki} é um trio $<u_i, w_{ku_i}, x_k>$, onde u_i é o i-ésimo canal de dados do ambiente U, e w_{ku_i} o seu respectivo peso.
- s é um valor em $\mathbb{Q}$, denominado *estado* ou *memória local*.
- $\delta : \mathbb{Q}^{N+M+1} \rightarrow \mathbb{Q}$, é uma função que determina o valor que o neurónio computada a partir dos valores de entrada e da sua memória local, s, denominada por *função de transferência*.
- $\phi : \mathbb{Q} \rightarrow \mathbb{Q}$, é uma função que determina o valor a ser enviado para os neurónios necessários e/ou para o ambiente, denominada por *função de activação*. ◁

Observação 2.2.5: Diz-se que uma sinapse s é *excitatória* se e só se o respectivo peso sináptico for positivo. Se o peso for negativo, a sinapse diz-se *inibitória*.

sinapse excitatória
sinapse inibitória

Observação 2.2.6: Em muitos modelos, como no perceptrão, não existe uma memória local. O valor de saída do neurónio é exclusivamente calculado a partir dos valores de entrada.

Observação 2.2.7: Existe, em certos modelos, um valor de entrada especial denominado por *pendor*. Nesta definição, o pendor é representado como um neurónio especial, designado por x_0, cujo valor de saída é dado pela função $x_0(t)=1$, $t \geq 0$. A ligação ao neurónio ocorre pela sinapse s_{k0} com peso sináptico igual ao valor do pendor.

ing., bias

Uma das funções de transferência mais comum é a aplicação linear das suas entradas com os pesos das respectivas ligações entre os neurónios, i.e.,

$$\delta_L(x_0, x_1, ..., x_N, u_1, ..., u_M, s) = \sum_{i=0}^{N} a_{ki}x_i(t) + \sum_{i=1}^{M} b_{ki}u_i(t) \qquad (1)$$

aplicação de transferência linear

para o k-ésimo neurónio, onde os racionais a_{ki} e b_{ki} representam os pesos das sinapses s_i e das conexões de entrada aos canais u_i, respectivamente.

Outras funções são possíveis. Uma função a utilizar no capítulo 5 será a aplicação de uma função polinomial aos valores de entrada, definindo assim, funções de ordem dois ou superior. Existem várias funções de activação na literatura. A função de activação proposta inicialmente em [McCulloch e Pitts 43], denominada por função *degrau*, é uma das mais simples, podendo ser descrita pela seguinte função:

ing., Heavyside ou Step

$$\phi_H(x) = \begin{cases} 1 & , x>0 \\ 0 & , \text{caso contrário} \end{cases} \qquad (2)$$

função degrau

Para tornar diferenciável a função de activação ϕ_H, foi posteriormente criada a função sigmóide:

$$\phi_S(x) = \frac{1}{1 + e^{-x}} \qquad (3)$$

função sigmóide

Outra função a ser utilizada nos próximos capítulos, é a função linear saturada, denominada doravante por σ:

$$\sigma(x) = \begin{cases} 1 & , x\geq 1 \\ x & , 0<x<1 \\ 0 & , x\leq 0 \end{cases} \qquad (4)$$

função linear saturada

A partir das definições de ambiente e de neurónio, é possível definir uma rede neuronal.

Definição 2.2.8: Seja X um conjunto de neurónios e U um ambiente. Uma *rede neuronal* R é um tuplo <X,U,Y>, onde $Y \subseteq X$ é denominado conjunto dos neurónios de saída. ◁

rede neuronal

Observação 2.2.9: Nesta definição, não é necessário que todos os neurónios da rede tenham as mesmas funções de transferência ou activação, e que pode existir neurónios que não tenham acesso ao ambiente nem pertençam a Y, sendo denominados por *neurónios escondidos*. Neurónios que não são neurónios de entrada para nenhuma sinapse, são designados *neurónios terminais*.

ing., hidden neurons

Definição 2.2.10: Seja X o conjunto de neurónios da rede neuronal R. A *topologia* de R é a relação $W_R \subseteq X \times \mathbb{Q} \times X$, definida pelo conjunto das sinapses em X. ◁

topologia

A topologia de uma rede neuronal define um grafo directo constituído pelos neurónios e respectivas sinapses da rede em questão.

Definição 2.2.11: Seja uma rede neuronal R = <X,U,Y>. Diz-se que R é *não-recorrente* se a topologia definida por X não tiver ciclos. Caso contrário, a rede diz-se *recorrente*. ◁

ing., feedforward net

ing., feedback net

Observação 2.2.12: Considerámos que os valores processados pela rede fossem representados pelo conjunto dos números racionais. Este tipo de rede, capaz do processamento de valores contínuos diz-se uma *rede analógica*, oposta à noção de uma rede capaz de processar apenas valores discretos (usualmente, valores binários) denominada por *rede digital* ou *discreta*.

rede analógica
rede discreta

Nos próximos dois capítulos serão utilizadas redes neuronais com o seguinte tipo de arquitectura, designada por rede-σ.

Definição 2.2.13: Uma *rede-σ* é uma rede neuronal recorrente R = <X,U,Y>, onde a computação realizada por cada neurónio (i.e., a composição da função de transferência e da função de activação neuronal) é dada pela seguinte expressão:

rede -σ

$$x_j(t+1) = \sigma\left(\sum_{i=1}^{N} a_{ji} x_i(t) + \sum_{k=1}^{M} b_{jk} u_k(t) + c_j \right) \qquad (5)$$

onde c_j é o pendor, i.e., $c_j = a_{j0} x_0(t)$, e a função σ é descrita pela função (4). ◁

Graficamente, os diversos elementos do neurónio descrito em (5) são representados da seguinte forma:

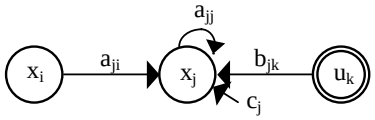


fig. 2.2.14 : representação gráfica de um neurónio numa rede-σ.

3. Universalidade das Redes-σ

As redes neuronais foram desenvolvidas principalmente como mecanismos de optimização e aproximação. Já o aspecto da computabilidade foi menos estudado pela comunidade científica até ao fim dos anos oitenta, a mesma década que viu ressurgir o interesse geral pelas redes neuronais.

Em 1987, Pollack demonstrou em [Pollack 87] que um determinado tipo de rede recorrente era universal. Esse modelo consistia num número finito de neurónios com dois tipos diferentes de funções de activação, a função identidade e uma função de limiar. Além disso, as equações que representavam as redes tinham activações multiplicadas entre si. Pollack deixou aberta a questão se seriam precisos sistemas do tipo que ele apresentava para obter a universalidade, conjecturando que sim.

Outros autores investigaram o poder computacional de arquitecturas com neurónios mais simples, mas com um número não limitado de unidades de processamento. Os trabalhos de Garzon e Franklin (cf. [Garzon e Franklin 89])

assumem um número ilimitado de neurónios para obter o mesmo poder computacional[6].

Já no artigo de Wolpert (cf. [Wolpert 91]), é estudado um modelo onde todos os neurónios possuem apenas funções de activação linear, de tal modo que o modelo simula a máquina de Turing (possuindo mesmo capacidades super-Turing). No entanto, o número de neurónios necessários é infinito, sendo mesmo não contável, dado que a construção necessita de grupos de neurónios para codificar as diferentes configurações de memória de uma máquina de Turing.

Os trabalhos de Hava Siegelmann e de Eduardo Sontag (cf. [Siegelmann 93] e [Siegelmann e Sontag 94, 95]) procuraram responder à mesma questão de qual seria o poder computacional de uma rede finita de neurónios. No entanto, uma diferença fundamental relativamente aos trabalhos anteriores foi a escolha da função de activação σ. Mais detalhes sobre computabilidade deste tipo de redes pode ser lido em [Gavaldà e Siegelmann 99] e [Siegelmann 99].

$$\sigma(x)=\begin{cases}0, & x<0\\ x, & 0\le x\le 1\\ 1, & x>1\end{cases}$$

Desenvolvendo o mesmo tipo de investigação, Pascal Koiran estudou os aspectos da computabilidade para famílias de funções de activação (cf. [Koiran 96]) e para espaços de duas dimensões (cf. [Koiran *et al.* 94]), com a preocupação de manter a característica universal em ambos os modelos.

Um outro trabalho realizado por Alessandro Sperduti em [Sperduti 97] apresenta e desenvolve a capacidade computacional de determinados tipos de arquitectura neuronal, dado ser conhecido que ao restringir a topologia da rede, se obtêm diferentes poderes computacionais e capacidades de aprendizagem. Outros estudos sobre o poder computacional e a complexidade de vários modelos neuronais podem ser encontrados em [Orponen 92], [Andonie 96], [Siegelmann e Giles 97] e [Garzon e Botelho 99].

---

[6] De certa forma uma ideia equivalente à própria ideia da máquina de Turing, que apenas precisa de um número finito de elementos de memória para resolver qualquer problema computável, apesar de não se poder saber à partida a dimensão dessa necessidade.

O desenvolvimento do trabalho de Siegelmann e Sontag demonstrou que o poder computacional de redes desse tipo era equivalente ao das máquinas de Turing. O método de prova proposto foi o de construir um processo de compilação que transforma qualquer máquina de Turing numa rede finita com pesos racionais (portanto computáveis). Desde modo, a compilação de uma máquina de Turing universal é imediata, ficando provada a equivalência entre os dois sistemas[7].

A ideia apresentada neste capítulo é semelhante. O nosso modelo também é baseado numa rede finita de neurónios com função de activação σ e com pesos sinápticos racionais. No entanto, é no método da prova que reside a diferença. Vamos utilizar o conceito de computabilidade da máquina URM e das funções parciais recursivas, construindo um processo de compilação que produz uma rede neuronal específica para cada função computável apresentada. Mas uma rede possuidora de características modulares e de sincronização que facilitam tanto a construção como a prova da sua correcção.

3.1. Funções Parciais Recursivas

As abordagens à computabilidade centraram-se no aspecto referente à mecanização do algoritmo, ou seja, uma determinada máquina apresenta na sua própria definição, como se deve executar um conjunto bem definido de tarefas para obter o resultado pretendido (cf. secção 2.1).

No entanto, nem todas as abordagens à computação se preocuparam com este aspecto. A teoria da computação vista pela teoria das funções parciais recursivas, centra-se na definibilidade das próprias funções computáveis. Assim, criou-se um sistema axiomático bem definido, onde fosse possível extrapolar como teoremas todas as funções computáveis.

[7] Dado que uma máquina de Turing é capaz de emular redes neuronais desse tipo, bastando executar um programa apropriado.

O algoritmo, a descrição dos passos a executar por uma dada máquina, é substituída aqui pela descrição da função associada, i.e., pela apresentação da demonstração obtida a partir do dado sistema axiomático. Apresentamos as definições sintácticas e semânticas do sistema axiomático que determina o conjunto das funções parciais recursivas.

Sintaxe

Nesta primeira definição, apresentaremos de forma recursiva, todas as palavras bem formadas que formam a linguagem que utilizaremos ao longo deste capítulo.

linguagem $\mathcal{R}$

Definição 3.1.1: Uma palavra em $\mathcal{R}$ é definida por uma das seguintes expressões:

- W com aridade 0
- S com aridade 1
- $U_{i,n}$ com aridade n, $n \geq 1$, $1 \leq i \leq n$
- Se $g \in \mathcal{R}$ com aridade n, e $f_1, ..., f_n \in \mathcal{R}$ todos com aridade k, então $C(f_1, ..., f_n, g) \in \mathcal{R}$ com aridade k
- Se $g \in \mathcal{R}$ com aridade n+2, e $f \in \mathcal{R}$ com aridade n, então $R(f, g) \in \mathcal{R}$ com aridade n+1
- Se $f \in \mathcal{R}$ com aridade n+1, então $M(f) \in \mathcal{R}$ com aridade n
- Uma palavra pertence a $\mathcal{R}$ somente se for obtida pela aplicação de um ou mais dos pontos anteriores. ◁

descrição

Observação 3.1.2: A linguagem $\mathcal{R}$ será designada por conjunto das descrições das *funções parciais recursivas*. Uma palavra de $\mathcal{R}$ designar-se-á por *descrição*. W será doravante denotada por *constante zero*, S por *sucessor*, $U_{i,n}$ por *i-ésima projecção n-ária*, C por *composição*, R por *recursão* ou *recorrência* e M por *minimização*.

Exemplo 3.1.3: O elemento $R(W,U_{1,2}) \in \mathcal{R}$ com aridade 1, dado que $W \in \mathcal{R}$ com aridade 0 e $U_{1,2} \in \mathcal{R}$ com aridade 2.

Exemplo 3.1.4: O elemento $R(W,S) \notin \mathcal{R}$, dado que $W \in \mathcal{R}$ com aridade 0 e $S \in \mathcal{R}$ somente com aridade 1.

Semântica

Vamos de seguida descrever o significado das descrições de $\mathcal{R}$, atribuindo a cada elemento, o seu comportamento dentro de cada expressão válida que os contém (para um resumo de semântica denotacional cf. [Gunter e Scott 90], [Moses 90]).

Definição 3.1.5: A cada definição $\ell \in \mathcal{R}$, corresponde uma função parcial $f:\mathbb{N}^n \to \mathbb{N}$, representada por $\mathcal{F}[\![\ell]\!]$, sendo:

$\mathcal{F}[\![W]\!] = 0$

$\mathcal{F}[\![S]\!](x) = x+1$

$\mathcal{F}[\![U_{i,n}]\!](\vec{x}) = x_i$

$\mathcal{F}[\![C(f_1, \ldots, f_k, g)]\!](\vec{x}) = \mathcal{F}[\![g]\!](\mathcal{F}[\![f_1]\!](\vec{x}), \ldots, \mathcal{F}[\![f_k]\!](\vec{x}))$

$\mathcal{F}[\![R(f, g)]\!](\vec{x}, 0) = \mathcal{F}[\![f]\!](\vec{x})$

$\mathcal{F}[\![R(f, g)]\!](\vec{x}, y+1) = \mathcal{F}[\![g]\!](\vec{x}, y, \mathcal{F}[\![R(f, g)]\!](\vec{x}, y))$

$\mathcal{F}[\![M(f)]\!](\vec{x}) =$ $\min\{ y \in \mathbb{N} : \mathcal{F}[\![f]\!](\vec{x}, y) = 0 \wedge \bigvee_{0 \le z < y} \mathcal{F}[\![f]\!](\vec{x}, z)$ está definido $\}$, se existir,

indefinido, caso contrário. ◁

Computabilidade de uma descrição

Para provar que a uma qualquer descrição $\ell \in \mathcal{R}$ corresponde efectivamente uma função computável, utilizamos a máquina URM com a sua noção de programa, para computar a função $\mathcal{F}[\![\ell]\!]$. Se essa função pode ser computada por um determinado programa URM, ela é computável. Fica assim demonstrado que as funções descritas em $\mathcal{R}$ estão contidas no conjunto das funções parciais recursivas.

Teorema 3.1.6: Para qualquer $\ell \in \mathcal{R}$ com aridade n, existe um programa URM P, tal que o resultado final da computação de P com argumentos iniciais $x_1,\ldots,x_n$, é igual ao valor de $\mathcal{F}[\![\ell]\!](x_1,\ldots,x_n)$.

Demonstração: A demonstração será realizada por indução na estrutura de uma descrição. Prova-se, que para cada elemento W, S, $U_{i,n}$, e composições C, R e M, existem programas URM que computam os seus valores.

Utilizar-se-á a noção de subprograma, onde $F[x_1, ..., x_n \rightarrow y]$ significa executar o programa F com os argumentos de entrada nos registos x_1, ..., x_n e guardar o resultado da computação no registo y.

A execução de um subprograma não interfere com a do programa principal, pois é possível transferir a computação para uma área de memória não utilizada e quando ela terminar, transferir o resultado para o registo pretendido (consultar [Cutland 80] para mais detalhes).

- W é computado pelo programa URM $P_W = < Z(1) >$
- S é computado pelo programa URM $P_S = < S(1) >$
- $U_{i,n}$ é computado pelo programa URM $P_{Ui} = < T(i,1) >$
- $C(f_1, ..., f_k, g)$ é computado pelo programa abstracto URM

  $P_C = <$ $T(1,m+1), \ldots, T(n,m+n),$
  $F_1[m+1,...,m+n \rightarrow m+n+1],$
  $\ldots,$
  $F_k[m+1,...,m+n \rightarrow m+n+k],$
  $G[m+n+1,...,m+n+k \rightarrow 1] >$

  onde F_1, …, F_k e G são os programas URM que computam as descrições f_1, ..., f_k e g, respectivamente; *n* o número de argumentos de F_i; e *m* o índice seguinte ao maior índice de registo utilizado pelos subprogramas F_1, …, F_k, G.

  O funcionamento de P_C é bastante simples. Computam-se os k valores de $f_i(x_1, ..., x_n)$ e guardam-se esses valores nos registos m+n+1 a m+n+k. Em seguida, esses valores são usados como argumentos para a computação do valor de g. As transferências iniciais servem para

salvaguardar os valores x_1, ..., x_n, durante as execuções dos vários subprogramas F_i.

- R(f, g) é computado pelo programa URM

P_R = < T(1,m+1), …, T(n,m+n), T(n+1,m+n+1),
F[1,...,n→m+n+3],
I_q : J(m+n+1,m+n+2,p),
G[m+1,...,m+n, m+n+2,m+n+3→m+n+3],
S(m+n+2),
J(1,1,q),
I_p: T(m+n+3,1) >

onde F e G são os programas URM que computam as descrições f e g, respectivamente; *n* o número de argumentos de R; e *m* o índice seguinte ao maior índice de registo utilizado pelos subprogramas F e G.

Para o programa F_R, executa-se inicialmente o subprograma F. Se o último argumento da função (guardado no registo m+n+1) for igual a zero, o programa termina com o resultado obtido pela execução de F. Caso contrário, o subprograma G é executado quantas vezes forem necessárias até que se atinja o valor do registo m+n+1.

- M(f) é computado pelo programa URM

P_M = < T(1,m+1), …, T(n,m+n), T(n+1,m+n+1),
I_q : F[m+1,...,m+n,m+n+1→1],
J(1,m+n+2,p),
S(m+n+1),
J(1,1,q),
I_p : T(m+n+1,1) >

onde F é o programa URM que computa a descrição f; *n* o número de argumentos de M; e *m* o índice seguinte ao maior índice de registo utilizado por F.

Para o programa F_M, executa-se o subprograma F enquanto o resultado da computação não for igual a zero. Para cada computação de F, o valor do seu último argumento é incrementado (registo n+m+1).

Provou-se que as funções descritas por W, S e $U_{i,n}$ são funções computáveis. Provou-se também que a composição, recursão e minimização de funções computáveis, resultam igualmente em funções computáveis (ou seja, existe sempre um programa URM que computa os valores da função descrita). Qualquer descrição $\ell \in \mathcal{R}$ é resultado de uma combinação finita de elementos W, S e $U_{i,n}$ e de composições C, R e M. Logo, a função $\mathcal{F}[\![\ell]\!]$ descrita por ℓ é computável. ◁

3.2. Universalidade da linguagem $\mathcal{R}$

Provámos na secção anterior que qualquer descrição de $\mathcal{R}$ descreve uma função computável. Será que se pode descrever em $\mathcal{R}$ todas as funções computáveis? Para responder a esta questão, é necessário provar que dada uma qualquer função computável $f:\mathbb{N}^n \rightarrow \mathbb{N}$, existe uma descrição $\ell \in \mathcal{R}$, tal que $\mathcal{F}[\![\ell]\!] = f$.

Sabemos dos resultados da Teoria da Computação que a máquina URM tem a propriedade da universalidade (ver [Lewis 81]). Se provarmos que a função computada por qualquer programa URM pode ser descrita em $\mathcal{R}$, garantimos a universalidade de $\mathcal{R}$.

Para isso, mostra-se como se pode construir uma descrição $\mathcal{H} \in \mathcal{R}$ que tem a seguinte propriedade: dada uma função computável $f:\mathbb{N}^n \rightarrow \mathbb{N}$ computada pelo programa URM P, codificado de forma adequada por $x \in \mathbb{N}$, e dado uma sequência de argumentos x_1, ..., x_n, codificados por $y \in \mathbb{N}$, então $\mathcal{F}[\![\mathcal{H}]\!](x,y) = f(x_1,...,x_n)$.

O conceito central para a construção da descrição $\mathcal{H}$ é emular o funcionamento da máquina URM ao executar o programa P cujo código é x, sobre a sequência de argumentos cujo código é y.

Codificação de um programa URM

Definição 3.2.1: Seja a seguinte codificação para a instrução[8] URM, I:

$$\alpha_I(I) = \begin{cases} 2.3^n & , I = Z(n) \\ 2^2.3^n & , I = S(n) \\ 2^3.3^n.5^m.7^q & , I = J(n,m,q) \end{cases} \quad (6)$$

◁

Definição 3.2.2: Seja a seguinte codificação para um programa URM, $P = < I_1, I_2, \ldots, I_k >$:

$$\alpha_P(P) = 2^k \ \ 3^{\alpha_I(I_1)} \ \ 5^{\alpha_I(I_2)} \ \ \ldots \ P_r(i+1)^{\alpha_I(I_i)} \ \ldots \quad (7)$$

sendo $P_r(i)$ o i-ésimo primo . ◁

Codificação dos Argumentos

Cada programa P possui n argumentos naturais necessários à computação do valor final da função computada por P. Esta sequência de valores é codificada da seguinte forma:

Definição 3.2.3: Seja a seguinte codificação para a sequência $x_1, \ldots, x_n$:

$$\alpha_{arg}(x_1, \ldots, x_n) = 3^{x_1} \ \ 5^{x_2} \ \ 7^{x_3} \ \ \ldots \ P_r(n+1)^{x_n} \quad (8)$$

◁

Para extrair informação do natural assim obtido, utiliza-se a seguinte função computável REG(y, i) que devolve o conteúdo do i-ésimo registo de memória codificado no natural y.

$$\text{REG}(\ y, i\ ) = (y)_{i+1}$$

sendo $(x)_y$ a função computável que devolve o expoente do y-ésimo primo da factorização prima de x (sendo que o primeiro primo é o número 2).

[8] A instrução T pode ser definida a partir das outras três instruções (*vide.* cap. 2).

Cada programa URM pode assim, ser transformado num número positivo através do processo de codificação apresentado.

Para extrair informação desse número, utilizam-se as seguintes funções computáveis: ARG1(x, i), ARG2(x, i), ARG3(x, i), que devolvem o 1º, 2º e 3º argumento da i-ésima instrução (se estiver definido) do programa F tal que $x=\alpha_P(F)$; e COD(x,i) que devolve um se a i-ésima instrução é uma instrução Z, 2 se for S, 3 se for J, e zero se não existir (i.e., i é maior que o número de instruções de F).

$$\text{ARG1}(x, i) = ((x)_{Pr(i+1)})_2$$

$$\text{ARG2}(x, i) = ((x)_{Pr(i+1)})_3$$

$$\text{ARG3}(x, i) = ((x)_{Pr(i+1)})_4$$

$$\text{COD}(x, i) = \begin{cases} ((x)_{Pr(i+1)})_1 & , i \leq (x)_1 \\ 0 & , i > (x)_1 \end{cases}$$

Funções auxiliares

Apresentamos ainda um conjunto de funções que vão servir para definir a descrição $\mathcal{H}$.

A primeira é a função INST que devolve qual o índice da próxima instrução a ser executada, dado o programa de código x, uma instrução actual I_i e uma sequência de registos de código y. Devolve zero, se a máquina parar no próximo instante. Esta função, bem como as seguintes, são funções de um argumento, o que implica que se tem de codificar x, i e y num só natural. Seja $a = 2^x 3^{2^i y}$.

$$\text{INST}(a) = \begin{cases} i+1 & , A \\ \text{ARG3}(x, i) & , B \\ 0 & , C \end{cases}$$

$$A \equiv ((\text{COD}(x, i)=1) \vee (\text{COD}(x, i)=2) \vee (\text{COD}(x, i)=3 \wedge \text{REG}(y, \text{ARG1}(x, i)) \neq \text{REG}(y, \text{ARG2}(x, i))) \wedge ((i+1) \leq (x)_1)$$

$B \equiv ((\text{COD}(x, i)=3 \wedge \text{REG}(y, \text{ARG}1(x, i)) = \text{REG}(y, \text{ARG}2(x, i)))$
$\quad \wedge \ (\text{ARG}3(x, i) \leq (x)_1)$

$C \equiv \ \neg A \wedge \neg B$

A função seguinte, MEM, devolve o próximo estado da memória depois de se executar a instrução I_i do programa de código x, e cujo estado actual é dado por y. Seja $a = 2^x 3^{2^i y}$.

$$\text{MEM}(a) = \begin{cases} y / P_r(n+1)^{\text{REG}(y,n)} & , \text{COD}(x, i) = 1 \\ y * P_r(n+1) & , \text{COD}(x, i) = 2 \\ y & , \text{COD}(x, i) \in \{0,3\} \end{cases}$$

com $n = \text{ARG}3(x, i)$.

Finalmente, a partir das funções INST e MEM, pode construir-se a função UPDATE que dado um determinado estado da máquina (x, i, y) devolve o próximo estado (x, INST(a), MEM(a)) com $a = 2^x 3^{2^i y}$.

$$\text{UPDATE}(a) = 2^x \ 3^{2^{\text{INST}(a)}_{\text{MEM}(a)}}$$

Como INST, MEM e UPDATE são funções obtidas a partir de sucessivas composições de funções computáveis, são elas próprias funções computáveis.

A função EXEC(a, n) é uma função que devolve o estado da máquina ao fim de n+1 iterações, a partir de um estado inicial (x, 1, y), i.e., $a = 2^x 3^{2^1 y}$. Esta função obtém-se por recorrência no segundo argumento,

$$\begin{aligned} \text{EXEC}(a, 0) &= \text{UPDATE}(a) \\ \text{EXEC}(a, n+1) &= \text{UPDATE}(\text{EXEC}(a, n)) \end{aligned}$$

Construção da descrição universal $\mathcal{H}$

Como mostrado na secção anterior, a função $\mathcal{F}[\![\mathcal{H}]\!]$ é a função binária que ao receber um código de um programa URM P e um código de uma sequência de argumentos, devolve o resultado obtido pela execução pela máquina URM desse programa com esses argumentos situados nos registos iniciais. Qualquer programa URM, se convergir, deixa o resultado no primeiro registo. Se divergir,

a computação não termina, sendo que o valor da função $\mathcal{F}[\![\mathcal{H}]\!]$ neste ponto não está definido.

Se o programa convergir, o cálculo do estado final da máquina é resultado do número de iterações necessárias para concluir a execução do programa. Isto é dado pela expressão[9],

$$N = \mu_z \left(\left(\left(\text{EXEC}(2^x 3^{2^1 y}, z) \right)_2 \right)_1 = 0 \right)$$

ou seja, no momento em que a próxima instrução tem código zero.

Sabendo o número N, o valor da função $\mathcal{F}[\![\mathcal{H}]\!](x,y)$ é o conteúdo do primeiro registo de memória depois da execução dessas N iterações, i.e.,

$$\mathcal{H}(x,y) = \left(\left(\text{EXEC}(2^x 3^{2^1 y}, N) \right)_2 \right)_2$$

Obtemos assim, uma descrição universal $\mathcal{H} \in \mathcal{R}$.[10]

Teorema 3.2.4: Para qualquer função parcial recursiva $f:\mathbb{N}^n \rightarrow \mathbb{N}$, existe pelo menos uma descrição $\ell \in \mathcal{R}$ tal que $\mathcal{F}[\![\ell]\!] = f$.

Demonstração: Seja um programa URM P que computa os valores da função $f(x_1,...,x_n)$. Seja $x = \alpha_P(P)$ e $y = \alpha_{arg}(x_1,...,x_n)$. Tem-se pela descrição universal que $\mathcal{F}[\![\mathcal{H}]\!](x,y) = f(x_1,...,x_n)$. Ao apresentar uma descrição $\ell \in \mathcal{R}$ tal que $\mathcal{F}[\![\ell]\!](y) = \mathcal{F}[\![\mathcal{H}]\!](x,y)$ a prova fica concluída.

Para achar ℓ é necessário compor $\mathcal{H}$ com os argumentos x e y. Seja X = S(S(S(....(S(W))....))) onde se aplicou a função sucessor x vezes. Assim, ℓ = C(R(X,$U_{2,2}$), $U_{1,1}$, $\mathcal{H}$), obtendo-se a descrição que descreve a função f pretendida, $\mathcal{F}[\![\ell]\!] = f$. ◁

---

[9] A minimização só é utilizada para descobrir N, facto que mostra que a prova axiomática de qualquer função computável precisa de aplicar a regra da minimização, no máximo, uma única vez.

[10] Se aplicarmos o processo de tradução das descrições para programas URM à descrição $\mathcal{H}$, obtemos um programa URM capaz de simular a computação de qualquer outro programa!

Construção da descrição de uma função parcial recursiva

Nem sempre é necessário recorrer à descrição universal para obter o resultado de uma dada função parcial recursiva aplicada a uma sequência de argumentos. Em muitos casos, existem descrições em $\mathcal{R}$ mais simples que a descrição universal, que resolvem o mesmo problema.

O processo apresentado de seguida procura decompor a função inicial em funções mais simples por aplicação da composição, da recorrência ou da minimização. Quando todas as subfunções são descritas directamente pelas descrições W, S ou $U_{i,n}$, o processo é reconstruído para se obter a descrição final. Neste sentido, diz-se que W, S e $U_{i,n}$ são axiomas e C, R e M, regras de construção.

Usaremos a notação-λ (apresentada na próxima definição) para simplificar a exposição do processo.

notação-λ

Definição 3.2.5: Seja uma função com argumentos x_1, …, x_n cujo valor é dado pela expressão E. A descrição da função utilizando notação-λ é $\lambda x_1.\lambda x_2 \ldots \lambda x_n.E$. A aplicação dos argumentos $a_1,a_2,\ldots,a_n$ na função $\lambda x_1.\lambda x_2 \ldots \lambda x_n.E$, é dada pela expressão $(\lambda x_1.\lambda x_2 \ldots \lambda x_n.E)a_1\, a_2 \ldots a_n$. ◁

Exemplo 3.2.6: O sucessor é, na notação-λ, descrito por λx.x+1. A soma binária por λx.λy.x+y. O resultado de (λx.λy.x+y)3 4 é 7.

Exemplo 3.2.7: Vejamos o exemplo da adição binária λx.λy.x+y. A adição tem como descritor central a recorrência. Esta divide o problema em duas funções mais simples, λx.x e λx.λy.λz.z+1. O processo continua até que todas as subfunções daí decorrentes são descritas por um dos axiomas.

$$\dfrac{\dfrac{\lambda x.\lambda y.x+y}{\dfrac{\lambda x.x}{}\,U_{1,1}\,\dfrac{\lambda x.\lambda y.\lambda z.z+1}{\dfrac{\lambda x.\lambda y.\lambda z.z}{}\,U_{3,3}\,\dfrac{\lambda x.x+1}{}\,S}\,C}\,R}{}$$

fig. 3.2.8 : método para encontrar uma descrição da adição binária.

Quando a decomposição termina, a descrição é encontrada ao reunir os vários axiomas e regras de construção numa expressão única. Assim, a soma binária é descrita por $R(U_{1,1},C(U_{3,3},S))$.

Exemplo 3.2.9: Para encontrar uma descrição da multiplicação $\lambda x.\lambda y.x*y$ encontramos a seguinte prova:

$$\dfrac{\lambda x.\lambda y.x*y}{\dfrac{\lambda x.0}{\dfrac{0}{W}}R \quad \dfrac{\lambda x.\lambda y.\lambda z.x+z}{\dfrac{\lambda x.\lambda y.y}{U_{2,2}} \quad \dfrac{\lambda x.\lambda y.\lambda z.x}{U_{1,3}} \quad \dfrac{\lambda x.\lambda y.\lambda z.z}{U_{3,3}} \quad \dfrac{\lambda x.\lambda y.x+y}{T}}C}R$$

fig. 3.2.10 : método para encontrar uma descrição da multiplicação binária.

Após reconstruir o processo, a descrição da multiplicação é dada pela expressão $R(R(W,U_{2,2}),C(U_{1,3},U_{3,3},T)$, sendo T a descrição da soma discutida no exemplo anterior.

3.3. Computação de descrições $\mathcal{R}$ em Redes σ

Na demonstração do teorema 3.1.6 mostramos como se pode tornar operacional o conceito das funções parciais recursivas, utilizando a máquina URM. Mostramos em seguida, como utilizar o conceito de computação neuronal para atingir o mesmo objectivo. É de realçar duas consequências importantes.

Primeiro, ao estabelecer uma relação entre redes neuronais e funções parciais recursivas, estamos a afirmar a universalidade do modelo neuronal utilizado, as redes σ. Ainda neste caso, como já foi apresentado um mecanismo de construção de uma descrição universal, conclui-se que existe uma rede universal.

Segundo, o próprio carácter das descrições $\mathcal{R}$ indica uma forte componente modular, dado que a cada função está associada uma descrição autónoma que pode ser utilizada num contexto mais geral (como foi o caso no exemplo 3.2.9, da descrição da adição no processo construtivo da descrição da multiplicação). É esta noção de modularidade intrínseca que dá relevo e originalidade a esta abordagem.

Veremos nos capítulos seguintes como este conceito pode ser aproveitado e generalizado.

Estrutura da Rede Final

A estrutura geral da rede é composta por três módulos sequenciais,

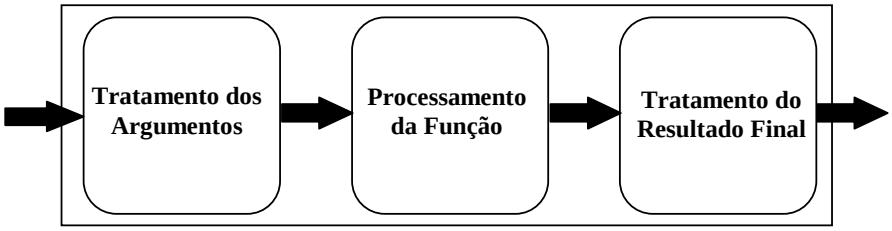


fig. 3.3.1 : estrutura geral da rede.

Para a emulação de uma função na rede neuronal respectiva é necessário transformar os argumentos da função numa representação adequada. Uma vez completada essa fase, o processamento da função é iniciado. Concluída a computação, o resultado da execução da rede é tratado (realizando o processo inverso ao ocorrido na primeira parte) e enviado pelo canal de saída.

Sincronização Interna da Rede

Todas as subredes constituintes da rede final possuem um mecanismo interno de sincronização. Este mecanismo, referenciado nos diagramas pelas palavras IN e OUT, define o momento em que a rede começa a processar a informação (entra o sinal 1 pelo canal IN) e quando o resultado final está disponível (sai o sinal 1 pelo canal OUT).

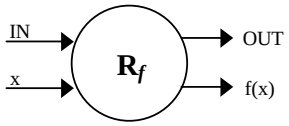


fig. 3.3.2 : sincronização de uma rede que emula $f(x) \in \mathcal{R}$.

É garantida a entrada simultânea dos dados e do sinal IN, bem como a saída simultânea do resultado da função e do sinal OUT.

É utilizado outro tipo de sincronização cada vez que for necessário esperar até que um conjunto de resultados fique disponível. Para isso, utilizar-se-á a rede tipo seguinte,

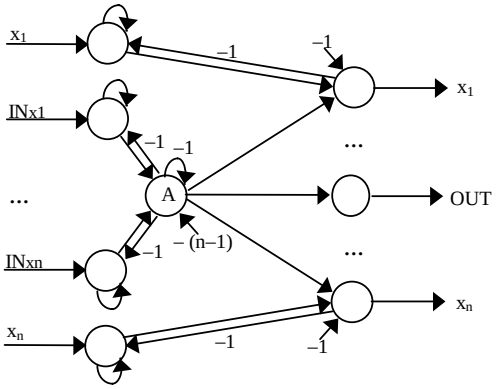


fig. 3.3.3 : sincronização de n entradas.

Esta rede guarda os valores que recebeu até que os n resultados estejam disponíveis. Isso é realizado pelo neurónio A, que só permite o fluxo de informação quando os n neurónios que guardam o sinal proveniente dos canais IN estiverem a 1. Também aqui é enviado um sinal OUT simultaneamente com os n resultados.

Aqui, como em todas as redes que se seguem, os arcos sem um número associado possuem peso sináptico 1.

Representação Numérica da Informação

É preciso determinar qual será a representação do domínio e do contradomínio das funções recursivas. Qualquer função parcial recursiva *f* está definida em $\mathbb{N}$. Considerando a arquitectura dos neurónios, a representação dos números naturais terá de se situar no interior do intervalo real [0,1].

$f:\mathbb{N}^n \rightarrow \mathbb{N}$

Definição 3.3.4: Uma *codificação* α de um domínio D de objectos num domínio E, é uma injecção $\alpha:D \rightarrow E$. Um objecto $d \in D$ é codificado pelo elemento $\alpha(d) \in E$. ◁

codificação

A codificação pretendida é dada pela função $\alpha:\mathbb{N}\rightarrow]0,1[$,

$$\alpha(n) = \sum_{i=1}^{n+1} 10^{-i} \qquad (9)$$

i.e.,

$$\alpha(0) = 0.1,\ \alpha(1) = 0.11,\ \alpha(2) = 0.111,\ \ldots,\ \alpha(n) = 0.1^{n+1}$$

codificação unária

Escolheu-se uma *codificação unária* do conjunto dos naturais por facilitar em larga medida a construção das redes neuronais. O seu principal defeito em comparação, por exemplo, com uma codificação binária, é o crescimento linear do número de casas decimais à medida que se guardam números progressivamente maiores, em comparação com o crescimento logarítmico de uma representação de base superior. No entanto, isso não é essencial, dado que o nosso objectivo é estabelecer limites inferiores do seu poder computacional.

conjunto A_N

Definição 3.3.5: O conjunto $A_N = \{ \alpha(n) : n \in \mathbb{N} \}$. ◁

Observação 3.3.6: A_N é um subconjunto de [0,1].

rede do sinal

Exemplo 3.3.7: A seguinte rede indica se o número que recebe é positivo ou zero, enviando o sinal 1 pelo canal respectivo.

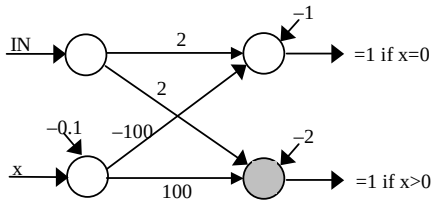


fig. 3.3.8 : rede do sinal (Sg).

Se entrar pelo canal x um número positivo, então $\alpha(x)>0.1$. Ao subtrair ao número 0.1 e multiplicar o resultado por 100, obtemos 1 se de facto o número inicial é positivo, ou zero caso contrário. É esse o sinal que sai do neurónio representado a cinzento.

Exemplo 3.3.9: A seguinte rede calcula o sucessor do número inicial.

rede do sucessor

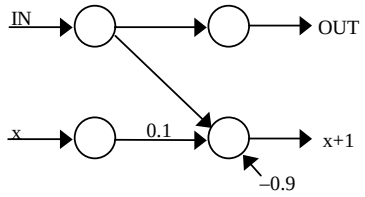


fig. 3.3.10 : rede do sucessor (Succ).

O sucessor é calculado por $\alpha(x+1) = 0.1\ \alpha(x) + 0.1$.

Canais de Entrada/Saída

Qualquer módulo, para poder ser útil e acessível, necessita de ter a possibilidade de comunicar com o exterior, seja com o ambiente, seja com outros módulos relacionadas e interdependentes. Utilizamos aqui um processo inspirado no mecanismo de entrada e saída de informação de [Siegelmann e Sontag 95].

Definição 3.3.11: Cada rede R_f tem dois canais binários de entrada de informação por cada argumento da função recursiva *f* a emular: canais D_{IN}, denotados por *canais de entrada de dados* e canais V_{IN}, denotados por *canais de entrada de validação*. A rede tem ainda dois canais binários de saída de informação: um canal D_{OUT}, denotado por *canal de saída de dados* e um canal V_{OUT}, denotado por *canal de saída de validação*. ◁

canais D_{IN} *e* V_{IN}

canais D_{OUT} *e* V_{OUT}

Os canais de entrada de validação são utilizados para indicar à rede em que momento a informação está disponível nos canais de entrada de dados respectivos e durante quanto tempo ela é processada nesses canais.

O canal de validação está inicialmente a zero. Quando os dados estão disponíveis passa a 1, mantendo-se assim, durante todo o período de entrada de informação. No momento imediatamente a seguir ao fim da transmissão dos dados, o canal de validação volta ao estado zero, i.e.,

$$\forall_{t<I} \quad : V_{IN}(t) = 0 \wedge D_{IN}(t) = 0$$
$$\forall_{t\in I..I+\Delta I} \quad : V_{IN}(t) = 1 \wedge D_{IN}(t) = 1 \qquad (10)$$
$$\forall_{t>I+\Delta I} \quad : V_{IN}(t) = 0 \wedge D_{IN}(t) = 0$$

sendo I o momento de início da transmissão de dados e ΔI o período de transmissão.

O canal de saída funciona de igual forma. Quando o resultado da função está disponível é enviado pelo canal D_{OUT} enquanto o canal V_{OUT} emite o sinal 1.

Codificação dos Argumentos da Função

Apresenta-se a rede que recebe um argumento da função $f(x_1,\ldots,x_n)$ através do par (D_{IN},V_{IN}) e devolve o valor codificado. Para o argumento $x_i \in \mathbb{N}$, serão enviados x_i+1 sinais 1 pelos canais (D_{IN},V_{IN}). O valor produzido pela rede será $\alpha(x_i)$. O canal de sincronização OUT envia para a rede de processamento o sinal que o valor $\alpha(x_i)$ está disponível naquele instante.

Rede de Codificação
Em cada instante que é introduzido um novo valor (=1), um valor é multiplicado por 0.1 e somado mais 0.1. Deste modo, o valor terá um número de uns igual ao tempo (número de instantes) gasto pela transmissão, codificando assim, o argumento pretendido.

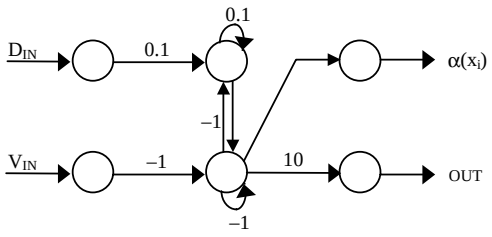


fig. 3.3.12 : rede que recebe o i-ésimo argumento.

Existe uma rede igual a esta para cada argumento da função $f(x_1,\ldots,x_n)$. Todas estas redes estarão ligadas a uma rede de sincronização como a da figura 3.3.3.

Descodificação do Resultado da Função

Uma vez concluído o processamento, a rede possui o resultado da função codificado. A partir desse valor calcula-se o resultado que é enviado

temporalmente pelos canais (D_{OUT},V_{OUT}) durante $f(x_1,\ldots,x_n)+1$ períodos de tempo.

Esta descodificação é realizada pela seguinte rede.

Rede de Descodificação
É somada uma unidade ao valor final usando o sinal de entrada para eliminar apenas 0.9 do pendor. Este novo valor é subtraído por 1, sendo enviado por ambos os canais de saída, enquanto não for zero. Isso demora o número de instantes que o argumento codificava na entrada da rede.

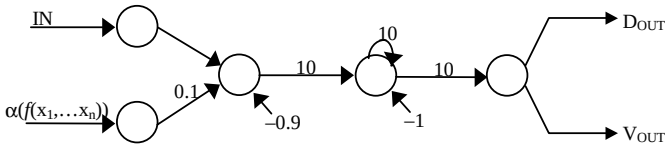


fig. 3.3.13 : rede que envia o resultado de $f(x_1,\ldots,x_n)$.

Rede de Processamento

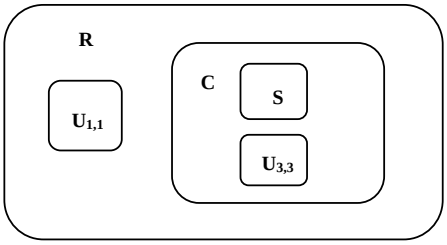
A construção da rede de processamento da função *f* em questão é organizada recursivamente conforme a função parcial recursiva utilizada para a definir. Essa expressão é o resultado de uma sequência finita de utilizações das regras de construção sobre um ou mais dos axiomas iniciais.

Conhecendo à partida como se constrói, por exemplo, uma minimização, basta-nos conhecer a subrede que emula a função a aplicar sobre a minimização para construir a rede final. E assim sucessivamente, dado que essa subrede denota por sua vez uma função recursiva.

A compilação da expressão constrói redes de forma modular, simplificando as expressões até chegar aos axiomas. No fim, o compilador encaixa as diversas redes numa única rede que emula a função inicial.

Exemplo 3.3.14: A adição binária pode ser descrita por $R(U_{1,1},C(U_{3,3},S))$ (*vide.* exemplo 3.2.7). A rede central executará uma recursão, que por sua vez operará sobre uma projecção e uma composição. A composição, por sua vez, compõe o sucessor sobre uma projecção.

O esquema geral da rede produzida será:



Construção Top-Down

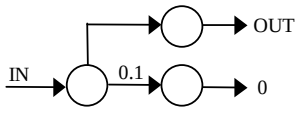
Os módulos são inseridos recursivamente desde o módulo central até às unidades mais básicas, i.e., um dos axiomas do sistema formal utilizado.

fig. 3.3.15 : esquema geral da rede que computa a função descrita por $R(U_{1,1},C(U_{3,3},S))$.

Neste ponto, resta conhecer as redes que calculam os axiomas e implementam as regras.

Os Axiomas

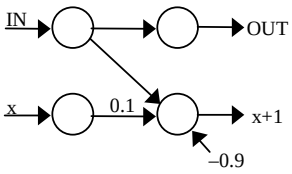
Estas são as redes que executam os três axiomas descritos. Os axiomas das projecções denotam um conjunto de funções definidas pelo par (i,n).



Rede W

O sinal IN (=1) é usado para codificar o código do valor zero, i.e., o racional 0.1

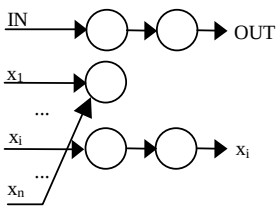
fig. 3.3.16 : Axioma W – função Zero.



Rede S

O sinal IN (=1) é usado para eliminar o pendor de forma a sobrar apenas 0.1. Este valor é somado ao produto de 0.1 por x, o que resulta no incremento de uma unidade.

fig. 3.3.17 : Axioma S – função Sucessor.



Rede $U_{i,n}$

Esta rede somente faz a transferência do valor determinado (o i-ésimo argumento), eliminando a actividade de todas as outras entradas. O sinal IN é atrasado de forma a obter-se a sincronização apropriada.

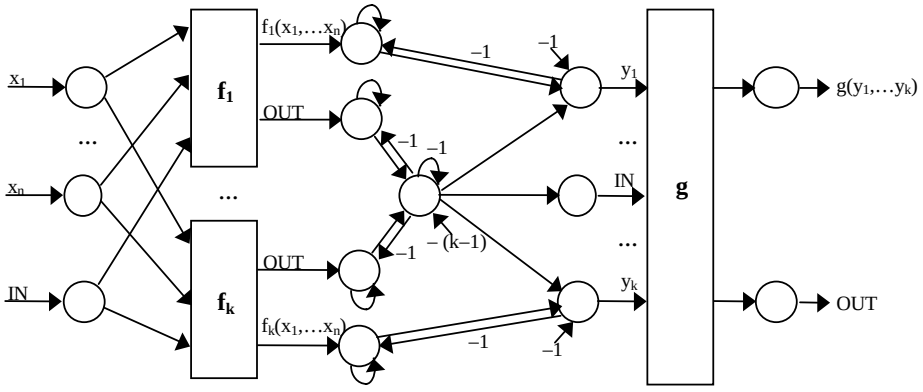
fig. 3.3.18 : Axioma $U_{i,n}$ – função Projecção.

Regra da Composição

O funcionamento desta regra é muito simples. Quando pretendemos calcular $g(f_1(\ldots), \ldots, f_k(\ldots))$, basta que calculemos os valores dos f_i (podendo este cálculo ser realizado em paralelo), esperar que todos os resultados estejam disponíveis (usando um mecanismo de sincronização como o da figura 3.3.3) e finalmente introduzir todos os f_i como argumentos da função g, obtendo-se assim, o resultado pretendido.

A rede que implementa esta regra de construção é dada pelo esquema que se segue. Não é possível apresentar a estrutura neuronal completa, dado que, para cada conjunto de funções componentes, os módulos constituintes serão diferentes.

Como referido anteriormente, o processo de conexão dos módulos é efectuado para que cada argumento seja transferido de forma correcta, desde o módulo central até às unidades mais básicas (ou seja, as redes que executam os axiomas). Assim, mantém-se coerente a função computada pela execução da rede final produzida com a descrição inicial desejada.



Rede Composição
Os argumentos são enviados em paralelo para cada um dos módulos das funções f. Como cada função tem o seu tempo de execução, é necessária uma rede de sincronização (vide 3.3.3*) para reter os resultados até todos estarem disponíveis. Nesse momento, a função central g é activada e o seu resultado é o resultado da rede da composição.*

fig. 3.3.19 : esquema da composição.

Exemplo 3.3.20: Seja F a descrição da função f. A composição permite obter funções como as seguintes:

h(x,y)	= f(y,x)	(reordenar variáveis – $C(U_{2,2},U_{1,2},F)$)
h(x)	= f(x,x)	(identificar variáveis – $C(U_{1,1},U_{1,1},F)$)
h(x,y,z)	= f(x,y)	(adicionar variáveis – $C(U_{1,3},U_{2,3},F)$)
h(x,y,z)	= f(f(x,y),z)	(aplicação sequencial – $C(C(U_{1,3},U_{2,3},F),U_{3,3},F)$)

Regra da Recursão

Na recursão define-se cada valor da função à custa dos valores previamente calculados. A regra da recursão pode ser decomposta no seguinte algoritmo.

```
K←0;
H←f(x1,…,xn);
while y>0 do
  begin
    H←g(x1,…,xn,K,H);
    K←K+1;
    y←y–1;
  end;
h(x1,…,xn,y)←H;
```

fig. 3.3.21 : algoritmo de recursão.

A variável K vai acumulando o número de iterações já realizadas para o cálculo da função, onde por sua vez o argumento *y* vai sendo decrementado em simultâneo. Quando *y* chegar a zero, significa que foram realizadas *y* cálculos da função (K=*y*), e o resultado final encontra-se em H.

Isto é simulado pela rede, onde X_i, Y, K, H são neurónios que guardam os valores respectivos das variáveis com o mesmo nome. O resto da estrutura é necessária para sincronizar o cálculo das funções *f* e *g*[11], para incrementar K e decrementar *y*. Denota-se *f* por função base e *g* por função passo.

função base, função passo

[11] Aqui representadas igualmente como caixas, na medida em que a sua estrutura é independente do contexto actual, do mesmo modo que o era na composição.

Optou-se por esconder os pormenores de implementação da rede sinal (Sg), para simplificar a visualização.

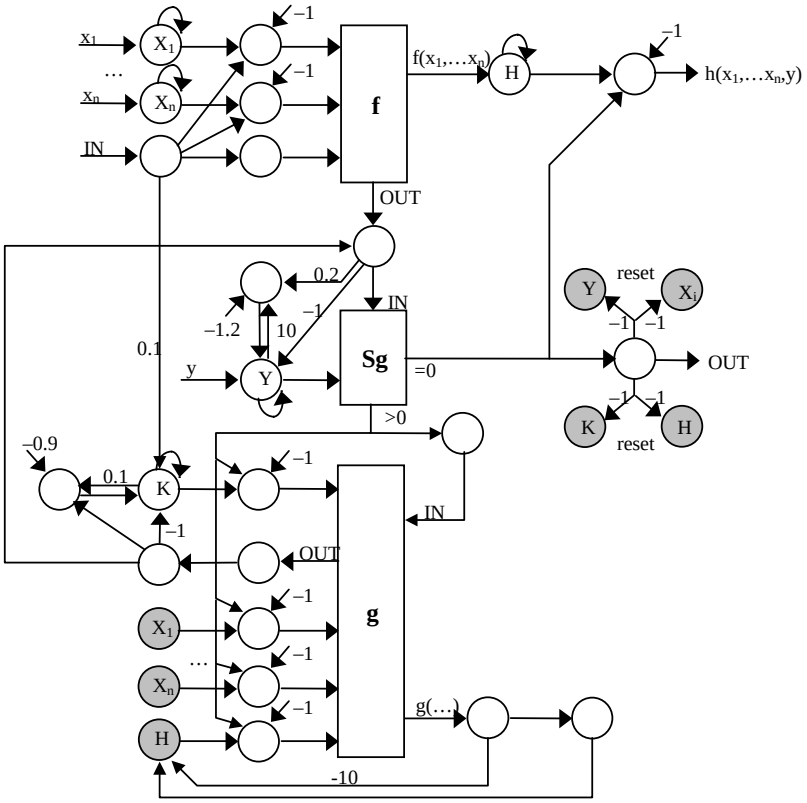


fig. 3.3.22 : esquema da recursão.

Regra da Minimização

A minimização devolve o menor valor de *y* que torne zero a função *f*(x_1,…,x_n,y), desde que todos as chamadas de *f* anteriores a esse mínimo terem convergido e devolvido valores positivos. Se *y* não existir, a função é indefinida nesse ponto.

Apresenta-se o algoritmo que executa a minimização.

Y←0;
while f(x_1,…,x_n,Y)≠0 **do** Y←Y+1;
h(x_1,…,x_n)←Y;

fig. 3.3.23 : algoritmo de minimização.

A rede vai calcular o valor da função para valores crescentes de Y (iniciando no zero), até que a igualdade seja satisfeita. Se tal Y não existir (a função *f* é parcial e não está definida nesse ponto), o algoritmo diverge, i.e., a computação não termina.

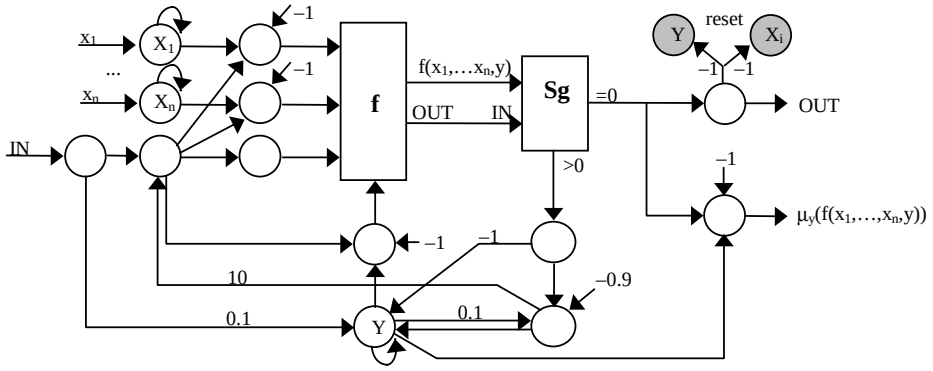


fig. 3.3.24 : esquema da minimização.

Dimensão das Redes Compiladas

Cada módulo, seja um dos axiomas, seja uma das regras de construção, possui uma dimensão bem definida, que faz com que seja fácil calcular a dimensão total da rede uma vez conseguida a descrição da função desejada. Para calcular o número de neurónios de uma dada rede R_ℓ, que calcula a função descrita por $\ell \in \mathcal{R}$, introduzimos a função dim : $\mathcal{R} \to \mathbb{N}$.

Definição 3.3.25: A função *dim* é uma aplicação de $\mathcal{R}$ em $\mathbb{N}$, que aplica a descrição $\ell \in \mathcal{R}$ no número de neurónios da rede R_ℓ produzida na compilação da sua respectiva prova axiomática. *dim*

$\dim(\ell) = 9n + 7 + F(\ell)$

$$F(\ell) = \begin{cases} 3 & , \ell \text{ é W} \\ 4 & , \ell \text{ é S} \\ 5 & , \ell \text{ é } U_{i,n} \\ 3k+n+5+\sum_{i=1}^{k+1} F(\ell_i) & , \ell \text{ é } C(\ell_1, \ldots, \ell_k, \ell_{k+1}) \\ 3n+21+F(\ell_1)+F(\ell_2) & , \ell \text{ é } R(\ell_1, \ell_2) \\ 2n+13+F(\ell_1) & , \ell \text{ é } M(\ell_1) \end{cases} \qquad (11)$$

◁

Exemplo 3.3.26: Para a adição binária, compilando a descrição $\ell_{\text{adição}} = R(U_{1,1},C(U_{3,3},S))$, tem-se que $\dim(\ell_{\text{adição}}) = 76$. Ou seja, a rede, contando já com o sistema de codificação de entrada e descodificação de saída, possui um total de 76 neurónios.

Exemplo 3.3.27: Para a multiplicação binária, tem-se a descrição $\ell_{\text{produto}} = R(R(W,U_{2,2}),C(U_{3,3},U_{1,3},R(U_{1,1},C(U_{3,3},S))))$. $\dim(\ell_{\text{produto}}) = 160$. Podia utilizar-se o módulo da adição para facilitar os cálculos deste exemplo (já que a soma é utilizada para o cálculo do produto). Nesse caso, ter-se-ia de descontar a parte de entrada/saída da adição, que são 9n+7=25 neurónios, ficando o módulo da soma com 51 neurónios.

Complexidade das redes compiladas

Analisaremos em primeiro lugar a complexidade temporal das estruturas neuronais apresentadas.

Cada um dos axiomas demora um tempo constante de execução. Assim, $\Theta(W)=1$, $\Theta(S)=1$ e $\Theta(U_{i,n})=1$. Para uma composição C, sabendo a complexidade superior dos seus argumentos, ou seja, sabendo $O(f_i)_{i=1..n}$ e $O(g)$, a complexidade de C é de $O(\max(O(f_1), ..., O(f_n), O(g)))$, i.e., não acrescenta complexidade adicional nas funções utilizadas.

Para a recursão $R(f, g)$, conhecendo $O(f)$ e $O(g)$, a complexidade é dada por $O(\max(O(f), n*O(g)))$, ou seja, introduz no máximo uma complexidade linear em relação à complexidade da função passo g utilizada. Para a minimização $M(f)$, dado $O(f)$, a complexidade é $O(n*O(g))$, ou seja, também insere uma componente de complexidade linear. Este acréscimo justifica-se pela existência de um ciclo em cada uma destas regras de construção. Do mesmo modo, num modelo operacional como a máquina de Turing, esta complexidade é similar. Assim, o sistema de construção neuronal proposto não acrescenta complexidade aos algoritmos que descreve.

Para analisar a complexidade espacial, neste caso o número de neurónios utilizados, basta verificar que a função *dim* definida em 3.3.25, introduz um número constante de neurónios para a entrada, a saída e a emulação dos axiomas e cresce linearmente para qualquer uma das regras de construção. Assim, a complexidade espacial é igualmente de ordem linear em relação ao algoritmo definido pela descrição axiomática.

A questão da complexidade relativa a problemas de reconhecimento de linguagens, usando redes neuronais é abordada em [Orponen 92], [Maass 96], [Siegelmann e Giles 97], [Maass e Sontag 99] e [Siegelmann 99].

3.4. Modularidade

Uma característica determinante na construção deste modelo é a modularidade das redes padrão. Cada rede tem um funcionamento interno relacionando-se com o exterior somente através dos canais de entrada e de saída. Nenhuma rede induz efeitos secundários sobre redes não relacionadas.

Ao atacar o problema de forma não sistemática, tentando construir redes capazes de realizar computações não triviais, surgem problemas de sincronização complexos e de difícil gestão. Ao construir a função problema como um conjunto de pequenas redes de fácil construção (e de verificação acessível relativamente à sua correcção), fomos levados a um sistema de sincronização geral e à noção informal de redes modulares. Outros trabalhos sobre modularidade em redes neuronais podem ser encontrados em [Caelli *et al*. 99], [Hrycej 92], [Hussain 95], [Boers *et al*. 93] e [Happe e Murrel 94].

No caso específico das seis redes propostas, sabemos exactamente que tipo de funções são implementadas, dado que basta observar a descrição da função parcial que cada uma define. Para o caso geral de uma rede neuronal modular baseada no sistema de sincronização proposto, é apropriado definir formalmente o conjunto de ideias aqui apresentado.

Definição 3.4.1: Um canal temporal de dados é uma função de domínio $\mathbb{N}$ e contradomínio A_N. Um canal temporal de dados de sincronização é uma função de domínio $\mathbb{N}$ e contradomínio $\{0,1\}$. ◁

canal temporal

Observação 3.4.2: O domínio dos naturais representa unidades de tempo, e os contradomínios referidos representam a informação transferida a cada instante.

Definição 3.4.3: Um módulo M é um tuplo <S, X, IN, RES, OUT> onde X é um conjunto de n≥0 canais temporais de dados $\{x_1, \ldots, x_n\}$ designados por canais de entrada de dados. RES é um canal temporal de dados designado por saída de dados. IN e OUT são canais temporais de sincronização designados por entrada de sincronização e saída de sincronização, respectivamente. S é um sistema dinâmico de equações não lineares descrito por uma rede finita de neurónios σ. ◁

módulo

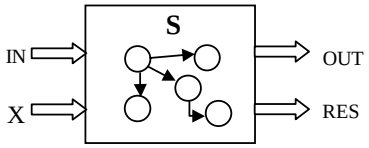


fig. 3.4.4 : elementos constituintes de um módulo.

Nestas redes, existe um conjunto de neurónios específicos que são ligados aos canais de entrada/saída. Cada canal envia a sua informação para um único neurónio responsável pela transferência dessa informação para o(s) neurónio(s) adequado(s) dentro da rede. Estes neurónios – com nomes comuns para todas as redes – chamam-se X_{IN}, X_{OUT}, X_{RES} e X_{Xi}, que ligam respectivamente os canais entrada de sincronização, saída de sincronização, saída de dados, e ao i-ésimo canal de entrada de dados.

O esquema de sincronização obriga a que todos os módulos respeitem as seguintes propriedades:

$$\underset{t\in\mathbb{N}}{\forall} : \mathrm{IN}(t) = 1 \Leftrightarrow x_i(t) \in A_N, \quad i=1, \ldots, n \tag{12}$$

$$\underset{t\in\mathbb{N}}{\forall} \; \underset{t_1\in\mathbb{N}}{\exists} : \mathrm{OUT}(t) = 1 \Rightarrow \mathrm{IN}(t-t_1) = 1 \wedge \underset{t_2\in(t-t_1+1)..t}{\forall} \mathrm{IN}(t_2) = 0 \tag{13}$$

$$\forall_{t \in \mathbb{N}} : \text{OUT}(t) = 1 \Leftrightarrow \text{RES}(t) \in A_N \qquad (14)$$

A primeira propriedade refere o facto da simultaneidade entre os canais de entrada de dados e os sinais de entrada de sincronização. Ou seja, só quando o canal de sincronização transferir o valor 1, é que a rede (ou um observador externo) deve ler a informação transmitida pelos canais de entrada de dados. A segunda propriedade indica que cada vez que um módulo enviar um sinal de saída, indicando a sua terminação, significa que houve pelo menos um momento no passado onde existiu um requerimento de processamento (ou seja, um sinal enviado pelo canal de sincronização de entrada) e que em todos os outros momentos intermédios, não existiu qualquer nova requisição sobre o módulo (isto garante que o módulo não está sendo executado mais do que uma vez num dado momento, essencial devido ao facto do módulo poder ter ciclos, o que comprometeria a sua correcção). Por fim, a terceira propriedade é semelhante à primeira, informando que também os canais de saída (de dados e de sincronização) transmitem as suas respectivas informações de forma simultânea.

É possível definir convergência de um módulo utilizando as propriedades de sincronização.

convergência de um módulo

Definição 3.4.5: Para uma entrada $\vec{x}$, um módulo *converge* se e só se $\text{IN}(0) = 1 \wedge X(0) = \vec{x} \Rightarrow \exists_{t \in \mathbb{N}} \text{OUT}(t) = 1$ ◁

Resta definir, uma vez terminada a execução do módulo, qual é o valor que esse módulo computa.

função associada de um módulo

Definição 3.4.6: Seja um módulo M que respeite as condições de sincronização (12) – (14). A *função associada* ao módulo M é a função [M] de domínio $\mathbb{N}\times\{0,1\}\times A_N^n$ e contradomínio A_N, definida pelas seguintes expressões:

$$[M](t,1,\vec{x}) = \{ \text{RES}(t+\Delta t_1) : \text{OUT}(t+\Delta t_1)=1 \wedge \forall_{0<\Delta t_2<\Delta t_1} \text{OUT}(t+\Delta t_2)=0 \}$$

$$[M](t,0,\vec{x}) = [M](t-\Delta t,1,\vec{x}) \text{ com } \text{IN}(t-\Delta t)=1 \wedge \forall_{t_1 \in (t-\Delta t+1)..t} \text{IN}(t_1)=0,\ \Delta t > 0$$

Os valores dos canais RES e OUT são calculados a partir do sistema de equações definidos por S. ◁

Observação 3.4.7: Apesar de não ser exigido, os módulos até aqui apresentados têm, para todos os neurónios em S, uma actividade nula no instante t=0, bem como entre os intervalos temporais $t+1..t+\Delta t-1$ tais que $\mathrm{OUT}(t)=1$ e $\mathrm{IN}(t+\Delta t)=1$.

Para o axioma W, os canais RES e OUT da função associada ao módulo correspondente são dados pelo seguinte sistema de equações:

$$
\begin{aligned}
X_{IN}^{+} &= \sigma(\mathrm{IN}) \\
X_{RES}^{+} &= \sigma(0.1\, X_{IN}) \\
X_{OUT}^{+} &= \sigma(X_{IN}) \\
\mathrm{RES}^{+} &= \sigma(X_{RES}) \\
\mathrm{OUT}^{+} &= \sigma(X_{OUT})
\end{aligned}
$$

Para o axioma S, os canais RES e OUT da função associada ao módulo correspondente são dados pelo seguinte sistema de equações:

$$
\begin{aligned}
X_{IN}^{+} &= \sigma(\mathrm{IN}) \\
X_{X1}^{+} &= \sigma(X_1) \\
X_{RES}^{+} &= \sigma(X_{IN} + 0.1\, X_{X1} - 0.9) \\
X_{OUT}^{+} &= \sigma(X_{IN}) \\
\mathrm{RES}^{+} &= \sigma(X_{RES}) \\
\mathrm{OUT}^{+} &= \sigma(X_{OUT})
\end{aligned}
$$

Para o axioma $U_{i,n}$, os canais RES e OUT da função associada ao módulo correspondente são dados pelo seguinte sistema de equações:

$$
\begin{aligned}
X_{IN}^{+} &= \sigma(\mathrm{IN}) \\
X_{Xi}^{+} &= \sigma(X_i) \\
X_1^{+} &= \sigma(X_1 + \ldots + X_{i+1} + X_{i+1} + \ldots + X_n) \\
X_{RES}^{+} &= \sigma(X_{Xi}) \\
X_{OUT}^{+} &= \sigma(X_{IN}) \\
\mathrm{RES}^{+} &= \sigma(X_{RES}) \\
\mathrm{OUT}^{+} &= \sigma(X_{OUT})
\end{aligned}
$$

Do mesmo modo se calculam as funções associadas dos módulos que implementam a composição, a recorrência e a minimização.

Semântica

Com esta definição podemos associar a cada módulo uma semântica no contexto das funções de domínio e contradomínio natural.

Definição 3.4.8: A *função computada* por um módulo M, é a função $[M]:\mathbb{N}^n \rightarrow \mathbb{N}$ definida pela expressão $\alpha^{-1}([M](0,1,<\alpha(x_1), \alpha(x_2), \ldots, \alpha(x_n)>))$, sendo α a função de codificação dos naturais para o conjunto A_N apresentada em 3.3.4. ◁

função computada por um módulo

$$\alpha(x) = \sum_{i=1}^{n+1} 10^{-i}$$

Teorema 3.4.9: Seja um módulo R_ℓ construído a partir do processo de compilação da descrição $\ell \in \mathcal{R}$. A função associada $[R_\ell]$ é dada pela função $\mathcal{F}[\![\ell]\!]$, ou seja,

$$\forall_{\ell \in \mathcal{R}} : [R_\ell] = \mathcal{F}[\![\ell]\!]. \qquad (15)$$

Demonstração: A prova será efectuada por indução na estrutura das descrições capazes de construir qualquer descrição ℓ. O passo da indução é obtido pela correcção de cada um dos módulos usados pelos axiomas e regras de construção.

- Axioma W

Para ℓ = W, a função $\mathcal{F}[\![\ell]\!] = \mathcal{F}[\![\,\mathrm{W}\,]\!] = 0$.

$[R_\ell] = [R_W]$ por definição, é igual à função $\alpha^{-1}([R_W](0,1))$, ou seja, o resultado descodificado do neurónio RES da rede R_W, no primeiro instante em que o neurónio OUT produz o valor 1, tendo em conta que no instante t=0 se insere o valor 1 no neurónio IN (estando a rede inactiva nesse instante). Assim, basta verificar qual é o primeiro instante t tal que OUT(t) = 1. O valor de $[R_W](0,1)$ será igual a RES(t).

Relembrando a descrição da rede R_W:

$X_{IN}^{+} = \sigma(\text{IN})$
$X_{RES}^{+} = \sigma(0.1\ X_{IN})$
$X_{OUT}^{+} = \sigma(X_{IN})$
$\text{RES}^{+} = \sigma(X_{RES})$
$\text{OUT}^{+} = \sigma(X_{OUT})$

A evolução temporal de R_W é apresentada na seguinte tabela:

instante t	0	1	2	3
neurónio				
x_{IN}	0	1	0	0
x_{RES}	0	0	0.1	0
x_{OUT}	0	0	1	0
canal				
IN	1	0	0	0
RES	0	0	0	**0.1**
OUT	0	0	0	**1**

tab. 3.4.1: dinâmica de R_W.

No instante t=3, o canal OUT tem o valor 1. Assim, $[R_W](0,1) = \text{RES}(3) = 0.1$.
Logo, $[R_W] = \alpha^{-1}([R_W](0,1)) = \alpha^{-1}(0.1) = 0$.

Prova-se assim, que para $\ell = W$, $[R_\ell] = \mathcal{F}[\![\ell]\!]$.

- Axioma S

Para $\ell = S$, a função $\mathcal{F}[\![\ell]\!] = \mathcal{F}[\![\ S\ ]\!](x) = x+1$.

Por definição, $[R_\ell](x) = [R_S](x) = \alpha^{-1}([R_S](0,1,\alpha(x)))$

Relembrando a descrição da rede R_S:

$X_{IN}^{+} = \sigma(\text{IN})$
$X_{X1}^{+} = \sigma(X_1)$
$X_{RES}^{+} = \sigma(X_{IN} + 0.1\ X_{X1} - 0.9)$
$X_{OUT}^{+} = \sigma(X_{IN})$
$\text{RES}^{+} = \sigma(X_{RES})$
$\text{OUT}^{+} = \sigma(X_{OUT})$

A evolução temporal de R_S é apresentada na seguinte tabela:

instante t	0	1	2	3
neurónio				
x_{IN}	0	1	0	0
x_{X1}	0	$\alpha(x)$	0	0
x_{RES}	0	0	$0.1\alpha(x)+0.1$	0
x_{OUT}	0	0	1	0
canal				
IN	1	0	0	0
X_1	$\alpha(x)$	0	0	0
RES	0	0	0	**$0.1\alpha(x)+0.1$**
OUT	0	0	0	**1**

tab. 3.4.2: dinâmica de R_S.

No instante t=3, o canal OUT tem o valor 1. Assim, $[R_W](0,1,\alpha(x)) = \text{RES}(3) = 0.1\alpha(x)+0.1 = \alpha(x+1)$. Logo, $[R_W](x) = \alpha^{-1}([R_W](0,1,\alpha(x))) = \alpha^{-1}(\alpha(x+1)) = x+1$.

Prova-se assim, que para $\ell = S$, $[R\ell] = \mathcal{F}[\![\ell]\!]$.

- Axioma $U_{i,n}$

Para $\ell = U_{i,n}$, a função $\mathcal{F}[\![\ell]\!](x_1,\ldots,x_n) = \mathcal{F}[\![U_{i,n}]\!](x_1,\ldots,x_n) = x_i$.

Por definição, $[R\ell](x_1,\ldots,x_n) = [R_{Ui,n}](x_1,\ldots,x_n) = \alpha^{-1}([R_{Ui,n}](0,1,<\alpha(x_1),\ldots, \alpha(x_n)>))$

Relembrando a descrição da rede $R_{Ui,n}$:

$X_{IN}^{+} = \sigma(IN)$
$X_{Xi}^{+} = \sigma(X_i)$
$S_1^{+} = \sigma(X_1 + \ldots + X_{i+1} + X_{i+1} + \ldots + X_n)$
$X_{RES}^{+} = \sigma(X_{Xi})$
$X_{OUT}^{+} = \sigma(X_{IN})$
$RES^{+} = \sigma(X_{RES})$
$OUT^{+} = \sigma(X_{OUT})$

A evolução temporal de $R_{Ui,n}$ é apresentada na seguinte tabela:

instante t	0	1	2	3
neurónio				
x_{IN}	0	1	0	0
x_{Xi}	0	$\alpha(x_i)$	0	0
S_1	0	$\alpha(x_1)+\ldots+\alpha(x_n)$	0	0
x_{RES}	0	0	$\alpha(x_i)$	0
x_{OUT}	0	0	1	0
canal				
IN	1	0	0	0
X_i	$\alpha(x_i)$	0	0	0
$X_{j\ (j\neq i)}$	$\alpha(x_j)$	0	0	0
RES	0	0	0	$\boldsymbol{\alpha(x_i)}$
OUT	0	0	0	**1**

tab. 3.4.3: dinâmica de $R_{Ui,n}$.

No instante t=3, o canal OUT tem o valor 1. Assim, $[R_{Ui,n}](0,1,<\alpha(x_1),\ldots, \alpha(x_n)> = \text{RES}(3) = \alpha(x_i)$. Logo, $[R_{Ui,n}](x) = \alpha^{-1}([R_{Ui,n}](0,1,<\alpha(x_1),\ldots, \alpha(x_n)>) = \alpha^{-1}(\alpha(x_i)) = x_i$.

Prova-se assim, que para $\ell = U_{i,n}$, $[R\ell] = \mathcal{F}[\![\ell]\!]$.

- Composição

Sejam as descrições G, F_1, …, F_k das funções g, f_1, …, f_k utilizadas na composição. Para $\ell = C(F_1,\ldots,F_k,G)$, a função $\mathcal{F}[\![\ell]\!](x_1,\ldots,x_n) = \mathcal{F}[\![C(F_1, \ldots, F_k,G)]\!](x_1,\ldots,x_n) = g(f_1(x_1,\ldots,x_n), \ldots, f_k(x_1,\ldots,x_n))$.

Por definição, $[R\ell](x_1,\ldots,x_n) = [R_{C(F1,\ldots,Fk,G)}](x_1,\ldots,x_n) = \alpha^{-1}([R_{C(F1,\ldots,Fk,G)}](0,1, <\alpha(x_1),\ldots, \alpha(x_n)>))$

A descrição da rede $R_{C(G,F1,\ldots,Fk)}$:

$X_{IN}^{+} = \sigma(IN)$
$X_{Xi}^{+} = \sigma(X_i),\ i=1,\ldots,n$
$S_{Fj}^{+} = \sigma(RES_{Fj} + S_{Fj} - S_{yj}),\ j = 1,\ldots,k$
$S_{outFj}^{+} = \sigma(OUT_{Fj} + S_{outFj} - S_{wait}),\ j = 1,\ldots,k$
$S_{wait}^{+} = \sigma(S_{outF1} + \ldots + S_{outFk} - (k-1) - S_{wait})$
$S_G^{+} = \sigma(S_{wait})$
$S_{yj}^{+} = \sigma(S_{Fj} + S_{wait} - 1)$, $j = 1,\ldots,k$
$X_{RES}^{+} = \sigma(RES_G)$
$X_{OUT}^{+} = \sigma(OUT_G)$
$RES^{+} = \sigma(X_{RES})$
$OUT^{+} = \sigma(X_{OUT})$

No instante t=2, os canais de entrada dos módulos das funções f_j recebem os valores respectivos, sendo assim, activados. Considere-se que todas as funções g e f_j convergem (no caso contrário, a rede divergia não produzindo um valor final, facto coerente com a indefinição de $\mathcal{F}[\![\ell]\!](x_1,\ldots,x_n)$ nesse caso).

Seja t_{Fj} o tempo de processamento necessário ao módulo que computa a função f_j. Para cada j = 1, …, k, $S_{Fj}(t_{Fj}+2) = 0$ e $S_{Fj}(t_{Fj}+3) = \alpha(f_j(x_1,\ldots,x_n))$, ou seja, a partir do instante $t_{Fj} + 3$, o neurónio S_{Fj} guarda o valor $f_j(x_1,\ldots,x_n)$. Observa-se pela equação do neurónio que o seu valor é preservado enquanto não houver uma activação do neurónio S_{yj}. Do mesmo modo, os neurónios que guardam o resultado OUT das funções, S_{outFj}, preservam o valor de activação 1.

Seja $t_f = \max(t_{F1},\ldots,t_{Fk})$. Desde o instante t=2 até $t=t_f - 1$ só existe no máximo, k–1 módulos terminados, o que implica que o neurónio S_{wait} se mantém inactivo (devido ao pendor -(k-1)). No instante t_f, a activação de S_{wait} é 1.

No instante t_f +1, a activação dos neurónios S_{yj} vai ser de $\alpha(f_j(x_1,\ldots,x_n))$ e do neurónio S_G, vai ser de 1. Estes valores são introduzidos nos canais de entrada do módulo G. O neurónio x_{RES} vai guardar o resultado da execução do módulo, ou seja, vai receber o valor $[R_G](0,1,<\alpha(f_1(x_1,\ldots,x_n)),\ldots, \alpha(f_k(x_1,\ldots,x_n))>)$ no mesmo instante em que x_{OUT} recebe 1. Este valor de x_{RES} é precisamente o valor da função associada ao módulo G, ou seja, $\alpha(\ g(\ f_1(x_1,\ldots,x_n), \ldots, f_k(x_1,\ldots,x_n)\ )\ )$.

Logo, $[R_{C(F1,\ldots,Fk,G)}](x_1,\ldots,x_n) = \alpha^{-1}([R_{C(F1,\ldots,Fk,G)}](0,1,<\alpha(x_1),\ldots, \alpha(x_n)>) = [R_G](0,1,<\alpha(f_1(x_1,\ldots,x_n)), \ldots, \alpha(f_k(x_1,\ldots,x_n))>) = \alpha^{-1}(\alpha(\ g(\ f_1(x_1,\ldots,x_n), \ldots, f_k(x_1,\ldots,x_n)\ )\ )) = g(\ f_1(x_1,\ldots,x_n), \ldots, f_k(x_1,\ldots,x_n)\ )$.

Assim, para $\ell = C(F_1,\ldots,F_k,G)$, $[R\ell] = \mathcal{F}[\![\ell]\!]$.

- Recorrência

Sejam as descrições F e G das funções f e g utilizadas na recursão. Para ℓ = R(F, G), a função $\mathcal{F}[\![\ell]\!](x_1,\ldots,x_n,y) = \mathcal{F}[\![R(F, G)]\!](x_1,\ldots,x_n,y)$ é definida por:

$\mathcal{F}[\![R(F, G)]\!](x_1,\ldots,x_n,0) = \mathcal{F}[\![F]\!](x_1,\ldots,x_n)$

$\mathcal{F}[\![R(F, G)]\!](x_1,\ldots,x_n,y+1) = \mathcal{F}[\![G]\!](x_1,\ldots,x_n, y, \mathcal{F}[\![R(F, G)]\!](x_1,\ldots,x_n, y))$

Por definição, $[R_\ell](x_1,\ldots,x_n,y) = [R_{R(F,G)}](x_1,\ldots,x_n,y) = \alpha^{-1}([R_{R(F,G)}](0,1, <\alpha(x_1),\ldots, \alpha(x_n),\alpha(y)>))$

A descrição da rede $R_{R(F,G)}$:

$X_{IN}^{+} = \sigma(IN)$
$X_{Xi}^{+} = \sigma(X_i + X_{Xi} - X_{OUT}),\ i = 1,\ldots,n$
$X_{Y}^{+} = \sigma(Y + X_Y + S_{y1} + S_{outF} - X_{OUT})$
$S_{Fi}^{+} = \sigma(X_{Xi} + X_{IN} - 1),\ i =1,\ldots,n$
$S_{Fin}^{+} = \sigma(X_{IN})$
$S_{H}^{+} = \sigma(RES_F - 10\ S_{resG} + S_{resG1} - X_{OUT})$
$S_{outF}^{+} = \sigma(OUT_F + S_{K2})$
$S_{sgIN}^{+} = \sigma(S_{outF})$
$S_{sgX}^{+} = \sigma(X_Y - 0.1)$
$S_{sg>0}^{+} = \sigma(2\ S_{sgIN} + 100\ S_{sgX} - 2)$
$S_{sg=0}^{+} = \sigma(2\ S_{sgIN} - 100\ S_{sgX} - 1)$
$S_{y1}^{+} = \sigma(0.2\ S_{outF} - 10\ X_Y - 1.2)$
$S_{K}^{+} = \sigma(0.1\ X_{IN} + S_K + S_{K1} - S_{K2} - X_{OUT})$

$S_{K1}^{+} = \sigma(0.1\ S_K + S_{K2} - 0.9)$
$S_{K2}^{+} = \sigma(S_{outG})$
$S_{Gi}^{+} = \sigma(X_{Xi} + S_{sg>0} - 1),\ i =1,\ldots,n$
$S_{Gy}^{+} = \sigma(S_K + S_{sg>0} - 1)$
$S_{Gh}^{+} = \sigma(S_H + S_{sg>0} - 1)$
$S_{Gin}^{+} = \sigma(S_{sg>0})$
$S_{resG}^{+} = \sigma(RES_G)$
$S_{resG1}^{+} = \sigma(S_{resG})$
$S_{outG}^{+} = \sigma(OUT_G)$
$X_{RES}^{+} = \sigma(S_H + S_{sg=0} - 1)$
$X_{OUT}^{+} = \sigma(S_{sg=0})$
$RES^{+} = \sigma(X_{RES})$
$OUT^{+} = \sigma(X_{OUT})$

A rede $R_{R(F,G)}$ só termina no instante t_{fim} se o neurónio $S_{SG=0}$ tiver como activação o valor 1 em t_{fim}-1, dado ser a única ligação para o neurónio x_{OUT}. $S_{SG=0}$ é o resultado do facto do neurónio x_y possuir o valor 0 em t_{fim}-3.

Do mesmo modo que na composição, consideramos que os módulos F e G convergem para os dados de entrada. Se isso não acontecer, o executar o módulo que diverge implica directamente a divergência da rede da recursão, o que corresponde à esperada indefinição da função associada.

A rede vai executar o algoritmo 3.3.21, ou seja, vai incrementar um contador que é guardado no neurónio S_K ao mesmo tempo que decrementa o valor inicial de y guardado em x_y, até que este atinja o valor zero. S_K é utilizado para

calcular o próximo valor da função associada do módulo G. Quando x_y atingir o valor zero, o módulo G foi chamado y vezes, obtendo-se assim, o resultado pretendido.

Quando é iniciada a rede, o módulo F calcula a sua função associada $f(x_1,\ldots,x_n)$ que é guardada no neurónio S_H. Ao mesmo tempo, por S_{outF} é executada a rede do sinal para avaliar se y já é zero. Se for, o neurónio $S_{sg=0}$ tem como activação 1, e a rede termina, passando S_H para x_{RES}. O que é correcto, pois para y=0 o valor da recursão é $f(x_1,\ldots,x_n)$.

Se isso não acontecer, a rede do sinal activa o neurónio $S_{sg>0}$ que vai activar os neurónios S_{Gxx} que permitem a execução do módulo G com parâmetros $x_1,\ldots,x_n,0,f(x_1,\ldots,x_n)$. Ao mesmo tempo, o valor de y é decrementado em uma unidade pela aplicação dos neurónios x_y e S_{y1}.

Quando o módulo G termina, os neurónios S_{resG} e S_{resG1} substituem o valor de S_H pelo valor da função associada de G. De seguida, o valor guardado em S_K é incrementado numa unidade pelos neurónios S_{K1} e S_{K2}. Ao mesmo tempo, S_{K2} envia o valor um para activar a rede sinal. O ciclo fecha-se. O valor de y é novamente testado e o módulo G é iterado o número de vezes necessárias para calcular o valor final da recursão.

Assim, a rede termina quando y=0, e para cada decremento de y, existe uma chamada do módulo G.

Se y=0, o resultado da rede é o valor da função $f(x_1,\ldots,x_n)$, i.e., $[R_{R(F,G)}](x_1,\ldots,x_n,0) = \alpha^{-1}([R_{R(F,G)}](0,1,<\alpha(x_1),\ldots,\alpha(x_n), \alpha(0)>) = \alpha^{-1}(\alpha(f(x_1,\ldots,x_n))) = f(x_1,\ldots,x_n)$.

Se y>0, o resultado da rede é o valor da função associada ao módulo G, com parâmetros $x_1,\ldots,x_n,y-1,h$, sendo h o resultado da última execução do módulo F (se y=1) ou do módulo G (se y>1), i.e.,

$[R_{R(F,G)}](x_1,\ldots,x_n,y+1) = \alpha^{-1}([R_{R(F,G)}](0,1,<\alpha(x_1),\ldots,\alpha(x_n),\alpha(y+1)>) = \alpha^{-1}(\alpha([R_G](0,1,<\alpha(x_1),\ldots,\alpha(x_n),\alpha(y)>,[R_{R(F,G)}](0,1,<\alpha(x_1),\ldots,\alpha(x_n),\alpha(y)>)))) = [R_G](0,1,<\alpha(x_1),\ldots,\alpha(x_n),\alpha(y)>,[R_{R(F,G)}](0,1,<\alpha(x_1),\ldots,\alpha(x_n),\alpha(y)>))$.

Verifica-se assim, que para ℓ = R(F,G), $[R_\ell] = \mathcal{F}[\![\ell]\!]$.

- Minimização

Seja a descrição F da função f utilizada na minimização. Para ℓ = M(F), a função $\mathcal{F}[\![\ell]\!](x_1,\ldots,x_n,y) = \mathcal{F}[\![M(F)]\!](x_1,\ldots,x_n,y)$ é definida por:

$\mathcal{F}[\![M(F)]\!](x_1,\ldots,x_n) = \min\{ y\in\mathbb{N} : \mathcal{F}[\![F]\!](x_1,\ldots,x_n,y) = 0 \wedge \mathcal{F}[\![F]\!](x_1,\ldots,x_n,z)$ está definido, $z<y\}$, se existir, indefinido, caso contrário.

Por definição, $[R_\ell](x_1,\ldots,x_n,y) = [R_{M(F)}](x_1,\ldots,x_n,y) = \alpha^{-1}([R_{M(F)}](0,1,<\alpha(x_1),\ldots,\alpha(x_n),\alpha(y)>))$

A descrição da rede $R_{M(F)}$:

$X_{IN}^{+} = \sigma(IN)$
$X_{Xi}^{+} = \sigma(X_i + X_{Xi} - X_{OUT})$, $i = 1,\ldots,n$
$S_{IN}^{+} = \sigma(X_{IN} + 10\, S_{Y1})$
$S_{Fi}^{+} = \sigma(X_{Xi} + S_{IN} - 1)$, $i =1,\ldots,n$
$S_{Fy}^{+} = \sigma(S_Y + S_{IN} - 1)$
$S_{Fin}^{+} = \sigma(S_{IN})$
$S_{sgIN}^{+} = \sigma(OUT_F)$
$S_{sgX}^{+} = \sigma(RES_F - 0.1)$
$S_{sg>0}^{+} = \sigma(2\, S_{sgIN} + 100\, S_{sgX} - 2)$
$S_{sg=0}^{+} = \sigma(2\, S_{sgIN} - 100\, S_{sgX} - 1)$
$S_Y^{+} = \sigma(0.1X_{IN} + S_Y + S_{Y1} - S_{Y2} - X_{OUT})$
$S_{Y1}^{+} = \sigma(0.1X_{IN} + 0.1\, S_Y + S_{Y2} - 0.9)$
$S_{Y2}^{+} = \sigma(S_{sg>0})$
$X_{RES}^{+} = \sigma(X_Y + S_{sg=0} - 1)$
$X_{OUT}^{+} = \sigma(S_{sg=0})$
$RES^{+} = \sigma(X_{RES})$
$OUT^{+} = \sigma(X_{OUT})$

A rede só termina, quando a rede sinal activa o neurónio $S_{sG=0}$. Dado que o valor de entrada da rede sinal é o valor da função associada ao módulo F, isto implica que a rede termina quando a função devolve zero.

A rede $R_{M(F)}$ está construída para executar o algoritmo 3.3.23. O neurónio S_Y vai ser iniciado, no instante t=2, com o valor 0.1 (ou seja, com o código de zero). O módulo F é executado com parâmetros $x_1,\ldots,x_n,0$, e ao terminar, envia

o seu valor para a rede sinal. Se for zero, a rede termina, devolvendo o valor de S_Y. Se não for igual a zero, o valor S_Y é incrementado numa unidade pelos neurónios S_{Y1} e S_{Y2}. Há a garantia que, se existir um y, tal que a função associada a F seja zero, a rede devolve esse y, já que a rede executa um ciclo de teste crescente e começando no zero. Se não existir nenhum y que faça que a função descrita por F se anule, então a rede $R_{M(F)}$ diverge, i.e.,

Se y existe: $[R_{M(F)}](x_1,\ldots,x_n) = \alpha^{-1}([R_{M(F)}](0,1,<\alpha(x_1),\ldots,\alpha(x_n)>) = \alpha^{-1}(\alpha(y)) = y$. Caso contrário, a função não está definida, dado que $[R_{M(F)}]$ diverge.

Verifica-se assim, que para ℓ = M(F), $[R_\ell] = \mathcal{F}[\![\ell]\!]$. ◁

3.5. Universalidade das Redes σ

Apresentámos um método de transformar uma descrição de $\mathcal{R}$ em uma rede-σ. Podemos assim, apresentar o principal teorema deste capítulo.

Teorema 3.5.1: O modelo de redes-σ é universal

Demonstração: Para provar a universalidade das redes-σ, mostramos como se poderia produzir uma rede capaz de computar qualquer descrição de $\mathcal{R}$.

Seja a descrição $\mathcal{H}$ apresentada na secção 3.2. Se aplicarmos o processo de construção das redes apresentado na secção 3.3 a $\mathcal{H}$ obtemos uma rede $R_{\mathcal{H}}$. Seja x a codificação de uma descrição $\ell \in \mathcal{R}$, ou seja, da função parcial recursiva $\mathcal{F}[\![\ell]\!]$. Seja y a codificação dos argumentos $x_1,\ldots,x_n$, de $\mathcal{F}[\![\ell]\!]$.

A função computada por $R_{\mathcal{H}}$ é a função parcial recursiva $\mathcal{F}[\![\mathcal{H}]\!]$ tal que, $[R_{\mathcal{H}}](x, y) = \mathcal{F}[\![\mathcal{H}]\!](x,y) = \mathcal{F}[\![\ell]\!](x_1,\ldots,x_n)$.

$R_{\mathcal{H}}$ é capaz de calcular o resultado de qualquer função parcial recursiva com qualquer argumento válido, sendo assim uma rede-σ universal. ◁

Este resultado tem importância para o estudo da Neurodinâmica, pois com ele é estabelecida uma conexão entre a Teoria da Computação Clássica e as redes

neuronais. É possível herdar todas as propriedades dos sistemas universais para o contexto das redes σ, i.e., para o ramo dos sistemas dinâmicos não lineares. Ainda que este resultado não seja inédito (a demonstração foi inicialmente feita por Siegelmann e Sontag no inicio dos anos 90, e pode ser encontrada na sua forma mais conhecida em [Siegelmann e Sontag 95]), já a forma como a demonstração é realizada introduz algo de novo e muito relevante: um método operacional de construção efectiva de redes neuronais. E este método, não só revela como construir uma função universal, mas funciona para qualquer descrição de função parcial recursiva.

Outras contribuições podem ser encontradas nos problemas de indecidibilidade[12], ou ainda a constatação da existência de uma rede de dimensão máxima onde é possível resolver qualquer problema computável (no nosso caso, a rede $R_{\mathcal{H}}$).

3.6. Extensões do Modelo

Outros Domínios

Até este ponto, todas as funções descritas têm domínio e contradomínio nos naturais. Como resolver a questão para outros domínios, como $\mathbb{Z}$ ou $\mathbb{Q}$? A estratégia de resolução de problemas para um domínio D qualquer, exige que a cardinalidade de D seja menor ou igual à dos naturais (o que é o caso para $\mathbb{Z}$ e $\mathbb{Q}$). Deste modo, poderemos resolver o problema por meios de uma codificação $\alpha: D \rightarrow \mathbb{N}$ conveniente.

Uma função $f : D \rightarrow D$, pode ser emulada pela função $f^* : \mathbb{N} \rightarrow \mathbb{N}$, que aplica o objecto codificado $\alpha(d)$ no resultado codificado $\alpha(f(d))$, i.e., $f^* = \alpha \circ f \circ \alpha^{-1}$

[12] Uma outra consequência, apontada por Cristopher Moore em [Moore 90, 91], é que ao introduzir as noções da Teoria da Computação nos sistemas dinâmicos, surge uma imprevisibilidade qualitativamente mais forte do que aquela discutida pelo estudo dos sistemas caóticos. Agora, mesmo se se souber os dados iniciais do sistema com total exactidão, haverá questões para as quais não é possível saber a resposta (e.g., entrará o sistema numa determinada região do espaço de fases?). Quase qualquer pergunta – para ser exacto, qualquer pergunta não trivial – sobre a dinâmica a longo termo de um sistema torna-se indecidível!

(ver [Cutland 80]). Podemos assim, estender a noção de computabilidade para domínios D.

Definição 3.6.1: Uma função f:D→D é *computável* se e só se $f^* = \alpha \circ f \circ \alpha^{-1}$ for uma função computável (sendo α uma codificação de D em $\mathbb{N}$). ◁

Exemplo 3.6.2: Uma codificação α de $\mathbb{Z}$ em $\mathbb{N}$ é dada por:

$$\alpha(x) = \begin{cases} 2x & , x \geq 0 \\ -2x-1 & , x < 0 \end{cases} \qquad (16)$$

Assim, $\alpha(3) = 6$, $\alpha(-2) = 3$, …

Exemplo 3.6.3: Uma codificação α de $\mathbb{Q}$ em $\mathbb{N}$ é dada por:

$$\alpha(x/y) = 2 \left[\frac{(|x|+y)(|x|+y+1)}{2} + y \right] + r \qquad (17)$$

com x o numerador (herdando o eventual sinal do racional), y o denominador e r = 1 se x<0, ou 0 se x≥0. Assim, $\alpha(1/2) = 16$, $\alpha(-2/30) = 1117$, …

A extensão desta ideia para funções com n argumentos é directa. Codifica-se cada um dos argumentos antes de se computar a função f^* (como exemplo, ver a definição 3.2.3).

Tratamento de Ruído

Esta generalização é útil para lidar com problemas onde exista ruído na computação. Se soubéssemos que os neurónios podiam ser sujeitos a erros de cálculo a partir das P casas decimais, qualquer codificação dos naturais será limitada a um determinado valor máximo. É possível aumentar esse valor máximo utilizando vários neurónios para guardar o número em questão. A codificação dos naturais até ao majorante 10^M seria dada pela função de codificação, $\alpha(x) = <\beta_1(x), \ldots, \beta_n(x)>$, onde $n = \lceil \frac{M}{P-2} \rceil$, $\beta_i(x) = d((P-2)(i-1)+1, x)*10^0 + \ldots + d((P-2)i, x)*10^{(P-3)}$, sendo d(m, x) o digito na m-ésima casa da parte inteira da expansão decimal de x, sendo o erro de cálculo a partir das P casas decimais. A ideia básica é limitar a execução dos cálculos da rede nas

casas decimais onde é garantido que não ocorre nenhum erro. Por exemplo, consideremos P=4 (um erro máximo na casa das décimas milésimas). Para guardar valores até 10^6 bastam 3 neurónios. O primeiro neurónio guarda o valor das unidades e das dezenas, o segundo, as centenas e os milhares, e assim sucessivamente.

Construindo as redes para calcular o sucessor, o predecessor e o teste da igualdade a zero, pode alterar-se a estrutura das redes apresentadas nas figuras 3.3.22 e 3.3.24, para suportar naturais representados por múltiplos neurónios.

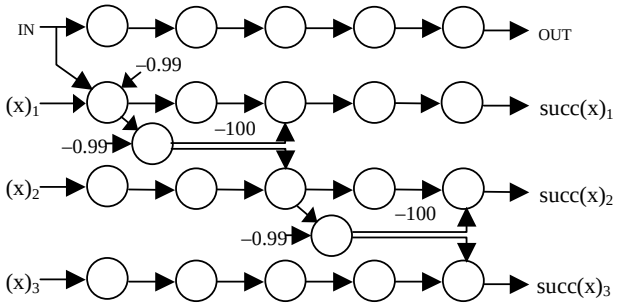


fig. 3.6.4 : sucessor para P=4 e M=6.

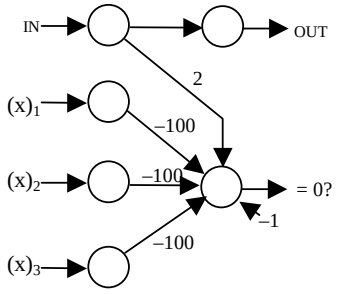


fig. 3.6.5 : teste ao valor zero para P=4 e M=6.

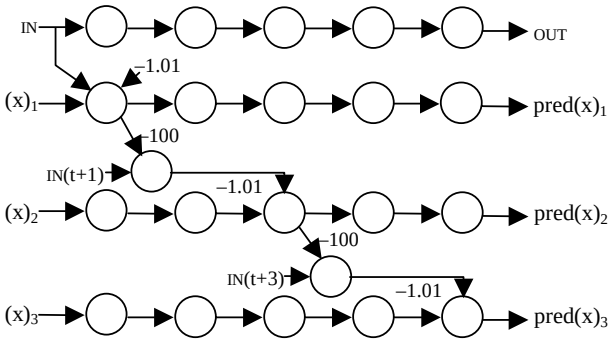


fig. 3.6.6 : predecessor para P=4 e M=6.

Deste modo, desde que a margem de erro seja inferior a uma centésima (P>2), qualquer função computável pode ser implementada em redes neuronais uma vez definido um majorante da computação. Este método apresenta uma grande vantagem relativamente ao número de neurónios necessários para realizar estas três operações: cresce logaritmicamente quanto maior o majorante, e decresce logaritmicamente quanto menor o erro (ou seja, quanto maior o P). Isto acontece dado que a dimensão das três redes ser função de $n = \lceil M/(P-2) \rceil$, onde M e P são, respectivamente, os expoentes do majorante (10^M) e do erro (10^{-P}).

Resta resolver a questão de como evitar o acumular dos pequenos erros de cada neurónio ao longo da execução da rede. A solução é tratar o eventual erro do neurónio na função de activação. Ao existir apenas um espectro de valores relevantes (os números racionais com precisão máxima de P-2 casas decimais) pode alterar-se a função para aproximar o valor produzido pelo neurónio ao valor correcto mais próximo.

$$f(x) = \begin{cases} 0 & , x<0 \\ \alpha(n) & , n = \text{ROUND}(x,P-2),\ 0\leq x\leq 1 \\ 1 & , x>1 \end{cases} \qquad (18)$$

onde ROUND(x, P) é a função que devolve o racional com P casas decimais mais perto de x.

Tem-se assim, com uma função de activação mais complexa e com um aumento linear do número de neurónios da rede e do tempo de processamento, a possibilidade de executar computação exacta num sistema dinâmico com precisão limitada.

Os neurónios apresentados têm uma garantia de precisão de P-1 casas decimais. Seriam possíveis computações com neurónios onde nem essa garantia existisse? Sejam dois exemplos, designados por neurónios de esquecimento e de persistência probabilística, cujas funções de activação são dadas pelas seguintes funções:

$$f_{e1}(x) = \begin{cases} 0 & \text{, com probabilidade p} \\ \text{valor normal} & \text{, com probabilidade 1-p} \end{cases} \quad (19)$$

activação com esquecimento

$$f_{e2}(x) = \begin{cases} \text{valor anterior} & \text{, com probabilidade p} \\ \text{valor normal} & \text{, com probabilidade 1-p} \end{cases} \quad (20)$$

activação com persistência

Em [Siegelmann 99] é observado que redes neuronais onde todos os neurónios possuem uma probabilidade de erro de cálculo (designadas por redes catastróficas) não podem reconhecer linguagens recursivas, i.e., não possuem o poder computacional das Máquinas de Turing. É sempre necessário existir uma de duas condições, ou a rede é composta por neurónios com precisão finita (o caso apresentado anteriormente), ou a rede é mista, sendo composta por neurónios não confiáveis, mas também com neurónios deterministas (para permitir corrigir os erros, e impedir a sua acumulação descontrolada). Com redes catastróficas, as únicas computações possíveis são aquelas que se podem efectuar em tempo constante (onde se conhece o majorante do erro).

Outros modelos de computação neuronal em tempo discreto com ruído (cf. [Casey 96, 98], [Maass e Orponen 98]), provam que qualquer linguagem reconhecida com fiabilidade (i.e., com probabilidade maior que 50% [13]) por redes neuronais com função de activação sigmoidal é regular, ou seja, o seu poder computacional é equivalente ao dos autómatos finitos.

Porém, em [Maass e Sontag 99] é demonstrado que se o nível de ruído for Gaussiano, ou qualquer outro modelo de ruído em que a distribuição seja não nula num intervalo suficientemente grande, o poder computacional das redes neuronais recorrentes reduz-se significativamente, não conseguindo sequer reconhecer o conjunto das linguagens regulares.

[13] Fiabilidade, devido ao facto de se poder iterar várias vezes a mesma computação, de modo a obter validações com arbitrário nível de segurança.

3.7. Compilador de Funções Parciais Recursivas

Foi desenvolvido um trabalho de pós-graduação pelos alunos Paulo Carreira e Miguel Rosa (cf. [Carreira *et al.* 98]), sendo orientado pelo autor desta dissertação, o qual consistiu na construção de uma aplicação capaz de compilar descrições de funções parciais recursivas nas redes propostas neste capítulo. Este programa, desenvolvido no ambiente de programação para o sistema operativo Windows – Delphi 3 – possui as ferramentas necessárias para editar, compilar, monitorar passo a passo, e executar uma determinada função parcial recursiva. Uma das suas características (que reflecte a abordagem tomada neste capítulo no que respeita à codificação dos naturais) é o uso de um sistema de representação unário sem limite, apenas restrito à memória disponível no computador. Deste modo, a aplicação pode aceitar números naturais arbitrariamente elevados desde que a sua memória assim o permita.

A construção da rede passa pela decomposição recursiva das descrições até atingir um dos axiomas e a sua posterior reconstrução para obter as redes desejadas. Por exemplo, para criar a rede neuronal capaz de computar a adição binária, o compilador interpreta a função descrita da seguinte forma:

```
proj/1 U(1,1)
proj/2 U(3,3)
comp   C( proj/2, S )
sum    R( proj/1, comp )
```

Em que a primeira palavra é o nome da função representada nessa linha e a segunda a descrição da função parcial recursiva correspondente. Neste exemplo, a função objectivo, *sum* (identificada como sendo a última linha da descrição) é obtida através da recursão da função *proj/1* (sendo o axioma $U_{1,1}$) e da função *comp*, a qual por sua vez, é uma composição da função *proj/2* (o axioma $U_{3,3}$) com o axioma S. Ao compilar, o programa produz a rede final descrita pelo seguinte conjunto de equações:

```
XRin_1_0(i+1) = sigma( 1.0*Ein(i) + 0 )
XRmid2_1_1(i+1) = sigma( 1.0*XRin_1_0(i) + 0 )
XRmid3_1_2(i+1) = sigma( 0.10*XRin_1_0(i) + 1.0*XRmid3_1_2(i) +
   1.0*XRmid4_1_3(i) + -1.0*XRmid5_1_4(i) + -1.0*XREout_1_15(i) + 0 )
XRmid4_1_3(i+1) = sigma( 0.10*XRmid3_1_2(i) + 1.0*XRmid5_1_4(i) + -0.9 )
XRmid5_1_4(i+1) = sigma( 1.0*XRmid6_1_5(i) + 0 )
XRmid6_1_5(i+1) = sigma( 1.0*XSout_5_7(i) + 0 )
XRmid7_1_6(i+1) = sigma( 1.0*XRmid3_1_2(i) + 1.0*XRmid15_1_14(i) + -1 )
```

```
XREres_1_7(i+1) = sigma( 1.0*XRmid13_1_12(i) + 1.0*XRmid19_1_18(i) + -1 )
XRmid9_1_8(i+1) = sigma( 1.0*XRmid5_1_4(i) + 1.0*XU(1,1)inout_3_0(i) + 0 )
XRmid10_1_9(i+1) = sigma( 0.20*XRmid9_1_8(i) + 10.0*XRdatay_1_10(i) + -1.2 )
XRdatay_1_10(i+1) = sigma( -1.0*XRmid9_1_8(i) + 1.0*XRmid10_1_9(i) +
   1.0*XRdatay_1_10(i) + -1.0*XREout_1_15(i) + 1.0*Edata2(i) + 0 )
XRmid12_1_11(i+1) = sigma( 1.0*XRmid9_1_8(i) + 0 )
XRmid13_1_12(i+1) = sigma( 2.0*XRmid12_1_11(i) + -100.0*XRmid14_1_13(i) + -1 )
XRmid14_1_13(i+1) = sigma( 1.0*XRdatay_1_10(i) + -0.1 )
XRmid15_1_14(i+1) = sigma( 2.0*XRmid12_1_11(i) + 100.0*XRmid14_1_13(i) + -2 )
XREout_1_15(i+1) = sigma( 1.0*XRmid13_1_12(i) + 0 )
XRmid17_1_16(i+1) = sigma( 1.0*XRmid15_1_14(i) + 0 )
XRmid18_1_17(i+1) = sigma( 1.0*XRmid15_1_14(i) + 1.0*XRmid19_1_18(i) + -1 )
XRmid19_1_18(i+1) = sigma( -1.0*XREout_1_15(i) + 1.0*XRmid19_1_18(i) + -
   10.0*XRmid20_1_19(i) + 1.0*XRmid21_1_20(i) + 1.0*XU(1,1)datares_3_1(i) + 0
   )
XRmid20_1_19(i+1) = sigma( 1.0*XSres_5_9(i) + 0 )
XRmid21_1_20(i+1) = sigma( 1.0*XRmid20_1_19(i) + 0 )
XRdata1_1_21(i+1) = sigma( 1.0*XRdata1_1_21(i) + -1.0*XREout_1_15(i) +
   1.0*Edata1(i) + 0 )
XRmid22_1_22(i+1) = sigma( 1.0*XRdata1_1_21(i) + 1.0*XRin_1_0(i) + -1 )
XRmid23_1_23(i+1) = sigma( 1.0*XRdata1_1_21(i) + 1.0*XRmid15_1_14(i) + -1 )
XU(1,1)inout_3_0(i+1) = sigma( 1.0*XRmid2_1_1(i) + 0 )
XU(1,1)datares_3_1(i+1) = sigma( 1.0*XRmid22_1_22(i) + 0 )
XU(1,1)data_3_2(i+1) = sigma( 0 )
XCmid1_3_3(i+1) = sigma( -1.0*XCmid1_3_3(i) + 1.0*XCmid8_3_7(i) + 0 )
XCmid2_3_4(i+1) = sigma( 1.0*XCmid1_3_3(i) + 0 )
XCmid6_3_5(i+1) = sigma( -1.0*XCmid7_3_6(i) + 1.0*XCmid6_3_5(i) +
   1.0*XU(3,3)datares_5_4(i) + 0 )
XCmid7_3_6(i+1) = sigma( 1.0*XCmid6_3_5(i) + 1.0*XCmid1_3_3(i) + -1 )
XCmid8_3_7(i+1) = sigma( -1.0*XCmid1_3_3(i) + 1.0*XCmid8_3_7(i) +
   1.0*XU(3,3)inout_5_3(i) + 0 )
XU(3,3)inout_5_3(i+1) = sigma( 1.0*XRmid17_1_16(i) + 0 )
XU(3,3)datares_5_4(i+1) = sigma( 1.0*XRmid18_1_17(i) + 0 )
XU(3,3)data_5_5(i+1) = sigma( 1.0*XRmid23_1_23(i) + 1.0*XRmid7_1_6(i) + 0 )
XSin_5_6(i+1) = sigma( 1.0*XCmid2_3_4(i) + 0 )
XSout_5_7(i+1) = sigma( 1.0*XSin_5_6(i) + 0 )
XSdata_5_8(i+1) = sigma( 1.0*XCmid7_3_6(i) + 0 )
XSres_5_9(i+1) = sigma( 1.0*XSin_5_6(i) + 0.10*XSdata_5_8(i) + -0.9 )
```

De reparar que todos os neurónios possuem a função de activação σ. Também os módulos ligam-se de forma automática uns nos outros através do sistema de sincronização proposto.

Como qualquer função parcial recursiva pode ser descrita usando a notação apresentada, esta revela-se apropriada para o estudo da computabilidade deste tipo de construções neuronais.

Este programa, mais um conjunto de informação complementar, podem ser encontrados na página *www.di.fc.ul.pt/~jpn/netdef/nwb.html*.

4. Soma e Controle

Para efectuar operações de controle, uma rede neuronal tem de possuir uma estrutura específica para resolver um dado problema. Ao ligar os neurónios um por um, essa tarefa assemelha-se à programação em linguagem máquina, com a desvantagem de se estar a trabalhar num ambiente maciçamente paralelo. Rapidamente, à medida que o problema proposto aumenta de complexidade, verifica-se que este método directo é de gestão muito complexa.

É mostrado neste capítulo como eliminar essa necessidade, ao criar um método automático de compilação de algoritmos escritos numa linguagem de alto nível, para uma rede neuronal que executa esse mesmo algoritmo. O processo é esquematizado na seguinte figura:

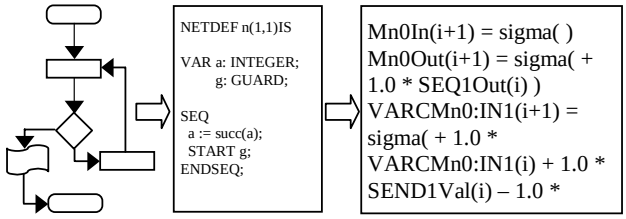


fig. 4.0.1 : (a) fluxograma do algoritmo **A**, (b) descrição de **A** numa linguagem apropriada, (c) Rede neuronal que executa **A**.

Haverá uma preocupação constante com a eficácia da rede final, tanto no aspecto da sua construção (construção modular simples e minimizando o número de neurónios necessários), bem como na sua eficácia (preservar a complexidade intrínseca do algoritmo inicial). O modelo de rede utilizado neste capítulo será idêntico ao apresentado no capítulo anterior, não só na arquitectura e na função de activação, como também na preocupação relativa à modularização e aos mecanismos de sincronização apresentados. A diferença reside na linguagem utilizada para a construção das redes-σ finais. Não usaremos mais o paradigma das funções parciais recursivas, mas sim a estrutura básica das linguagens de programação de alto nível.

É necessário referir que esta abordagem é diferente das linguagens de especificação de redes neuronais, onde o objectivo é o desenho da rede, ou seja, definir formalmente a topologia desta juntamente com a sua dinâmica. Elas fazem a separação entre especificação (a arquitectura) e implementação (o programa). Alguns exemplos podem ser encontrados no formalismo MIND (cf. [Koikkalainen 91]) uma extensão do CSP ([Hoare 78]), a linguagem P3 (cf. [Rumelhart e McClelland 86]) baseada no LISP, a linguagem AXON, [Hecht-Nielsen 90], linguagem imperativa centrada em objectos.

4.1. Trabalhos relacionados

Há dois trabalhos seminais sobre o assunto desta Tese. Referiremos aqui os seus principais pontos.

JaNNeT

Em 1995, Fréderic Gruau do Departamento de Investigação Fundamental do Centro de Estudos Nucleares de Grenoble, apresentou em [Gruau 95] um compilador neuronal denominado JaNNeT (*Just a Neural Network Translator*). Ele próprio indica que não procurou uma linguagem específica para o caso, utilizando o PASCAL, ao qual acrescentou certas instruções que permitem programação paralela.

Uma das preocupações principais foi a possibilidade de apresentar o desenho da rede neuronal final, depois de terminado o processo de compilação. Para isso, utilizou técnicas de rescrita de grafos que indicassem a posição de cada neurónio para gerir e minimizar o número de sinapses que se intersectavam (um grafo inteiramente planar é impossível, dadas as dimensões das redes produzidas).

O programa, respeitando a gramática da linguagem, é decomposto no formato de árvore. Em seguida, essa árvore é transformada num código específico, que Gruau denomina por código celular. Este código é constituído por um conjunto extenso de comandos primitivos muito simples, que por sua vez são utilizados para construir a rede neuronal final. A estrutura da rede neuronal é modular na medida em que a execução de uma dada instrução está restrita a uma certa área da rede global. A dinâmica desta rede resulta de uma sobreposição de comportamentos sequencial e paralelo, dado que cada neurónio só é activado quando todas as suas sinapses de entrada o são. Ou seja, em cada instante, mais do que um neurónio é activado, mas, possivelmente, há neurónios que permanecem inactivos. Gruau ainda refere a necessidade de outras duas dinâmicas diferentes para certos neurónios, referindo que a uniformização para uma só dinâmica se pode fazer à custa de um aumento polinomial do número de neurónios. No entanto, não apresenta uma concretização para tal proposta.

Há quatro tipos de neurónios para realizar as tarefas básicas, mais alguns para funcionalidades específicas (como o tratamento de vectores). Isto implica que a arquitectura neuronal final não é homogénea.

O JaNNeT também processa funções recursivas, mas com uma limitação. É necessário indicar a profundidade máxima da recursão. Tendo essa informação, o compilador cria uma rede expandida para tratar todas as potenciais chamadas! Alguns comentários:

- A necessidade de apresentação gráfica de uma rede final é muito relativa, a não ser que se esteja a pensar seriamente em construir fisicamente uma rede de neurónios. Sem querer entrar em profundidade nas questões de

hardware, isto pode trazer sérios problemas de construção, dado que os grafos produzidos estão muito longe de serem planares, tendo múltiplos pontos de intersecção entre sinapses. Existem formas mais simples e directas de implementar uma rede neuronal, como se verá no capítulo 5.9.

- Apesar de ser um problema complexo, dado que o número de neurónios é fixo depois do processo de compilação terminar, Gruau ao exigir um número máximo de chamadas recursivas da parte do programador, está a impedir todo o potencial que o uso da recursão reserva. Outro problema, ao produzir a rede final para tratar essas chamadas pode ser necessário um número exponencial de neurónios. Mas mesmo com estes problemas, continua a ser um dos pontos fortes do JaNNeT o facto de se poder usar funções recursivas na sua programação!
- A quantidade de neurónios diferentes e a dinâmica da rede produzem um processamento muito pesado (é necessário guardar para cada neurónio, quantas e quais as activações pendentes) e uma rede cujo processamento não é verdadeiramente paralelo (somente alguns neurónios são activados a cada instante).

NEL

No decorrer da investigação de Hava Siegelmann sobre o poder computacional das redes neuronais, surgiu o estudo de uma linguagem de alto nível para a criação de redes neuronais capazes de realizar certas tarefas algorítmicas (ver [Siegelmann 93, 96]). Em 1993, Hava Siegelmann apresentou uma linguagem, denominada NEL, com o poder expressivo das linguagens de alto nível.

Ao contrário de Gruau, que se preocupou no processo de construção das redes em detrimento do estudo da linguagem, Siegelmann fez o inverso. Estudou qual seria a linguagem indicada para um dado tipo de redes, não se preocupando inicialmente com aspectos mais práticos, como a construção de um compilador.

Cada instrução produz uma rede, que é sincronizada externamente por um sistema de sincronização global. A dinâmica da rede é única e verdadeiramente

paralela. Um passo da rede é simplesmente um sistema de atribuições paralelas. A arquitectura é homogénea, todos os neurónios são idênticos e utilizam a função de activação σ. Alguns comentários:

- Ao focar inteiramente o esforço de desenvolvimento na adequação da linguagem às funcionalidades dos neurónios com função de activação σ, há certos pontos no NEL que tornam complexa a sua simulação. Por exemplo, certos tipos de dados complexos apresentados necessitam de uma precisão consumidora de bastantes recursos – de facto, à partida, a precisão é ilimitada. Isto acontece porque apenas é utilizado um neurónio para guardar estruturas arbitrariamente complexas, como conjuntos e filas de espera. Para isso, Siegelmann utiliza codificações fractais baseadas num conjunto de Cantor. Por exemplo, uma palavra $L = a_1 a_2 \ldots a_{|L|}$ é codificada pelo seguinte número racional (onde *n* é a cardinalidade do conjunto definido pelos símbolos a_i):

$$\sum_{i=1}^{|L|} \frac{2a_i + 1}{(2n)^i} \qquad (21)$$

  Apesar de ser extraordinariamente compacta (qualquer palavra pode ser guardada num só racional), o grande problema desta codificação é de não se conhecer a precisão máxima, dado que as estruturas de dados podem crescer sem limite.

- A homogeneidade da rede e o tratamento das instruções, mais simples do que no trabalho de Gruau, aproxima-se da nossa própria abordagem. No entanto, o aspecto da modularidade não é tratado de forma conveniente, dado que o sistema de sincronização é global. Cada rede que representa uma função está dependente desse sistema de sincronização, que é específico para a rede final. Ou seja, ao mudar uma instrução local, a sincronização tem de ser alterada de forma global.

- O NEL não providencia muitos mecanismos essenciais para uma linguagem neuronal de alto nível. Apesar da capacidade de expressão de qualquer linguagem sequencial, tem sérias restrições em relação a questões de processamento paralelo: não suporta exclusão mútua para acesso seguro a

áreas de memória crítica; não possui processos temporais para aplicações em tempo real; não permite a chamada paralela de funções (apenas suportando uma limitada forma de subrotina); não tem processos de comunicação bloqueantes para iteração e sincronização concorrente de processos; não permite acesso dinâmico a elementos de um vector. Todos estes temas são tratados pelo NETDEF (como veremos neste capitulo) sem pôr em causa a modularidade.

4.2. A Linguagem NETDEF

A questão principal centrar-se-á em como simplificar a descrição do programa (de forma a ganhar em termos de eficiência, seja na programação, seja na execução), e ao mesmo tempo, conseguir a resolução do problema específico. Foram estudadas várias abordagens utilizadas pela comunidade científica. Aquela que mais se aproximou dos objectivos propostos foi o conceito defendido pelas linguagens imperativas de alto nível, com uma preocupação nos aspectos da concorrência (cf. [Watt 90], [Valliant 90], [Lamport 90], para várias abordagens). Isto é essencial dado ser uma das características centrais e mais atractivas das redes neuronais, o seu paralelismo intrínseco.

processo
canal

A linguagem utilizada neste capítulo (cf. [Neto *et al.* 98], [Neto e Costa 99] e [Neto *et al.* 2001a], para os artigos referentes a esta temática) é baseada num fragmento da linguagem de programação Occam, que designaremos de NETDEF (Network Definition). O Occam foi desenhado para exprimir algoritmos paralelos em redes de processadores (para mais informação, ver [SGS-THOMSOM 95]). Com esta linguagem, um programa pode ser descrito como uma colecção de processos executados de forma concorrente e comunicando entre si através de canais. Processos e canais são os conceitos mais importantes no paradigma de programação Occam.

Processos

Os programas são construídos por processos. O processo mais simples é a acção. Há três tipos de acções: atribuição do valor de uma expressão a uma

acção

variável, operações de entrada e operações de saída. Operação de entrada significa que um determinado valor foi recebido de um canal e atribuído a uma variável. Operação de saída significa que um dado valor de uma variável foi enviado através de um canal. Há ainda mais dois processos primitivos: *skip* e *stop*. O *skip* inicia, não realiza acção nenhuma e termina. O *stop* inicia, não realiza acção nenhuma e não termina.

skip
stop

Para construir processos mais complexos, existem várias instruções, nomeadamente, *while*, *if*, *seq*, *par*. A instrução *while* é uma instrução de ciclo, repetindo um dado processo enquanto uma expressão booleana associada é verdadeira. A instrução *if* combina vários processos, cada um guardado por uma expressão booleana. A instrução *seq* é um construtor que associa um conjunto de processos, onde cada um é executado sequencialmente. A instrução *par* é um construtor que associa um conjunto de processos, onde cada um é executado paralelamente.

while
seq
par
if

Canais

Um canal de comunicação permite a transferência de informação de forma unidirecional entre dois processos concorrentes, sem memória tampão. Um canal não possui memória e é bloqueante, ou seja, o processo receptor espera pelo envio do processo emissor (cf. [Pippenger 90] para um estudo de canais de redes de comunicação concorrentes). Veremos numa das seguintes secções, mais sobre estes canais de comunicação. Estes serão a peça central para a comunicação e coordenação entre módulos de execução paralelos e colaborantes.

Sintaxe

Segue uma gramática simplificada do NETDEF, em EBNF:

program ::= "NETDEF" id "IS" def-vars process ".".
process ::= assignment | send | receive | skip | stop | if-t-e | while-do | seq-block | par-block.

4.3. Tipos de Dados

NETDEF tem os seguintes tipos de dados: primitivos, canais e vectores. Os tipos primitivos são os tipos de valores que podem ser directamente utilizados na programação em NETDEF: booleanos, inteiros e reais. Os canais são tipos específicos para comunicação entre processos. Os vectores são estruturas de dados de um mesmo tipo de valor (inteiro, real ou booleano). Em termos do EBNF:

def-vars ::= "VAR" id ":" type ";" { id ":" type ";" }.
type ::= primitive-type | channel-type | composite-type.
primitive-type ::= "BOOLEAN" | "INTEGER" | "REAL" .
channel-type ::= "CHANNEL"
composite-type ::= "ARRAY" "[" number "]" "OF" primitive-type.

De referir ainda que o compilador de NETDEF não assinala conflitos de tipos. Isso implica que um valor de um tipo pode ser atribuído a uma variável de um outro tipo sem provocar um erro de compilação ou de execução (nada se poderá dizer sobre a correcção do resultado final). No entanto, no caso de uma atribuição, o compilador verifica se há conflito entre o número de neurónios referente à expressão e os neurónios da variável a ser atribuída o resultado dessa expressão.

Codificação

Para ser de alguma utilidade prática, a codificação dos valores possíveis para os vários tipos de dados deve ser realizada no contexto de recursos finitos (precisão finita nos cálculos do neurónio, número de neurónios finito). Somente assim, se poderá codificar informação no espaço limitado disponível pelos sistemas de computação efectivamente construídos.

De seguida, mostramos como facilitar a computação no processamento neuronal através de codificações adequadas aos vários tipos de dados estudados.

Tendo em consideração os limites de saturação mínimo e máximo da função de activação σ, cada valor x de um dado tipo de dados, deve ser codificado num valor dentro do intervalo [0,1]. Para cada tipo T, é necessário criar uma codificação α_T:T$\rightarrow$[0,1] que aplica o valor x$\in$T no seu código específico. E será esta codificação que determinará a arquitectura neuronal dos operadores relacionados. Sendo os recursos limitados, existe um limite para a precisão de cada valor nos cálculos do neurónio. Seja a precisão máxima de P dígitos numa codificação binária. A menor distância entre dois valores, nestas condições, é $1/M = 2^{-P}$, com $M=2^P$.

α_T

Majorante M

Para o tipo de dados Booleano, a codificação de B = {TRUE, FALSE} é definida por

$$\alpha_B(x) = \begin{cases} 0 & , x=\text{FALSE} \\ 1 & , x=\text{TRUE} \end{cases} \qquad (22)$$

Para o tipo de dados inteiro, a codificação de $Z = \{ -\frac{M}{2}, ..., \frac{M}{2} \}$ é definida por,

$$\alpha_Z(x) = \frac{M + 2x}{2M} \qquad (23)$$

Para o tipo de dados real, a codificação de [a, b], é definida por,

$$\alpha_{[a,b]}(x) = \frac{x-a}{b-a} \qquad (24)$$

Este tipo de codificação, para além da sua simplicidade (a qual se reflecte na simplicidade dos respectivos operadores, como se verá abaixo), tem a vantagem de que basta aumentar P para crescer a sua capacidade de representação. Por exemplo, para um sistema de 64 bits de representação (P=64), o intervalo disponível para representação dos inteiros, ou seja, o intervalo Z é igual a $\{-2^{63}, ..., 2^{63}\}$. Ao aumentar o valor de P, acresce automaticamente o intervalo Z. Ou seja, a capacidade de representação numérica é proporcional à memória disponível (do mesmo modo ao que se passa no paradigma das máquinas de Turing, ou na arquitectura clássica de construção dos computadores actuais).

Constantes, Variáveis e Expressões

Uma constante é um valor dado de um determinado tipo primitivo. Ela é guardada por um neurónio, cujo valor não é alterado.

constante

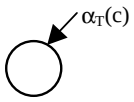


fig. 4.3.1 : constante c do tipo de dados T.

As variáveis são representadas de igual forma, mas o seu conteúdo pode ser alterado (por exemplo, no processo de atribuição).

variável

fig. 4.3.2 : variável A.

A definição de variáveis é dada pela seguinte expressão EBNF:

def-vars ::= "VAR" id ":" type ";" { id ":" type ";" }.

O NETDEF deve ser capaz de interpretar, guardar e manipular informação dos vários tipos de dados disponíveis. Com a introdução dos tipos de dados, são necessários diversos operadores que permitam efectuar os cálculos respectivos (operadores lógicos para variáveis lógicas, operadores relacionais e aritméticos para variáveis inteiras e reais). Uma estrutura arbitrária de operadores (conjuntamente com constantes, variáveis e dados de entrada) forma uma expressão que, para cada atribuição de valores às variáveis, devolve um valor de um determinado tipo. Cada rede que computa uma expressão é iniciada depois de recebido o sinal IN. Depois da computação, o resultado final é enviado pelo canal de saída de dados RES ao mesmo tempo que o sinal OUT (como sempre, sinapses sem número associado, possuem peso 1).

expressão

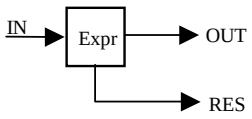


fig. 4.3.3 : configuração geral de uma rede representando uma expressão.

Para determinar a rede que calcula o valor de uma expressão f(x) para um tipo de dados T, é necessário criar a rede que calcula $\alpha_T(f(\alpha^{-1}{}_T(x)))$. Esta expressão justifica-se pelo seguinte motivo: Os valores que são recebidos pela rede vêm

codificados pela função α_T. Assim, antes de calcular a função f, é necessário descodificar os valores. Quando calculado o resultado, este tem de ser codificado novamente para poder ser enviado para a rede seguinte. Tudo isto, porque a área de trabalho disponível pela função de activação σ é dada no intervalo [0,1], intervalo este onde se codifica qualquer valor representável.

- Operadores Booleanos: estes são os típicos operadores booleanos de McCulloch – Pitts (cf. [McCulloch e Pitts 43])

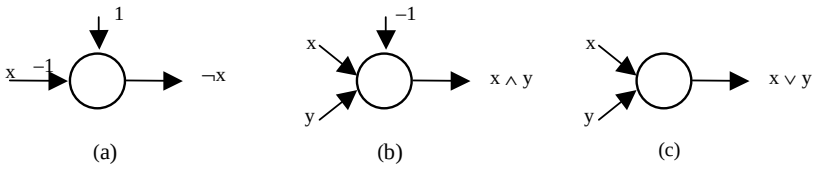


fig. 4.3.4 : operadores booleanos: (a) negação, (b) conjunção, (c) disjunção.

- Operadores Inteiros: Estes são os operadores aritméticos e relacionais para o tipo de dados inteiro. Para obter estas redes, foram calculados os valores das expressões respectivas usando a composição $\alpha_T \circ f \circ \alpha^{-1}{}_T$. Apresentamos em seguida, o exemplo da adição binária (notar que os valores x e y vêm codificados pela função α_Z):

$$\alpha_Z(x+y) = [M + 2(x+y)] / 2M = \alpha_Z(x) + \alpha_Z(y) - 0.5$$

Esta expressão conduz-nos à rede 4.3.5(b). As redes para as outras operações inteiras e reais são criadas de forma semelhante.

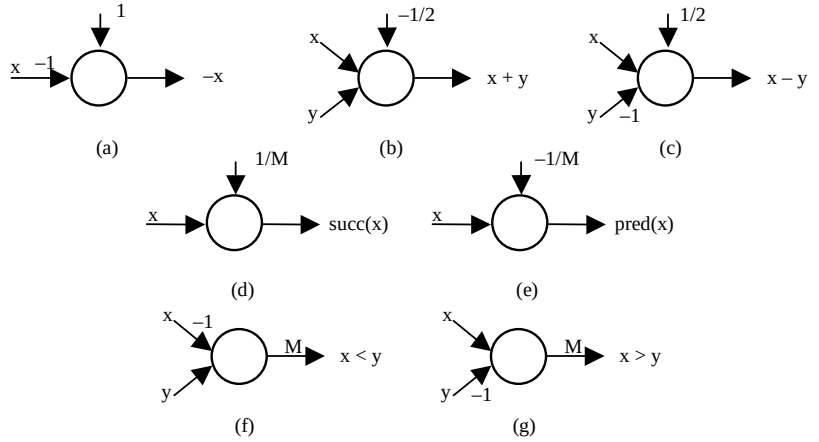


fig. 4.3.5 : operadores inteiros: (a) troca de sinal, (b) adição, (c) subtracção, (d) sucessor, (e) predecessor, (f) menor, (g) maior.

Para construir outros operadores relacionais basta compor os já existentes:

- x=y é definido por $\neg(x>y \vee x<y)$
- $x \neq y$ é definido por $x>y \vee x<y$
- $x \geq y$ é definido por $x>y \vee x=y$
- $x \leq y$ é definido por $x<y \vee x=y$

- Operadores Reais: estes são os operadores aritméticos para o tipo de dados real no intervalo [a, b] (os operadores relacionais são idênticos aos dos inteiros)

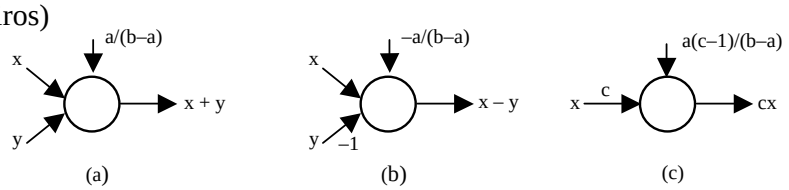


fig. 4.3.6 : operadores reais: (a) adição, (b) subtracção, (c) multiplicação por uma constante c.

Para além destes neurónios, é preciso ainda juntar o mecanismo de sincronização do sinal de entrada e de saída. Outra questão é que as expressões podem ser arbitrariamente complexas e alguns cálculos podem ter de esperar pela realização de outros cálculos, como no caso do uso de operadores mais complexos (como chamada de funções). O algoritmo utilizado pode ser descrito pelo seguinte método recursivo:

1) Dividir a expressão em duas partes (esquerda e direita), quando a operação mais exterior é binária, ou parte única (central) se for unária. Detectar a operação central a ser executada.
2) Para cada parte:
   2.1) Se for uma expressão atómica, ou seja, uma referência a uma variável ou constante, ler esse valor no neurónio adequado.
   2.2) Se não for, voltar ao passo 1)
3) Uma vez obtidas as duas partes, juntá-las numa estrutura de sincronização e efectuar a operação central.

Ou seja, torna-se necessário utilizar redes de sincronização, como a da figura 3.3.3. As redes para cálculo de uma expressão seguem o seguinte esquema:

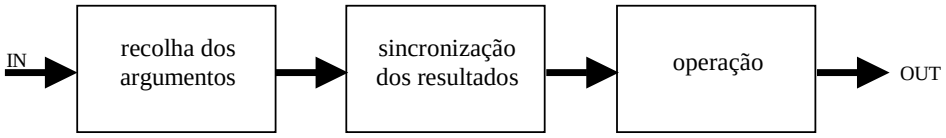


fig. 4.3.7 : cálculo de uma expressão.

Canais

Cada canal possui dois neurónios, um para manter o valor a transferir, e outro para indicar se o canal está ocupado ou não (1 se está vazio, 0 caso contrário). O NETDEF assume o protocolo de comunicação do Occam, onde as operações de enviar e de receber são bloqueantes, i.e., ao enviar para um canal ocupado, o processo fica à espera que o canal fique vazio, e ao pedir informação de um canal vazio, o processo fica à espera que o canal receba algo. Existem dois processos associados aos canais: SEND e RECEIVE.

send
receive

send ::= "SEND" id "INTO" id.

receive ::= "RECEIVE" id "FROM" id.

SEND envia um valor através do canal, bloqueando se o canal está cheio, e RECEIVE que recebe um valor de um dado canal, bloqueando se estiver vazio.

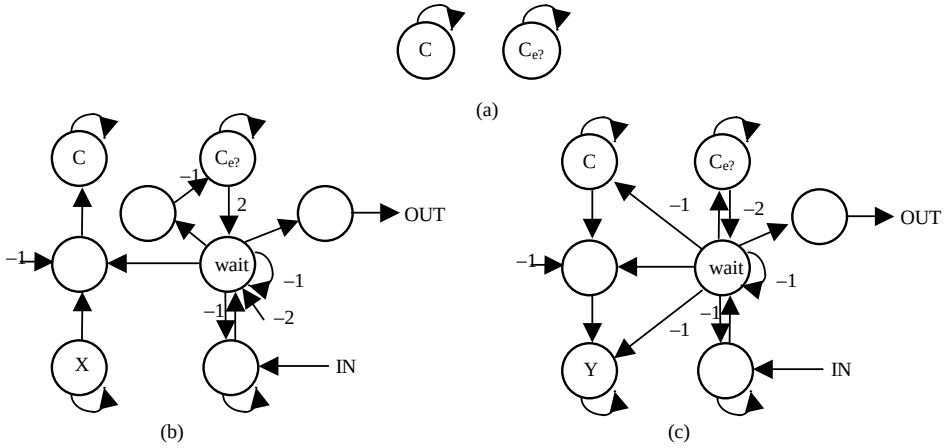


fig. 4.3.8 : (a) VAR C : CHANNEL, (b) SEND X INTO C (c) RECEIVE Y FROM C.

Cada canal tem apenas memória para um valor. No entanto, pode simular-se maiores memórias colocando canais em sequência. O próximo exemplo simula uma memória com dois elementos.

```
RECEIVE X1 FROM C1;
SEND X1 INTO C-TEMP;
RECEIVE X2 FROM C-TEMP;
SEND X2 INTO C2;
```

Como foi referido, os processos SEND e RECEIVE são síncronos. Poderá ser útil possuir um comando que permita continuar a execução de um processo, não esperando a disponibilidade de um recurso externo como é um canal de comunicação. Para um mecanismo assíncrono existe a função ISEMPTY(canal)

que retorna verdadeiro se o canal está vazio, e falso, caso contrário.

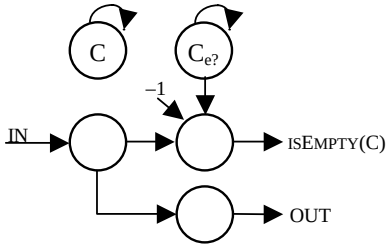


fig. 4.3.9 : função booleana ISEMPTY para chamadas de canais não bloqueantes.

Vectores

vector

Existe ainda a possibilidade de construir tipos de dados mais complexos utilizando vectores. Um vector é um conjunto indexado de valores do mesmo tipo de dados. A seguinte rede mostra o esquema geral de um vector com n elementos do tipo primitivo T, designado pelo identificador A.

fig. 4.3.10 : VAR A : ARRAY [n] OF T.

v[i] := a

a := v[j]

Ou seja, um vector é um conjunto de neurónios, onde cada neurónio guarda o valor de um dado elemento do vector. As duas operações básicas sobre o tipo vector são: a atribuição de um valor a uma componente do vector, e a busca do valor de uma componente.

A estrutura neuronal apresentada é única para cada vector. Cada atribuição é processada, enviando o valor a inserir e a respectiva posição do vector, pelos neurónios X e I.

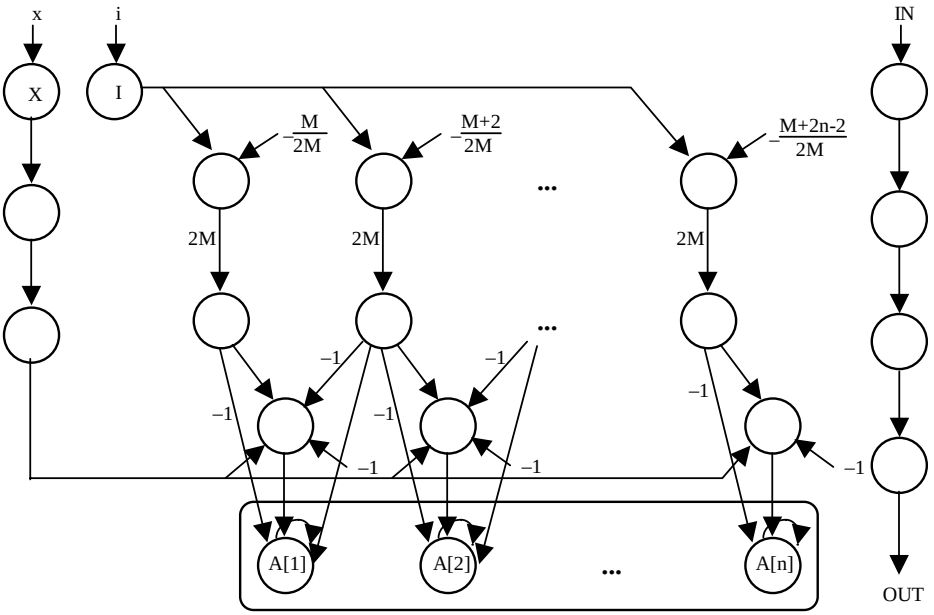


fig. 4.3.11 : inserção do valor x em A[i] .

De igual modo, a rede de obtenção de valores do vector, também é única, mudando apenas o valor do índice introduzido no neurónio J.

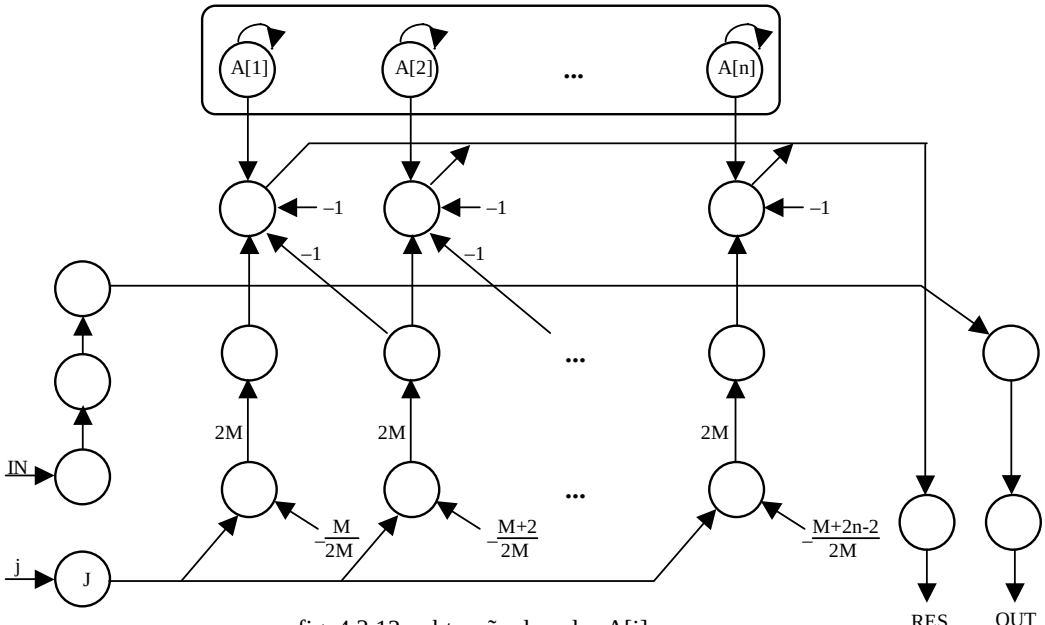


fig. 4.3.12 : obtenção do valor A[j].

Tendo em conta estas duas estruturas auxiliares para cada vector, o espaço requerido para o funcionamento de um vector de N elementos é de 5N+12 neurónios.

A complexidade destas estruturas deve-se ao facto do acesso aos elementos do vector ser dinâmico. Ou seja, pode ser decidido em tempo de execução qual a componente que se pretende consultar ou alterar, bastando para isso, aceder à estrutura vectorial usando uma variável no índice de consulta. Esta possibilidade releva uma potencialidade extra destas estruturas neuronais: o acesso dinâmico de informação em estruturas complexas.

acesso vectorial dinâmico

Dados de Entrada/Saída

Para lidar com o fluxo de entrada e saída entre o programa e o ambiente, o NETDEF usa as primitivas dos canais, com dois vectores próprios, IN_i e OUT_i (a sua dimensão é definida no programa). Cada canal de entrada IN_i está ligado ao ambiente através do canal de comunicação u_i[14].

IN_i
OUT_i

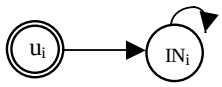


fig. 4.3.13 : ligação ao ambiente do canal IN_i .

Assim, operações de entrada/saída tornam-se simples processos de canais. Um exemplo (este código recebe dados do primeiro canal de entrada para a variável 'a' e envia-os pelo segundo canal de saída):

```
SEQ
  RECEIVE a FROM IN1;
  SEND a INTO OUT2;
ENDSEQ;
```

Estas chamadas, sendo processos de canais, são bloqueantes. Um exemplo de saída de dados assíncrona,

canal assíncrono

```
IF ISEMPTY(C) THEN
  SEND X INTO C
```

[14] Este assunto depende igualmente do contexto da aplicação a desenvolver e da forma como o ambiente é definido. Assim, a definição da estrutura específica destes canais fica em aberto. Em princípio, os canais de entrada precisarão: (a) de uma fila de espera de modo a guardar os dados de entrada ainda não tratados, (b) de manter coerente o estado do canal, i.e., actualizar o neurónio que informa se o canal está vazio ou não.

4.4. Processos

A forma como o NETDEF controla o fluxo da informação depende dos vários tipos de processos que permite. É o conjunto de processos e a sua estrutura que define o programa a executar pela rede final. Cada processo denota uma rede neuronal independente, como já vimos no capítulo anterior. A construção da rede final é então recursiva. Continua a existir a preocupação da modularização, mas agora é possível partilhar informação (via canais ou pelo acesso de informação ao conteúdo de variáveis comuns). Para além do mecanismo de sincronização IN/OUT, existe ainda um canal de entrada especial denominado por RESET para cada processo. Este canal está ligado a um neurónio especial que tem sinapses com peso –1 para todos os neurónios do processo. Assim, se o valor 1 for enviado pelo RESET, a actividade da rede cessa totalmente no instante seguinte. Por razões de simplificação, não mostramos essas ligações nas redes seguintes. Qualquer quadrado indica uma subrede que computa uma expressão ou uma subrede que executa um processo.

RESET

Processos Primitivos

Entre os processos primitivos, para além dos já referidos SEND e RECEIVE, existem ainda os processos de atribuição, SKIP e STOP.

atribuição

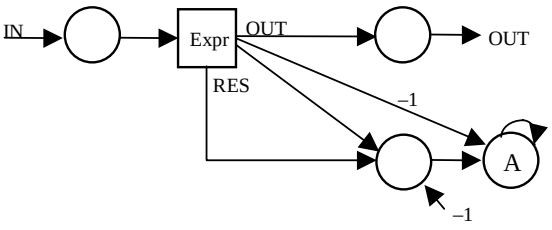


fig. 4.4.1 : Atribuição. A := Expr.

Na atribuição de um valor a uma variável, é necessário calcular primeiro o resultado da expressão, apagar o valor antigo que estava guardado na variável, e actualizar com o novo valor. Isto é realizado pela rede esquematizada acima.

skip
stop

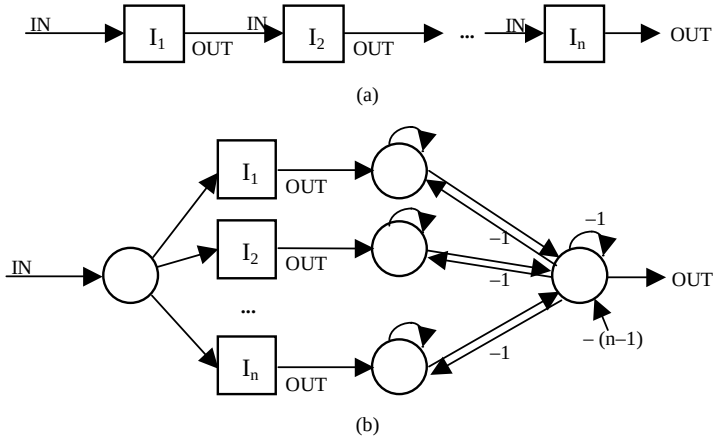
fig. 4.4.2 : Processos (a) SKIP, (b) STOP.

O processo SKIP inicia, não realiza acção nenhuma e termina. O processo STOP inicia, não realiza acção nenhuma e não termina.

Processos Estruturados

Apresentaremos 3 tipos de processos estruturados: os processos compostos, os processos condicionais e os iterados. Existem, no NETDEF, duas formas de compor processos: o bloco sequencial e o bloco paralelo.

seq-block ::= "SEQ" def-vars process { ";" process } "ENDSEQ".
par-block ::= "PAR" def-vars process { ";" process } "ENDPAR".



seq

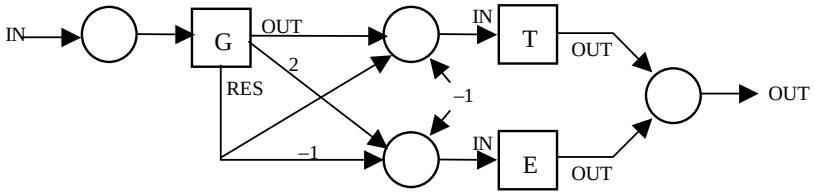
par

fig. 4.4.3 : Processos compostos (a) SEQ I_1; …; I_n ENDSEQ, (b) PAR I_1; …; I_n ENDPAR.

Cada processo num bloco sequencial, precisa somente esperar que o processo anterior termine a sua execução. No bloco paralelo, cada processo inicia a sua execução ao mesmo tempo, independentemente dos outros. Porém, o processo definido pelo bloco paralelo só termina a sua própria execução, quando todos os seus processos terminarem. Esta semântica exige um mecanismo de sincronização, para controlar os diferentes tempos de execução de cada processo.

O processo condicional IF permite a escolha entre dois processos diferentes, a partir do resultado de uma expressão lógica.

if then else

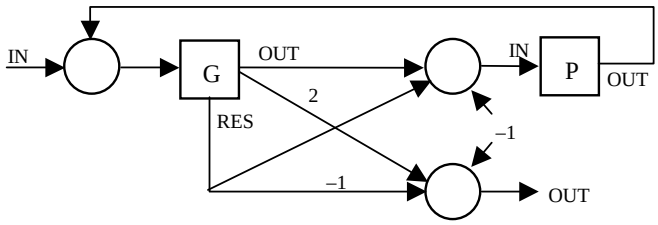


if

fig. 4.4.4 : Processo condicional: IF G THEN T ELSE E.

Podem ser definidos outros dois processos condicionais à custa do processo IF: o processo COND é um bloco sequencial de processos IF, e o processo CASE é um bloco paralelo de processos IF.

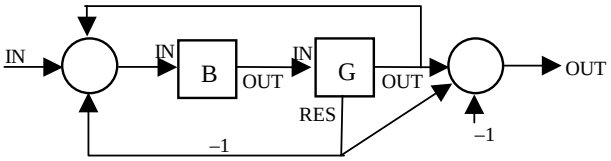
O processo iterado WHILE permite repetir um determinado processo, enquanto uma dada expressão lógica for verdadeira.



while

fig. 4.4.5 : Processo iterado: WHILE G DO B.

Outros processos iterados podem ser construídos de forma semelhante. Outro processo de iteração existente no NETDEF é o REPEAT – UNTIL, cuja rede é a seguinte:



repeat until

fig. 4.4.6 : Processo iterado: REPEAT B UNTIL G .

A estrutura neuronal destes processos é extremamente simples. Tendo em conta que estes processos (composição, condicional e iteração) são a parte central de qualquer algoritmo, a dimensão das redes resultado tenderá a crescer lentamente em proporção com a dimensão do algoritmo.

4.5. Modularização

Como tem sido apresentado, o esforço de modularizar as redes efectuado ao longo do capítulo anterior, é aproveitado aqui no desenvolvimento das redes produzidas para descrever e simular os processos constituintes da linguagem NETDEF. No entanto, verificam-se algumas diferenças.

comunicar por canais

Uma delas é a possibilidade de comunicação entre dois módulos distintos através da utilização de canais. Isso é essencial, dado que um módulo neste contexto, já não é só um bloco fechado efectuando uma operação bem definida, mas pode ser uma unidade operante em face de um exterior dinâmico, constituído tanto por outros módulos, como por um ambiente externo.

comunicar por variáveis

Outra diferença está na possibilidade de acesso de dois ou mais processos ao conteúdo de variáveis. Isto permite uma partilha de informação válida, desde que se tomem em conta os problemas relativos ao paralelismo neste contexto (por exemplo, evitar a escrita simultânea de dois processos sobre a mesma variável). Apresentaremos ainda neste capítulo alguns mecanismos que facilitam a resolução deste problema.

Âmbito das Variáveis

Em principio, cada neurónio pode aceder à informação de todos os outros neurónios. A questão do âmbito das variáveis é uma restrição à priori definida pelo processo de compilação. Isto permite uma programação mais modularizada e mais próxima das linguagens de programação habituais. Como regra geral, uma variável é vista apenas no contexto onde foi definida. Um processo não pode aceder às variáveis definidas pelos processos incluídos nele, mas o contrário é válido. Assim, variáveis definidas no processo central têm uma visibilidade semelhante a variáveis globais, através das quais todos os processos podem ler e escrever informação.

Comandos de Controle de Processos

Cada processo composto (SEQ e PAR) possui o seguinte conjunto de comandos que permitem controlar a sua execução:

- RETRY – anula a actividade da rede e reinicia a sua execução
- ABORT – anula a actividade da rede e termina
- RESET – anula a actividade da rede e não termina
- RETURN *var* – anula a actividade da rede, termina e devolve o conteúdo da variável *var*.

A rede neuronal de um processo, desde que haja a referência a um destes comandos, é acrescentada com a seguinte estrutura:

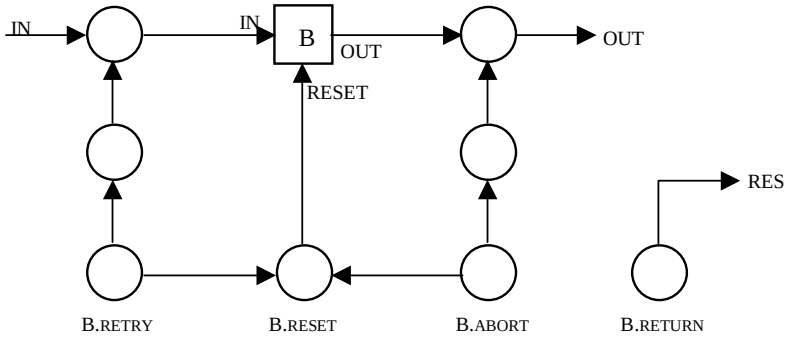


fig. 4.5.1 : estrutura de controle do processo B.

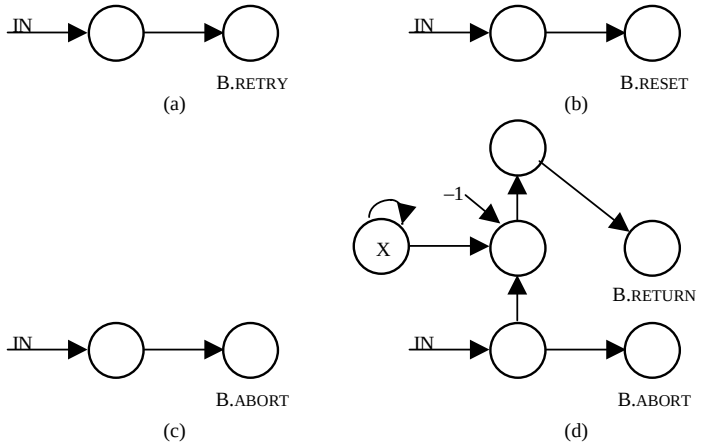


fig. 4.5.2 : redes dos comandos de controle:
(a) RETRY, (b) RESET, (c) ABORT, (d) RETURN x.

operador ^

Existe ainda uma forma directa de alterar o funcionamento de módulos superiores que é através da utilização do operador '^'. Por exemplo, a instrução RESET^ elimina a actividade do módulo pai (eliminando consequentemente a actividade do módulo onde se situa a própria instrução, bem como de todos os

seus módulos irmãos). Estes comandos são justificados nas secções seguintes, quando forem apresentados procedimentos que introduzem a componente temporal às computações neuronais, podendo assim, criar situações de excepção tais como tempo esgotado ou interrupções periódicas da execução normal.

Funções

Da mesma forma que ferramentas como a definição de funções, são úteis na programação, também elas encontram utilidade no NETDEF. Como em outras linguagens, o código de uma função não é duplicado. Elas possuem redes específicas que garantem que apenas uma chamada é executada num determinado instante, bloqueando, se necessário, outras chamadas até ao fim da execução corrente. Isto é essencial, tanto para a execução da função em si, como para a criação de mecanismos efectivos de acesso restrito a dados partilhados. Podem criar-se áreas de segurança em relação ao problema da escrita múltipla, definindo que certos dados só podem ser actualizados através da chamada de uma dada função.

parâmetro por valor

parâmetro por variável

As funções possuem parâmetros por valor (copiar o valor da expressão para o argumento da função) e parâmetros por variável (copiar o valor da expressão para o argumento da função, e quando a função termina, copiar o valor final desse argumento para a variável inicial). Para definir um parâmetro por variável prefixa-se o identificador com a palavra VAR.

A sintaxe de uma função é apresentada no seguinte exemplo:

```
FUNCTION mod (x : INTEGER, y : INTEGER) IS INTEGER
SEQ
  VAR a : INTEGER;
  a := x;
  WHILE a > y DO a := a - y;
  RETURN a
ENDSEQ;
```

No caso de haver expressões complexas nos argumentos, elas devem ser tratadas anteriormente (não se trata de uma verdadeira limitação, apenas reflecte a forma como a estrutura sintáctica foi definida). Por exemplo:

```
w := mod (x+y, succ(z))
```

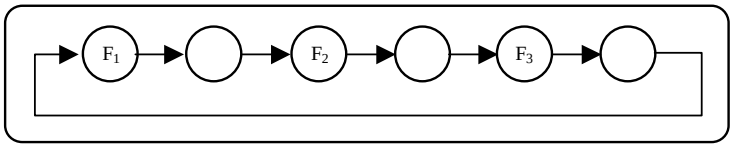
deve ser transformado em:

```
SEQ
  VAR _arg1, _arg2;
  PAR
    _arg1 := x+y;
    _arg2 := succ(z)
  ENDPAR;
  w := mod (_arg1, _arg2);
ENDSEQ;
```

Para que uma função seja apenas acedida uma vez em cada instante, é preciso contornar a questão do paralelismo puro inerente a uma arquitectura neuronal. A questão aqui é mais complexa do que um ambiente concorrente, onde, de facto, apenas um processo é activado num dado instante. Se existir um sistema que executa uma rede neuronal de forma paralela, pode acontecer que dois processos peçam um recurso exactamente no mesmo instante.

Para solucionar esta questão, foi criado um sistema de estafeta em que um processo só pode chamar uma função se e só se conseguir apanhar o testemunho.

ing., token



sistema de estafetas

fig. 4.5.3 : Sistema de estafeta para a função F,
a qual é chamada em três pontos diferentes no programa.

Os neurónios que se intercalam entre os neurónios de chamada são necessários à arquitectura da rede que trata da activação da função. Para uma descrição mais completa da arquitectura de uma função, ler o anexo A. Com este sistema, é possível criar programas com processos de escrita múltipla e paralela numa variável comum. Colocando-a no interior de uma função, fica garantida a exclusão mútua de escritas simultâneas,

```
FUNCTION escrever (valor : T) IS NIL
  varPartilhada := valor;
```

Fechaduras

O sistema de estafetas, utilizado nas funções para garantir exclusão mútua na chamada, pode ser usado igualmente por outros processos através do conceito da *fechadura*. Uma fechadura, é uma variável especial que só pode ser fechada uma única vez antes de ser aberta novamente. Assim, várias partes de código rodeadas por um fecho e uma abertura de uma mesma fechadura, têm garantida a sua exclusão mútua. Para fechar o processo B dentro da fechadura L:

fechadura

```
LOCK L;
  B;
ENDLOCK;
```

A ideia é muito semelhante à das funções. Depois de verificar quantas referências existem à fechadura em questão, cria-se um sistema de estafetas que garante um acesso único à fechadura em cada instante.

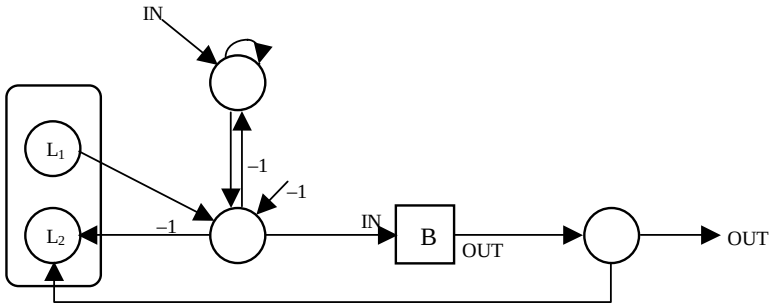


fig. 4.5.4 : Rede padrão para fechaduras. Os neurónios L_1 e L_2 fazem parte de um sistema de estafetas semelhante ao da fig. 4.5.3.

Enquanto o testemunho não está disponível, o sinal de sincronização IN fica guardado. Quando o testemunho finalmente é disponibilizado, ambas as actividades anulam o pendor –1 e o bloco fechado inicia a sua execução, retirando o testemunho do sistema de estafetas e eliminando a actividade IN que fica guardada. Quando o processo termina, o testemunho é reposto para ser disponibilizado para a próxima instância da fechadura.

Excepções

Nas linguagens de alto nível, como o Eiffel (para detalhes ver [Interactive 89]) ou o Ada (para detalhes ver [USDD 83]), as *excepções* são eventos inesperados

excepção

que ocorrem durante a execução, interrompendo o fluxo normal do programa (por exemplo, uma divisão por zero). Algumas excepções são criadas pelo sistema, outras são criadas pelo próprio programa. O (hipotético) *hardware* neuronal é homogéneo, implicando que as únicas excepções criadas pelo sistema são falhas ao nível do neurónio (numa das operações ou na função de activação) ou falha sináptica. Apesar dessas possibilidades, vamos focar apenas as excepções criadas pelo programa, dado que as outras são dependentes do contexto onde a arquitectura neuronal for desenvolvida. As excepções fornecem um controle adicional sobre os processos onde estão inseridas. Elas apenas podem ser aplicadas aos processos compostos, SEQ e PAR. Vejamos um exemplo.

Suponhamos que temos duas funções para realizar o mesmo trabalho, *func-1* e *func-2*. O processo tenta primeiro executar *func-1*. Se ela levantar a excepção *excp-1*, é porque não conseguiu cumprir o objectivo. No tratamento da excepção, o processo altera a variável booleana, reiniciando a sua própria execução. A diferença reside no facto de, agora, é executada a segunda função. Se também esta falhar (levantando a excepção *excp-2*) então o processo termina sem sucesso.

```
func1-falhou := FALSE;
SEQ
  IF NOT func1-falhou THEN func-1  -- com um 'RAISE excp-1'
                      ELSE func-2; -- com um 'RAISE excp-2'
  trabalhoConcluido := TRUE;
EXCEPTION
  WHEN excp-1 DO
    SEQ
      func1-falhou:= TRUE;
      RETRY;
    ENDSEQ;
  WHEN excp-2 DO
    SEQ
      trabalhoConcluido:= FALSE;
      ABORT;
    ENDSEQ;
ENDSEQ;
```

O comando RAISE E levanta a excepção E. Cada excepção tem um processo associado, mais o comando específico PROPAGATE, que apaga o processo e

levanta a mesma excepção no processo superior. Se nada for dito no fim do processamento de uma excepção, será executado um comando RETRY.

Cada processo com tratamento de excepções tem a sua arquitectura alterada da seguida forma:

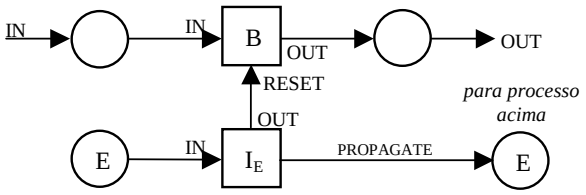


fig. 4.5.5 : A estrutura para tratamento de excepções do processo B.
I_E é o processo associado com a excepção E.

Cada excepção tem um neurónio próprio onde recebe o sinal de excepção do processo. Com este tipo de estrutura, RAISE é definido por

```
PAR
  E:=1;      ≡     RAISE E
  STOP;
ENDPAR;
```

Existe um efeito de propagação para as excepções não tratadas. Se um processo não trata uma dada excepção E, é colocado por defeito o comando,

```
WHEN E DO PROPAGATE;
```

Qualquer comando que propaga uma excepção não é reiniciado (a não ser que o processo superior tente novamente a sua execução).

Alarmes

alarme

Em aplicações reais, alguns processos podem criar situações em que o sistema fica bloqueado. Por exemplo, os processos SEND E RECEIVE são bloqueantes. Para tratar alguns destes problemas, o programador possui à sua disposição alarmes que permitem cancelar um processo, se ao fim de um dado período de tempo este não tiver terminado.

O primeiro tipo de alarme é denominado por TRY.

try ::= "TRY" "(" id ")" process.

Ele garante a terminação de um processo, se ele não terminar ao fim de um dado tempo (fornecido por uma variável inteira).

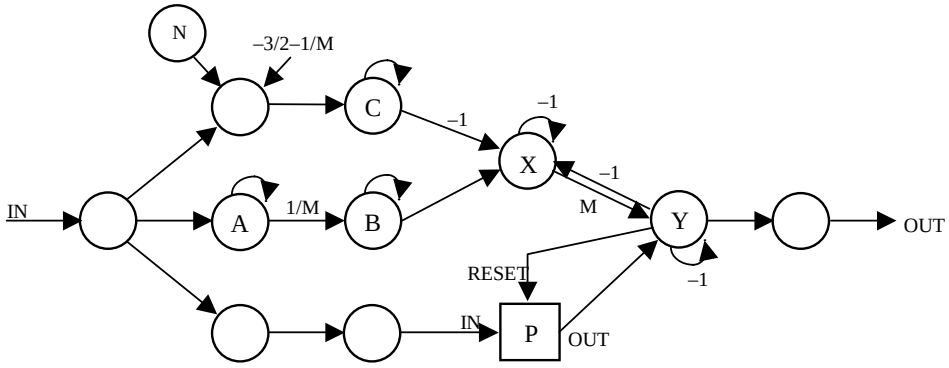


fig. 4.5.6 : TRY(N) P.

onde o neurónio X tem sinapse de peso –M e o neurónio Y tem sinapse de peso -1 para os neurónios A, B e C.

Alarmes de atrasos adiam a execução de um processo por um dado tempo.

delay ::= "DELAY" "(" id ")" process.

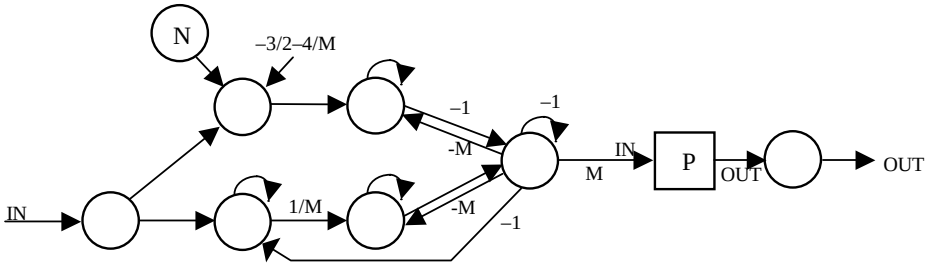


fig. 4.5.7 : DELAY(N) P.

Alarmes cíclicos reiniciam ciclicamente a execução de um processo ao fim de um dado intervalo de tempo. Podem ser usados para simular interrupções do sistema para tratamento de processos urgentes.

cycle ::= "CYCLE" "(" id ")" process.

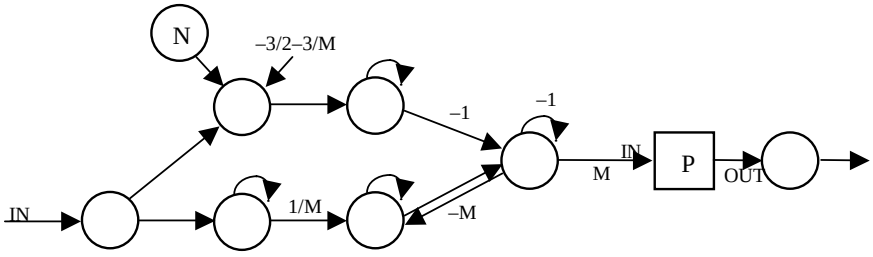


fig. 4.5.8 : CYCLE(N) P.

Cada alarme prefixado a um processo constrói um novo processo. Assim, é possível acoplar sequencialmente vários alarmes a um mesmo processo. Por exemplo,

```
CYCLE (10000) TRY(50) IF ok = 1 THEN SEND X INTO C;
```

significa que a cada 10000 instantes, o processo irá verificar se a variável ´ok´ tem o valor 1, para enviar o valor X pelo canal C. Se o valor não puder ser enviado ao fim de 50 instantes, o processo é terminado.

4.6. Compilador de NETDEF

A partir do estudo desenvolvido e apresentado neste capítulo, foi realizado um programa capaz de compilar a linguagem NETDEF com os processos descritos. O próprio desenvolvimento do programa foi essencial para optimização e correcção de erros das redes apresentadas e para a estrutura final deste capítulo.

Inicialmente, o núcleo básico do programa foi programado em Turbo Pascal, utilizando-se o construtor de gramáticas YACC para a definição da linguagem. Devido a sérias limitações de memória, dada a complexidade do compilador, fez-se a migração para o ambiente de desenvolvimento gráfico Delphi da Borland. Para além de todo um ambiente mais adequado para a concretização dos nossos objectivos, foi ainda possível construir um programa para Windows, com um conjunto de componentes gráficas que facilitam a percepção e o uso das potencialidades do compilador. Entre elas, existe um editor de texto próprio especializado na programação NETDEF, com um conjunto de ferramentas de diminuição de erros sintácticos e aumento consequente da facilidade e rapidez na programação. Um sistema de depuração em tempo de execução da rede produzida, com as utilidades típicas dos ambientes integrados de programação. Para isto, foi necessário criar ferramentas gráficas capazes de mostrar, na forma mais intuitiva possível, a estrutura da rede produzida, os neurónios individuais e as suas ligações pré e pós-sinápticas (bem como os pesos respectivos). Também foi programado um sistema de simulação da rede, para conferir o seu correcto funcionamento. Como consequência, durante a

programação do compilador, foram aperfeiçoadas e corrigidas as redes descritoras dos vários processos da linguagem.

Para facilitar a consulta e aprendizagem da linguagem e da aplicação, foi desenhado um ficheiro de apoio no sistema de ajuda do Windows, onde se apresenta uma introdução ao problema, uma descrição pormenorizada da cada função e das suas redes respectivas, e ainda um glossário com todas as palavras reservadas da linguagem, bem como exemplos de utilização e a sua correcta sintaxe.

Como exemplo da execução do compilador, vejamos o seguinte programa:

```
(* SIMULAÇÃO DA ENTRADA DE DOIS VALORES INTEIROS DO AMBIENTE E POSTERIOR
 * SOMA DESSES VALORES PARA SER ENVIADO PELO 1º CANAL DE SAÍDA
 *)

NETDEF SOMA(2,1) IS

VAR  I, J, RES  : INTEGER;

SEQ
  -- SIMULAÇÃO DA ENTRADA DOS VALORES
  VAR I1, I2 : INTEGER;

  I1 := 100;
  I2 := 123;
  PAR
    SEND I1 TO IN1;
    SEND I2 TO IN2;
  ENDPAR;

  PAR
    RECEIVE I FROM IN1;
    RECEIVE J FROM IN2;
  ENDPAR;

  RES := I + J;
  SEND RES TO OUT1;   -- ENVIO DO VALOR PELO CANAL DE SAÍDA

ENDSEQ;
ENDDEF.
```

Este exemplo recebe dois valores do exterior (simulado aqui por um processo paralelo que introduz nos canais de entrada, os respectivos valores), calculando a soma desses valores e enviando o resultado para o exterior. Ao compilar, o programa informa que construiu uma rede neuronal com 82 neurónios e 159 sinapses, apresentando uma rede no seguinte formato:

```
MN0IN(I+1) = SIGMA( )
MN0OUT(I+1) = SIGMA( + 1,0 * SEQ1OUT(I) )
VARCMN0:IN1(I+1) = SIGMA( + 1,0 * VARCMN0:IN1(I) + 1,0 * SEND1VAL(I) -
        1,0 * RECV1WAIT(I) )
VARCMN0:IN1FLG(I+1) = SIGMA( + 1,0 * MN0IN(I) + 1,0 * VARCMN0:IN1FLG(I)
        - 1,0 * SEND1FLG(I) + 1,0 * RECV1WAIT(I) )
VARCMN0:IN2(I+1) = SIGMA( + 1,0 * VARCMN0:IN2(I) + 1,0 * SEND2VAL(I) -
        1,0 * RECV2WAIT(I) )
VARCMN0:IN2FLG(I+1) = SIGMA( + 1,0 * MN0IN(I) + 1,0 * VARCMN0:IN2FLG(I)
        - 1,0 * SEND2FLG(I) + 1,0 * RECV2WAIT(I) )
VARCMN0:OUT1(I+1) = SIGMA( + 1,0 * VARCMN0:OUT1(I) + 1,0 * SEND3VAL(I)
        )
...
```

É apresentado, para cada neurónio, o conjunto das suas ligações sinápticas de entrada e quais os pesos respectivos.

Para evitar a simulação de entradas do ambiente como no exemplo anterior, foram incluídas no compilador de NETDEF duas ferramentas que facilitam essa troca de informação entre a rede e o exterior. Os dados de entrada podem ser guardados num arquivo de texto onde se informa qual o valor a ser enviado, para que canal e em que momento do tempo isso deve acontecer. Quando uma condição é obtida, o simulador encarrega-se automaticamente de introduzir o valor no canal respectivo simulando assim, o ambiente onde a rede estaria inserida. Se este canal estiver ocupado, o simulador possui um sistema de filas de espera onde são armazenados os valores enquanto não existir disponibilidade para os inserir na rede. Existe igualmente a possibilidade de guardar os valores produzidos pela rede nos seus canais de saída, em um outro arquivo para posterior análise numa ferramenta apropriada.

Este compilador mais o arquivo de ajuda, juntamente com um conjunto de informação complementar, incluindo o artigo [Neto *et al.* 2001a], podem ser encontrados na página *www.di.fc.ul.pt/~jpn/netdef/netdef.htm.*

5. Multiplicação e Aprendizagem

Tendo como base a estrutura desenvolvida no capítulo anterior, estudaremos neste capítulo uma possível forma de integrar nas redes-σ a computação simbólica com os processos de aprendizagem.

Na década de 90 surgiram múltiplas propostas de integração simbólica e sub-simbólica, tanto do ponto de vista cognitivo e explicativo, como do ponto de vista aplicacional. Cada tipo de computação tem os seus pontos fortes e fracos. A computação simbólica é indicada para processamento de aplicações onde o nível representacional é relevante, como por exemplo, em sistemas de dedução lógica, ou no processamento de linguagem natural. A sub-simbólica é utilizada principalmente em processos de adaptação e aprendizagem. O objectivo comum de propostas híbridas é obter sistemas que possam aproveitar o melhor de ambos os mundos, aumentando assim, a sua eficácia global.

Referiremos alguns trabalhos nesta área, bem como a comunidade científica tem definido e agrupado os vários pontos de vista desenvolvidos. Em seguida, será apresentado um novo modelo neuronal, generalizando o modelo das redes-σ de modo a incluir novas ferramentas, nomeadamente, processos de aprendizagem. Diferentes modos de aprendizagem, como a aprendizagem

Hebbiana e Competitiva, serão tratados e implementados na arquitectura neuronal existente.

5.1. Trabalhos Relacionados

Ao estudar o conjunto de trabalhos relacionados, observámos que podemos estabelecer a seguinte classificação tendo em conta como os sistemas são organizados internamente, tanto no nível da representação como no nível de processamento (cf. [Wilson e Hendler 93] e [Sun e Bookman 95]):

- A computação simbólica é processada de forma especializada e local, por redes apropriadas;
- A computação simbólica é processada de forma distribuída pela rede neuronal;
- Separação dos diferentes tipos de processamento (simbólico e sub-simbólico) em módulos (que será a perspectiva desenvolvida neste capítulo) que interagem de forma mais ou menos homogénea;
- Usar redes neuronais como base para a arquitectura simbólica e para as operações sobre os símbolos representados.

Os trabalhos que reflectem o primeiro ponto, representam conceitos individuais em neurónios individuais, e as relações sobre esses conceitos são expressas pelas conexões e seus pesos respectivos. Existe assim, um isomorfismo entre o sistema pretendido e a rede resultante, produzindo aplicações paralelas de processamentos à partida sequenciais. No trabalho de Trent Lange (cf. [Lange 95]), é definido um sistema neuronal onde é possível representar inferência por associação de variáveis e aplicação de regras. A associação é realizada usando diversos padrões de activação que identificam unicamente o conceito associado à variável. As regras são previamente construídas na estrutura sináptica da rede, resultando numa espécie de rede semântica.

Dentro do mesmo espírito, Lacher e Nguyen, (cf. [Lacher e Nguyen 95]), apresentam uma forma de combinar sistemas periciais com aprendizagem

ing., expert systems

neuronal. Os autores desenham o sistema pericial na arquitectura neuronal para ser capaz de aprender, melhorando a sua capacidade de classificação.

No trabalho de Alon, Dewdney e Ott (cf. [Alon *et al.* 91]) mostra-se uma forma de simular a execução de autómatos finitos através do uso de redes neuronais, fazendo-as processar informação simbólica codificada nas activações produzidas pela execução da respectiva rede.

Respeitante ao segundo ponto, a representação distribuída de conceitos simbólicos tende a utilizar essa vantagem intrínseca do paradigma neuronal. Esta abordagem é marcadamente conexionista. O seu principal objectivo é criar sistemas neuronais capazes de representação simbólica e onde o seu processamento possa ser executado através de algoritmos de aprendizagem (cf. [Sharkey e Jackson 95] para uma análise de várias teorias de representação distribuída).

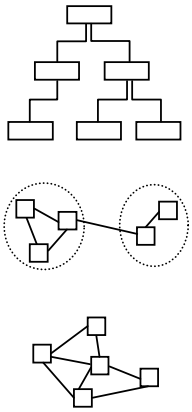
Tanto o trabalho de Pollack (cf. [Pollack 90]) como o de Miikkulainen (cf. [Miikkulainen 95]) apresentam sistemas neuronais capazes de processar sequências simbólicas, divididas em cláusulas apropriadas (por exemplo, as várias palavras de uma frase na língua natural) guardando as suas representações de forma dispersa e obtendo as propriedades típicas desta abordagem, como capacidade de generalização ou degradação suave da memória associativa.

Em [Browne 98], é mostrado como o processo lógico da resolução pode ser aproximado usando um sistema neuronal que guarda as representações lógicas de forma distribuída. O sistema, como o próprio autor afirma, é ainda limitado, mas indica uma direcção para o tratamento de uma tarefa tipicamente simbólica num modelo neuronal.

Resolução:
$a \vee b$
$\neg a \vee c$
$\therefore b \vee c$

Outra abordagem é encontrada em [Medler *et al.* 99] onde, em vez de extrair informação de áreas locais dentro da rede neuronal, o algoritmo procura reconhecer padrões comuns por toda a estrutura da rede, recolhendo conhecimento distribuído, indo assim, ao encontro da forma como os algoritmos de aprendizagem procedem.

A terceira abordagem baseia-se no uso de módulos de processamento independentes, onde diferentes tipos de computação possam ser executados. Esses módulos podem ser sistemas simbólicos ou conexionistas, e é na sua interligação e comunicação que reside principalmente o foco dos diversos estudos. Entre os pontos principais, estão os protocolos de comunicação (a estrutura do fluxo de dados), o grau de acessibilidade e de controlo entre módulos e a definição da estrutura geral do sistema (hierárquico, heterárquico ou anárquico[15]). O trabalho desenvolvido neste capítulo está dentro do espírito deste tipo de abordagem.



O sistema KBANN (cf. [Towell 91]) explora a capacidade de usar ao mesmo tempo conhecimento teórico traduzido em conjuntos de regras e conhecimento empírico obtido a partir de um conjunto de acontecimentos aprendidos. Os dois módulos interagem um com o outro, construindo regras novas e apagando outras consideradas obsoletas à medida que nova informação é recolhida. O refinamento do conhecimento é efectuado do seguinte modo: o conhecimento simbólico prévio é traduzido para uma rede neuronal apropriada. Esta é treinada usando a retropropagação com os exemplos de treino disponíveis. A rede final é então novamente traduzida para um sistema de regras que traduzem o estado actual dessa rede.

Um aperfeiçoamento deste sistema, denominado INSS (cf. [Osório e Amy 99]) tenta retirar as limitações do sistema anterior, nomeadamente no que respeita ao modelo neuronal e ao algoritmo de aprendizagem, usando o método de aprendizagem 'Cascade Correlation' (cf. [Fahlman e Lebiere 90] como fonte científica, ou [Haykin 99] para uma apresentação didáctica do método). Uma das maiores diferenças e vantagens para o sistema anterior é que o INSS é

[15] Um sistema hierárquico define uma árvore de responsabilidade e de controle que estabelece o funcionamento interno dos componentes do sistema. Num sistema heterárquico, cada módulo pode interagir com módulos com determinadas funções, não existindo um conceito forte de controle entre um módulo e outro, mas sim de cooperação e/ou prestação de serviços apropriados. Já num sistema anárquico não existem quaisquer restrições de comunicação entre os vários elementos do sistema.

capaz de aprender e construir regras de forma incremental, sem necessitar de reaprender todo o conjunto de dados adquiridos no passado.

Em [Handelman *et al.* 90], é apresentado um sistema para controle de um robot, onde são integrados sistemas de regras e redes neuronais. O sistema tem várias estruturas de controle. Uma primeira, de mais baixo nível, denominada reflexos inatos, é um sistema de regras pré-definido. Este sistema envia ao nível acima, uma rede neuronal, uma quantidade de amostras supervisionadas para esta aprender. Depois da aprendizagem inicial, ambos os sistemas interagem e colaboram no controle do sistema, existindo a possibilidade de reaprendizagem quando surgem alterações no ambiente ou nas tarefas a desempenhar.

Num sistema designado por SCRuFFy (cf. [Hendler e Dickens 91]), existe a operação e interacção de dois módulos distintos: um que realiza a execução de um sistema pericial e outro que executa um algoritmo de retropropagação, sendo o fluxo de informação regulado para que a rede neuronal forneça informação relevante ao sistema pericial.

Igualmente no trabalho de Wilson e Hendler (cf. [Wilson e Hendler 93]), é apresentado um sistema de interface que permite estabelecer uma comunicação efectiva entre o módulo sub-simbólico (onde é executado um algoritmo de aprendizagem) e o módulo simbólico que utiliza essa mesma informação. O sistema torna transparente a definição dos diversos módulos, deixando para o sistema de controle do protocolo de comunicação, designado por supervisor, a tarefa de traduzir o fluxo de informação no formato apropriado.

O sistema RAPTURE de Mahoney liga um sistema pericial com uma modificação da retropropagação. O algoritmo de aprendizagem é utilizado para ajustar e refinar os pesos dos factores do sistema pericial (cf. [Mahoney 92]).

Em [Apolloni e Zoppis 99], os autores apresentam um sistema híbrido modular (com módulos simbólicos e outros sub-simbólicos) para gerir um sistema de ordenação efectuado somente por redes neuronais. Os parâmetros de ambos os tipos de módulos são ajustados usando um algoritmo de retropropagação. O

módulo simbólico tem como função principal, manter um determinado conhecimento prévio para facilitar a aprendizagem e a tarefa de ordenação.

Chen e Honavar, em [Chen e Honavar 99], apresentam um modelo neuronal modular capaz de processar símbolos no contexto de uma determinada gramática. É construído um analisador lexical (para filtrar palavras não pertencentes à gramática em questão), e todas as estruturas de dados necessários para a análise sintáctica de uma linguagem.

Na quarta abordagem, os conceitos simbólicos são construídos a partir de pequenas redes neuronais, procurando associar as vantagens da computação simbólica e da tolerância a falhas. No trabalho de Ron Sun (cf. [Sun 95]), é apresentada uma arquitectura de dois níveis. Um nível é a rede de inferência com os neurónios representando conceitos e as sinapses regras entre esses conceitos. O nível inferior é uma réplica do primeiro, onde é utilizada uma representação distribuída. A interacção destes dois níveis é usada para deduzir informação a partir de funções de classificação baseadas numa mistura de aplicações de regras (do nível simbólico) e de padrões de similaridade (no nível sub-simbólico).

Regras M-de-N

Igualmente neste âmbito, em [Setiono 2000], são extraídas regras M-de-N de redes neuronais já treinadas com conjuntos de treino bipolares (–1 ou 1). Estas regras, mais expressivas que as condicionais padrão, são do tipo:

exemplo XOR:
SE (só uma das entradas é 1)
ENTÃO 1
SENÃO 0

> SE (exactamente M dos N antecedentes forem verdadeiros)
> ENTÃO ...
> SENÃO ...

Noutro trabalho relacionado (cf. [Tsukinoto 2000]), é apresentado um método de decomposição de qualquer rede neuronal que respeite a seguinte restrição: a sua função de activação na camada de saída deve ser monótona (como a função sigmoidal ou a função σ). Deste modo, o sistema apresentado pode ser usado mesmo em redes de múltiplas camadas ou redes recorrentes e não depende do algoritmo de aprendizagem utilizado. Esta decomposição devolve um conjunto de regras de decisão.

No trabalho de Frasconi, Gori e Sperduti (cf. [Frasconi *et al.* 98]) é apresentado um sistema de aprendizagem neuronal capaz de incorporar características simbólicas representadas entre os vários elementos de entrada, ou seja, capturar e utilizar informação previamente conhecida do problema, tendo sido esta codificada em estruturas simbólicas como grafos ou sequências.

Em [Tan 97], o autor mostra um sistema de aprendizagem denominado por "Cascade ARTMAP", onde integra computação neuronal com sistema de regras de classificação difusas. O sistema é capaz de guardar informação prévia através do seu sistema de regras para maximizar a qualidade da classificação final, mas também tem a capacidade de aprender e alterar (corrigir) o seu sistema de regras através de um algoritmo de aprendizagem específico. Ainda relacionado com a questão da lógica difusa, em [Li e Chen 2000], é demonstrada a equivalência computacional entre este tipo de lógica e as redes neuronais acíclicas com função de activação não linear. Esta equivalência significa que é possível aproximar arbitrariamente o comportamento de um sistema por outro.

Outra possibilidade é a extracção de conhecimento simbólico a partir da estrutura neuronal final, depois de estabilizado o processo de aprendizagem. Duas formas de produção de conhecimento simbólico são regras de classificação (cf. [Setiono e Liu 96], [Alexander e Mozer 99], [Taha e Gosh 99], [Li 98], [Sun 99]), ou autómatos finitos (cf. [Giles *et al.* 92], [Omlin e Giles 96b]). Uma lista de sistemas de produção de regras a partir de diversas estruturas neuronais pode ser encontrado em [Mitra e Hayashi 2000]. Estudos mais formais sobre a integração simbólica e sub-simbólica são menos comuns, e acreditamos que nesta área existe um grande potencial de desenvolvimento de trabalho científico (cf. [Lopin 99], [Zhang 99]).

Redes de Ordem Superior

O modelo que vai ser apresentado é um modelo neuronal de ordem superior. Estas redes possuem neurónios com funções de transferência de ordem superior, ou seja, os neurónios podem multiplicar os valores que recebem

através das suas sinapses de entrada. Este tipo de redes pode ser encontrado na literatura científica, onde a maior parte do trabalho realizado foca as capacidades computacionais superiores deste tipo de rede, em comparação com as redes de ordem simples.

Pollack, em [Pollack 87] construiu um modelo neuronal de ordem superior finito com propriedades universais. Em [Sun *et al.* 94] é considerado um sistema neuronal de ordem dois com o poder das máquinas de Turing. [Goudreau *et al.* 94] trabalham com redes neuronais com a função degrau como função de activação. Eles mostram que redes de ordem dois com uma camada são estritamente mais poderosas que redes de ordem um com igualmente uma camada. Uma aplicação para aprendizagem de autómatos finitos usando redes de ordem dois é descrita em [Giles *et al.* 92] e [Omlin e Giles 96a]. Um trabalho posterior (cf. [Omlin *et al.* 98]) mostra a capacidade de redes deste tipo para representar autómatos finitos difusos, ou seja, sistemas onde podem coexistir diversos estados activos num dado instante, sendo cada um ocupado num determinado grau. Outro trabalho que utiliza redes de segundo grau para codificação de autómatos finitos por ser lido em [Carrasco *et al.* 2000]. Ainda a respeito de autómatos, em [Tiño e Sadja 95], é também apresentado uma rede de segunda ordem capaz de aprender autómatos de Mealy (autómatos finitos com uma função de output, desempenhado tarefas de tradução de palavras de uma linguagem em palavras de outra) a partir de um conjunto de amostras.

Em [Santos e Zuben 99], é apresentado um algoritmo de gradiente descendente adaptado a topologias de segunda ordem. Outro modelo, destinado a tarefas de controle com redes de ordem dois, denominado redes NARX, pode ser encontrado em [Siegelmann *et al.* 97] e em [Lin *et al.* 98].

5.2. NETDEF+

O objectivo em introduzir este novo modelo neuronal (uma extensão das redes-σ apresentadas anteriormente), é permitir a integração da aprendizagem

nas ferramentas computacionais introduzidas no capítulo anterior. Para isso, o novo modelo deve ser criado para preservar a:

- Simplicidade – Deve permanecer o mais próximo do modelo da rede neuronal original e ao mesmo tempo ser compatível com as redes compiladas pelo compilador de NETDEF.
- Expressividade – Deve ser capaz de representar e computar um variado leque de ferramentas e mecanismos típicos da computação sub-simbólica.
- Modularidade – O aspecto modular, característico das redes até agora apresentadas, deve permanecer inalterado.

Para satisfazer estes requisitos, foi decidido estender o modelo com a possibilidade de existência de conexões neuro-sinápticas. Apesar de não existir nesta dissertação uma preocupação com a plausibilidade biológica, a existência de complexas árvores dendríticas nos cérebros animais é reconhecida (cf. [Sheperd 94] para maiores detalhes e [Neto *et al.* 97a] para uma demonstração do poder computacional deste tipo de estruturas). Neste modelo, a sua utilização central será a de transportar valores de activação neuronal para modificação de pesos sinápticos específicos.

A semântica desta nova ligação neuro-sináptica é descrita como a multiplicação dos dois valores de entrada, nomeadamente, do valor de activação do neurónio de entrada e do valor actual da conexão sináptica. Na equação dinâmica isso é descrito por um sistema dinâmico de ordem superior, nomeadamente, de ordem dois. Considerando a definição de rede-σ em 2.2.13 apresentamos o novo modelo neuronal.

Definição 5.2.1: Uma *rede-*σ^+ é uma rede neuronal recorrente R = <X,U,Y>, onde a computação realizada por cada neurónio (i.e., a composição da função de transferência e da função de activação neuronal) é dada pela seguinte expressão: *rede* -σ^+

$$x_j(t+1) = \sigma(P_j(x_1(t), ..., x_N(t), u_1(t), ..., u_M(t))) \qquad (25)$$

onde $P_j(.)$ é um polinómio de coeficientes racionais e a função σ é descrita pela função (4) apresentada na secção 2.2. ◁

A figura seguinte mostra uma conexão neuro-sináptica ligando o neurónio x à sinapse que conecta os neurónios y e z.

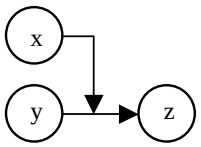


fig. 5.2.2 : Notação gráfica de uma ligação neuro-sináptica.

A dinâmica do neurónio z é dada pela seguinte equação de ordem dois:

$$z(t+1) = \sigma(\beta(x(t), y(t)))$$
$$= \sigma(2ax(t).y(t) - a(x(t) + y(t)) + 0.5a + 0.5) \qquad (26)$$

A função $\beta(x, y)$ é obtida a partir do seguinte raciocínio: Pretende-se multiplicar as duas entradas (no exemplo descrito pela figura anterior, os valores de x e y) da conexão. Esses valores, porém, estão codificados (dentro do intervalo [0,1]) como reais no intervalo [-a, a]. É necessário descodificar os valores, multiplicá-los e codificar novamente o resultado obtido (dado que não se obtém o resultado desejado ao multiplicar directamente os resultados codificados, como aconteceu para a soma e subtracção na figura 4.3.5).

A maior complexidade inerente a este modelo será compensada, não em termos de poder computacional – o modelo anterior já é computacionalmente completo – mas na facilidade com que certas funções específicas e úteis no contexto da aprendizagem podem ser implementadas.

Multiplicação

O módulo da multiplicação binária é dado pela seguinte rede.

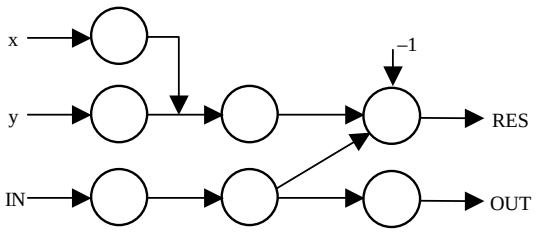


fig. 5.2.3 : Multiplicação.

A terceira camada da rede é necessária devido ao seguinte facto: quando está inactiva, a rede recebe o valor zero pelos canais de dados x e y. Aplicando a função β aos valores obtém-se um grande valor de activação que tem de ser eliminado para não sair pelo canal de resultado e interferir com os subsequentes módulos.

β(0,0) = 0.5a + a

5.3. Estruturas Dinâmicas

Uma rede baseada no modelo das redes-σ^+ permite modificar em tempo de execução alguns dos seus pesos sinápticos. Isto significa que a arquitectura neuronal pode ser auto-modificável, o que admite a possibilidade de adaptação e até a construção de algoritmos de aprendizagem como veremos nas secções seguintes. O caso mais simples refere-se à alteração directa de valor de uma determinada ligação sináptica. O processo LINK(*neurónio*, *valor sináptico*, *neurónio*) descrito na figura seguinte mostra como se efectua o mecanismo de mudança.

Processo LINK

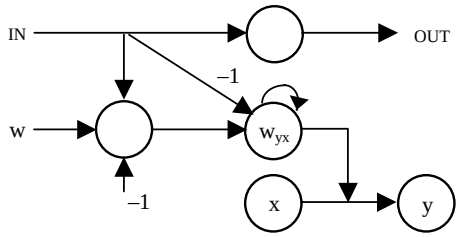


fig. 5.3.1 : Esquema da rede do processo LINK (x, w, y).

O valor da sinapse é guardado por um neurónio específico que é conectado à sinapse dos dois neurónios em questão por uma sinapse neuro-sináptica, efectuando assim a desejada multiplicação. Antes de inserir o novo valor, o módulo atrasa a atribuição para poder apagar o valor anterior. Quando o módulo indica o fim do seu processamento, o novo valor da sinapse já está guardado na memória do neurónio específico. Por exemplo, se entre os neurónios x e y, for executado o processo LINK(x,0,y), significa que a ligação entre os dois é anulada, implicando para todos os efeitos, a eliminação da sinapse. No entanto, ela pode ser reactivada executando um processo

LINK(x,w,y), com w≠0. Anulando todas as sinapses de um determinado neurónio, implica na sua separação do resto da rede.

É esta versatilidade que permite o mecanismo dinâmico de alteração da rede por ela própria. Veremos em seguida como isso pode ser aplicado para integrar na arquitectura algoritmos de aprendizagem.

5.4. Funções de Aprendizagem

Um sistema capaz de se alterar através de estímulos externos pode ser, em princípio, capaz de aprender. Um algoritmo de aprendizagem especifica, de uma forma estruturada, como essa alteração deve ocorrer. No contexto deste modelo, o acto de aprendizagem será representado pelo resultado da execução de um algoritmo codificado na rede, que permite alterar alguns dos seus pesos sinápticos para adequar-se aos dados recebidos. À partida, isto é possível usando o mecanismo neuro-sináptico do novo modelo neuronal. Conhecendo o conjunto de processos do NETDEF, como se poderá construir uma extensão da linguagem (a ser chamada NETDEF+) capaz de descrever algoritmos de aprendizagem? (cf. [Neto *et al.* 2000] para um artigo sobre estas questões).

Operadores de Aprendizagem

ing., Learning Function

Antes de entrar nos pormenores internos de funcionamento, descreveremos qual será a interface que o programador obtém para utilizar este tipo de ferramenta. Para a linguagem NETDEF+, um módulo de aprendizagem é uma função especial. Esta *função de aprendizagem* necessita de uma fase de aprendizagem. Para além disso, podem ainda ter alguns operadores adicionais. A lista completa é apresentada nos seguintes pontos:

- Definição – a topologia da rede depende do tipo de algoritmo pretendido.
- Inicialização – as ligações da rede são inicializadas.
- Aprendizagem – a rede adapta-se à informação fornecida pelo programa.
- Consulta – a rede devolve a resposta de uma amostra não classificada.

Estrutura Interna

Um módulo de aprendizagem é constituído por duas partes interdependentes, a rede de controle e a rede de aprendizagem. A rede de controle tem como objectivo controlar e regular o processo de aprendizagem quando este é iniciado. Uma parte será definida pela própria linguagem NETDEF. É ela que recebe o pedido do exterior e a informação associada, adaptando a rede de aprendizagem de forma adequada. A rede de aprendizagem é definida pela topologia típica para o algoritmo de aprendizagem em questão (por camadas, num esquema bidimensional). É ela que conserva o conhecimento produzido pela adaptação.

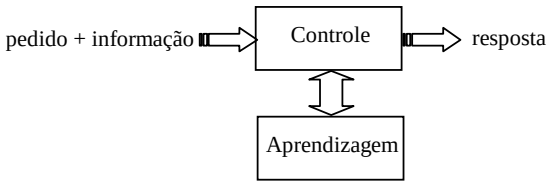


fig. 5.4.1 : Esquema de um módulo de aprendizagem.

São estas as duas partes que constituem o módulo de aprendizagem. Elas são construídas usando o mesmo tipo de modelo de rede neuronal (o modelo *rede-*σ^+), sendo, assim, uma construção neuronal homogénea.

É esta integração da computação simbólica (através das redes produzidas pelo compilador de NETDEF) com a computação sub-simbólica (com a integração e uso dos algoritmos de aprendizagem das secções seguintes) que serve de motivação central para este capítulo.

O aspecto modular do sistema não se altera. Isso permite múltiplos usos do sistema, como a aprendizagem separada de problemas não relacionados; a utilização de módulos de controle para suporte de tarefas sub-simbólicas; a divisão de tarefas não interdependentes em módulos paralelos, obtendo assim, um maior controle sobre a complexidade do problema.

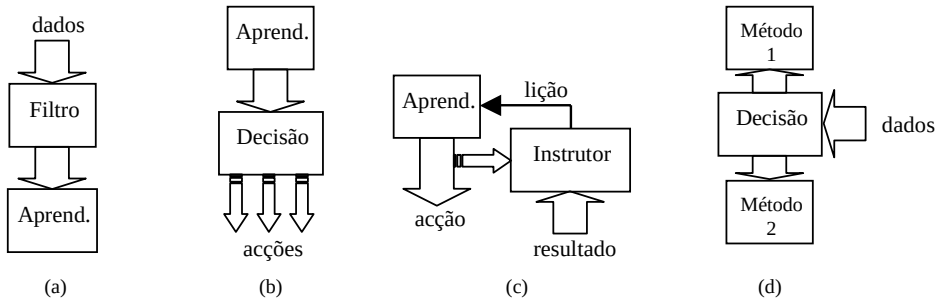


fig. 5.4.2 : Alguns exemplos de integração de módulos de controle e aprendizagem: (a) Filtro de dados antes da aprendizagem. (b) Sistema de decisão suportado por computação sub-simbólica. (c) Aprendizagem por Reforço. (d) Selecção de um método de aprendizagem.

Vectores de Aprendizagem (L-Arrays)

A transferências de dados entre as diversas camadas de uma rede de aprendizagem é definida por estruturas vectoriais. Para evitar usar as pesadas estruturas vectoriais do NETDEF, são definidas outras mais específicas denominadas por vectores de aprendizagem. Um vector de aprendizagem (ou L-Array) consiste simplesmente numa sequência de neurónios capazes de guardar valores reais. Graficamente,

ing., Learning Arrays

fig. 5.4.3 : vector de aprendizagem A.

A diferença para os vectores do NETDEF, é que não é possível aceder dinamicamente a uma determinada posição do vector. Cada atribuição ou acesso é estático. Isso simplifica a estrutura de consulta e escrita, reduzindo-as a simples atribuições de variáveis (nesse sentido, um L-Array é apenas um conjunto de variáveis reais). Usaremos somente a expressão vector quando o vector do NETDEF não estiver em questão. Um vector de aprendizagem constante é definido usando a seguinte sintaxe:

l-const ::= "[" number { "," number } "]".

Por exemplo, a expressão [2.3, 4.0, 0.1] define um vector de três elementos com os valores constantes dados pelos elementos da lista na expressão. Dois vectores podem ser usados numa atribuição.

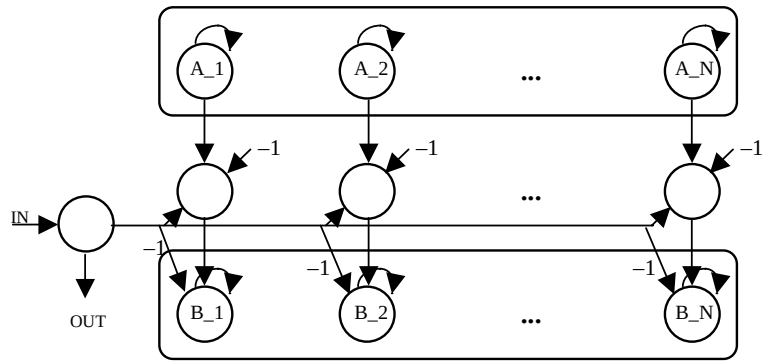


fig. 5.4.4 : rede da expressão B := A, onde A e B são vectores de aprendizagem.

Se na atribuição, os vectores possuírem dimensão diferente, dois casos podem acontecer: (a) o vector a copiar é maior, e nesse caso os valores em excesso não são copiados, e (b) o vector a ser copiado é maior, e nesse caso, os valores em excesso ficam a zero.

Um vector pode ser iniciado num valor aleatório em]0,x[, utilizando a expressão A := RANDR(N,x). Para a produção de números aleatórios, usando os canais de dados especiais denominados GUESSB e GUESSR, ler secção 5.9. A rede respectiva é dada por:

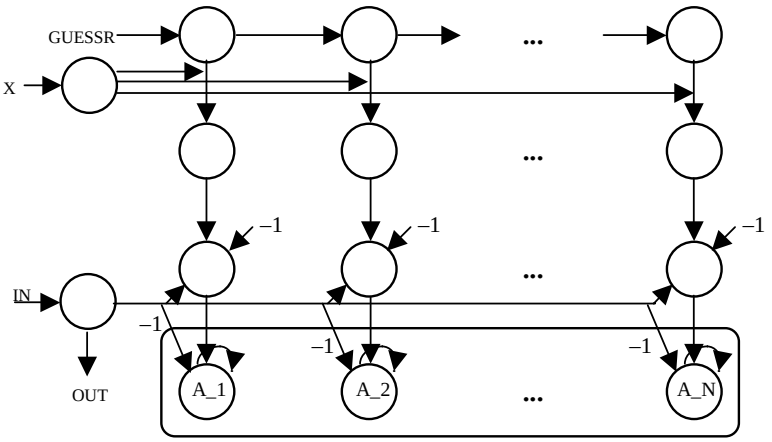


fig. 5.4.5 : rede da expressão A := RANDR(N,x).

A utilização destes vectores é realizada, como foi dito acima, de forma estática. Não é possível usar variáveis para aceder a uma posição do vector dinamicamente. Isto reflecte-se na sintaxe, por exemplo, a 3ª posição do vector

A é dada por A_3 e não por A[3]. Por exemplo, para um determinado vector A receber os dados dos canais de entrada IN_1 a IN_3, escreve-se assim:

Arrays – acesso dinâmico
L-Arrays – acesso estático

```
PAR
  RECEIVE A_1 FROM IN1;
  RECEIVE A_2 FROM IN2;
  RECEIVE A_3 FROM IN3;
ENDPAR
```

Sintaxe Geral das Funções de Aprendizagem

O primeiro passo para a construção de uma função de aprendizagem, é definir a sua estrutura interna e descrever quais os processos simbólicos que operarão sobre ela. Em termos sintácticos, teremos uma descrição deste tipo:

```
LFUNC F IS
  X_SIZE := N;
  Y_SIZE := M;
  ETA := 0.1;
  ...
  W_RULE := W + ETA*Y*D;
INIT
  P1;
FINAL
  P2;
ENDLFUNC;
```

A definição consiste em 3 partes distintas. A primeira parte descreve a estrutura da rede de aprendizagem (para redes não-recorrentes de uma camada): o número de neurónios da camada de entrada (X_SIZE) e da camada de saída (Y_SIZE); e ainda um conjunto de variáveis necessárias ao processo mais a descrição da regra de actualização (W_RULE e B_RULE, respectivamente para a actualização dos pesos sinápticos e dos pendores). A segunda parte diz respeito ao processo P1 que é executado cada vez que uma amostra entra para ser aprendida. Finalmente, o processo final P2 é executado no fim da actualização da rede para cada amostra.

Quando a descrição é compilada, é automaticamente definido um conjunto extra de estruturas necessárias para a correcta execução do algoritmo. Elas são:

- X e Y – vectores que guardam os dados de entrada e saída, respectivamente.
- W – vector que guarda os pesos sinápticos da rede. Ou seja, é em W que o conhecimento produzido é armazenado.

- D – vector que guarda os valores desejados de cada amostra. Esta estrutura serve nos casos de aprendizagem supervisionada. Possui a mesma dimensão do vector Y.
- B – vector que guarda os pendores de cada saída de Y.
- T – inteiro que guarda o número de vezes que a rede foi alterada desde a última inicialização.
- AVG – real que guarda a média das mudanças dos pesos sinápticos de W na última actualização.

A dinâmica de uma rede de aprendizagem é descrita pela seguinte figura:

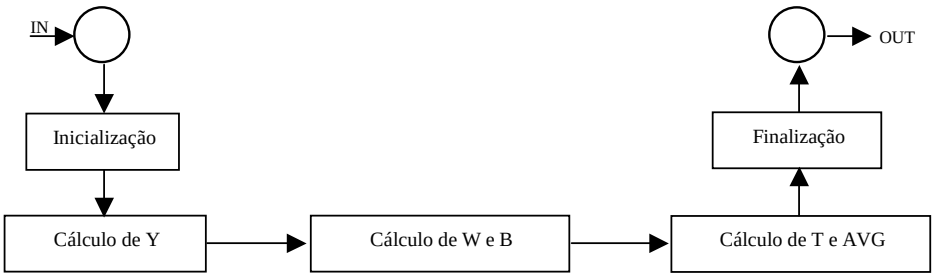


fig. 5.4.6 : Estrutura geral dos diferentes módulos de uma rede de aprendizagem.

Estas funções definem estruturas de manipulação e gestão da informação produzida pelas redes de aprendizagem. Sobre elas podem ser aplicadas leis de aprendizagem. Neste capítulo, estudaremos a integração da Lei de Hebb e da aprendizagem competitiva. Para distinguir entre estas duas arquitecturas neuronais (rede por camadas para aprendizagem Hebbiana, ou uma rede de neurónios para aprendizagem competitiva) introduzimos uma palavra chave no início da definição das funções de aprendizagem.

```
LFUNC F IS LAYERED
  ...
ENDLFUNC;
```

para aprendizagem Hebbiana, e:

```
LFUNC F IS COMPETITIVE
  ...
ENDLFUNC;
```

para aprendizagem Competitiva

Esta sintaxe permite esconder do programador a complexidade intrínseca da arquitectura neuronal que executa o processo de aprendizagem. Veremos em seguida a estrutura destas redes.

Processos de Inicialização, Aprendizagem e Classificação

O mecanismo de acesso e utilização destas funções de aprendizagem funciona da seguinte forma: Primeiro a rede é inicializada para em seguida aprender, para depois poder ser consultada.

O processo de inicialização passa por atribuir valores iniciais aos pesos sinápticos e aos pendores da rede de aprendizagem. Por exemplo, para inicializar o vector W (com N pesos sinápticos) com valores aleatórios entre 0 e 0.1, pode ser utilizado o seguinte código,

```
F_W := RANDR(N,0.1)
```

Se pretender inicializar com valores zero, pode-se usar a mesma função, desta forma:

```
F_W := RANDR(N,0)
```

Para os pendores (ou seja, para o vector B), o processo seria idêntico.

Para aprender, carregam-se as estruturas específicas (os vectores X e D se for supervisionada) com a informação apropriada e executa-se a função no modo de aprendizagem que é designado pelo comando LEARN. Por exemplo,

```
LOCK CALLF;
  PAR
    F_X := [1.0, 0.0, 0.0];
    F_D := [1.0];
  ENDPAR;
  LEARN F;
ENDLOCK;
```

Ou seja, a rede vai aprender o padrão [1,0,0] para obter o resultado [1]. Como a rede vai aprender esse padrão, depende exclusivamente do algoritmo de aprendizagem escolhido. É utilizada uma fechadura (*vide*. secção 4.5), para evitar problemas de acesso e rescrita sobre os vectores da função.

Por exemplo, para representar um algoritmo de aprendizagem iterativo, em que seria repetida a aprendizagem das amostras até obter um determinado critério, poder-se-ia usar um código como o seguinte:

```
REPEAT
  LOCK CALLF;
    PAR
      F_X := próxima amostra;
      F_D := classe de X;
    ENDPAR;
    LEARN F;
  ENDLOCK;
UNTIL critério;
```

O critério pode utilizar as estruturas AVG e T disponibilizadas pela rede, para facilitar a sua construção. Como exemplo, para um dado conjunto de N amostras, o processo continua até que se atinja uma convergência média na alteração dos pesos sinápticos, ou que um número máximo de P passos seja atingido (ou seja, tenha havido N*P chamadas do processo de aprendizagem LEARN):

```
REPEAT

  ...     -- processo a iterar

UNTIL ( F_T >= N*P )
   OR ( F_AVG < maxConvergência );
```

Para classificar, é utilizado o comando designado por CLASSIFY, usando o seguinte esquema:

```
LOCK CALLF;
  F_X := [1.0, 0.0, 0.0];
  CLASSIFY F;
  resultado := F_Y;
ENDLOCK;
```

Como o resultado da rede (em relação à entrada X) é guardada no vector Y, uma simples atribuição para um outro vector permite guardar esse resultado de forma persistente.

Antes de entrar nos pormenores das redes de aprendizagem, uma nota esclarecedora. As redes-σ^+ não aprendem os conceitos da forma que os algoritmos de aprendizagem comuns o fazem, ou seja, por alteração sináptica. Neste caso, a aprendizagem é codificada nas activações dos neurónios que realizam as conexões neuro-sinapticas, característica essencial do modelo. Estas activações podem ser mais ou menos douradoras, consoante a necessidade, mas é através deles que o conhecimento aprendido é conservado.

5.5. Redes de uma camada

Um determinado conjunto de regras de aprendizagem utiliza a topologia das redes por camadas. Apresentaremos nesta secção, a estrutura geral utilizada por algoritmos, como o de Hebb ou de Widrow-Hoff.

Aprendizagem

Começaremos por mostrar a estrutura neuronal inicial, ou seja, aquela que recebe um sinal do exterior para iniciar a sua execução. Nesta rede de uma camada, existem N entradas e K saídas. Logo, a dimensão do vector X é de N, a dimensão dos vectores Y e B é de K, e a dimensão do vector W é de NK.

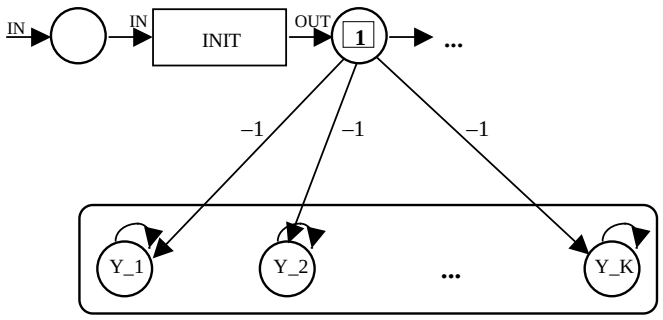


fig. 5.5.1 : Entrada do pedido de aprendizagem na função F.

Para além de executar o módulo de inicialização, no qual poderão ser executadas várias manipulações, tanto das variáveis internas como das chamadas a outros processos externos relacionados, a rede apaga a activação dos neurónios resultado de modo que quando os novos resultados forem calculados, estes possam ser guardados sem problemas. A partir daí, dados os valores de entrada X, os pesos sinápticos W, e os pendores B já estão disponíveis (o programador teve de ter essa preocupação antes de activar a função para aprender) a rede de controle pode executar a rede de camadas, obtendo o valor do vector de resultado Y.

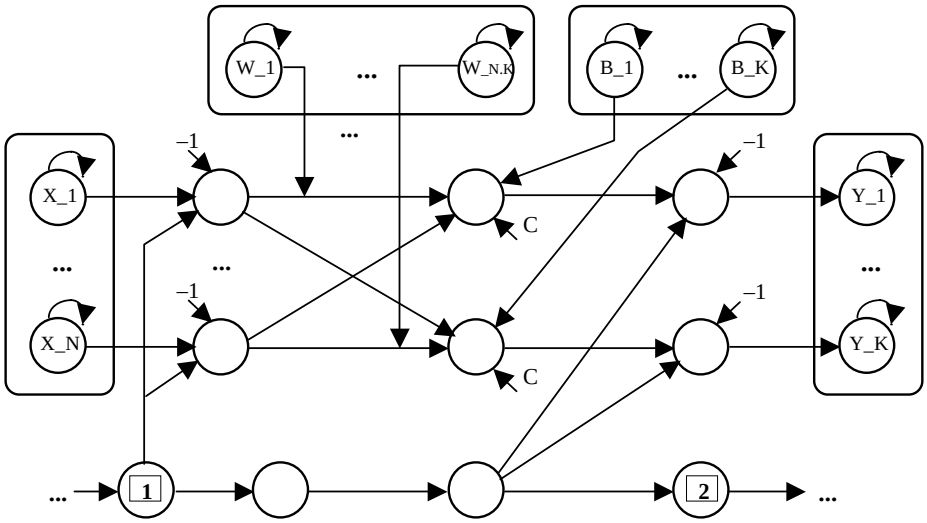


fig. 5.5.2 : Cálculo do vector Y a partir da informação de entrada (X) e do conhecimento adquirido previamente (W e B).

Os neurónios que recebem a soma ponderada das entradas pelos valores de W, mais a soma dos pendores respectivos, têm um pendor para compensar um excesso de valor referente ao processo de codificação. Denominado por C na figura, este valor tem de diminuir o somatório de N+1 parcelas (as N entradas mais o pendor), i.e., como uma soma binária de valores codificados acrescenta 0.5, N+1 parcelas acrescentam 0.5*N. Assim, o valor de C é igual a –N/2.

Em relação ao número de neurónios utilizados por estas duas primeiras fases, temos para a 1ª fase, somente dois neurónios (excluindo a rede da inicialização, cuja dimensão depende da complexidade da computação exigida do módulo), enquanto a 2ª fase necessita de N+2K+3 neurónios. Isto representa um total de N+2K+5 neurónios necessários antes da actualização dos pesos e pendores, ou seja, da aprendizagem propriamente dita. Uma vez calculado Y, é necessário actualizar os pesos sinápticos W e os pendores B, computando a fórmula existente na definição da função de aprendizagem. É executada, em paralelo, uma rede expressão que calcula o valor dessa fórmula para cada instância dos pesos e dos pendores.

É esta a parte do processo de aprendizagem que consome mais recursos. No entanto, o interesse da vantagem compensa esse problema: a velocidade de execução é independente da dimensão do problema!

O próximo passo, como já foi dito, é a actualização do conhecimento da rede de aprendizagem. As redes seguintes mostram os esquemas padrão para alteração de um peso e de um pendor arbitrários.

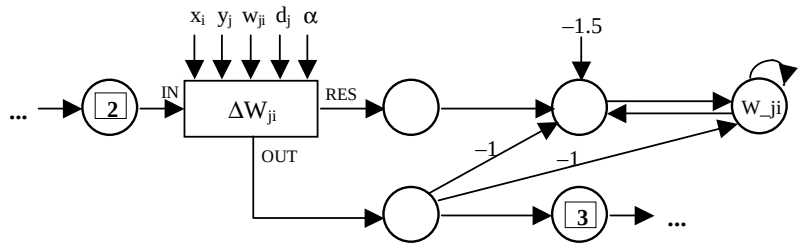


fig. 5.5.3 : Cálculo da mudança do peso sináptico Wji e actualização da respectiva posição de memória.

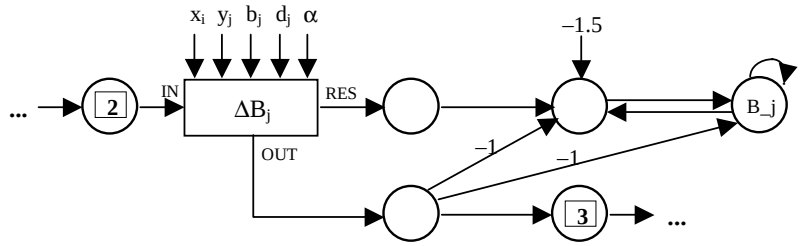


fig. 5.5.4 : Cálculo da mudança do pendor Bj e actualização da respectiva posição de memória.

A dimensão destas duas redes (excluindo a parte dos módulos de cálculo das diferenças que depende do algoritmo) é de 4(K+NK) neurónios.

Uma vez actualizados os pesos e os pendores, é necessário alterar os valores das variáveis internas, para manter a coerência da estrutura neuronal da função de aprendizagem. Este conjunto de variáveis é: T que representa o número de amostras aprendidas pela rede desde a última inicialização e AVG que representa a média das alterações dos pesos sinápticos da última amostra aprendida.

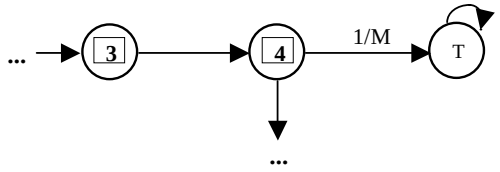


fig. 5.5.5 : Cálculo da variável interna T (número de aprendizagens executadas desde a última inicialização).

Assim, para actualizar a variável T, basta incrementar uma unidade à sua activação. A codificação utilizada neste caso, é a dos inteiros. Isso justifica o uso do peso sináptico 1/M como apresentado no sucessor da figura 4.3.5(d). De notar, que o neurónio de ligação 3 da figura anterior e também da próxima, basta vir de só uma das redes de actualização de pesos, pois a expressão para o cálculo dos pesos é a mesma para cada neurónio, significando isso, que todos esses módulos terminam a sua execução no mesmo instante (logo não é necessário nenhum processo de sincronização de resultados).

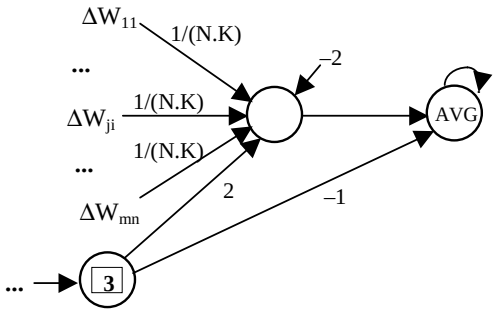


fig. 5.5.6 : Cálculo da variável interna AVG (média das alterações sinápticas ocorridas).

A forma de calcular a rede para a média é dada pela seguinte expressão:

$$\alpha_R\left(\left[\sum_n \sum_k \alpha_R^{-1}(w_{kn}) \right] / (NK) \right) = \ldots =$$

$$\sum_n \sum_k \left(w_{kn} / (NK) \right)$$

O número de neurónios destas duas redes é de somente quatro neurónios (dois dos quais para guardar as actividades de T e AVG), dado que as ligações aos N.K resultados podem ser realizados directamente dos canais de resultado das

suas redes respectivas. Finalmente, a rede que executa o processo de encerramento da aprendizagem de uma dada amostra.

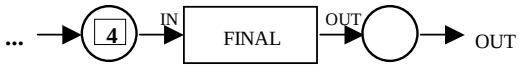


fig. 5.5.7 : Rede de finalização e envio de sinal para o canal de sincronização de saída.

Somando os neurónios de todas as redes apresentadas, não contando com os neurónios dos módulos, obtemos um total de 5NK + 9K + 2N + 8, dos quais NK + N + 3K + 2 são os neurónios que representam os vectores X, Y, W, D e B e as variáveis T e AVG da função de aprendizagem. Ou seja, a complexidade em termos dos neurónios necessários pertence ao conjunto Θ(NK), enquanto a complexidade temporal do processo de aprendizagem é constante.

Classificação

A estrutura do processo de classificação é mais simples. A única tarefa necessária passa por processar o vector dado X, e guardar o resultado da aplicação matricial de W em X, no vector Y.

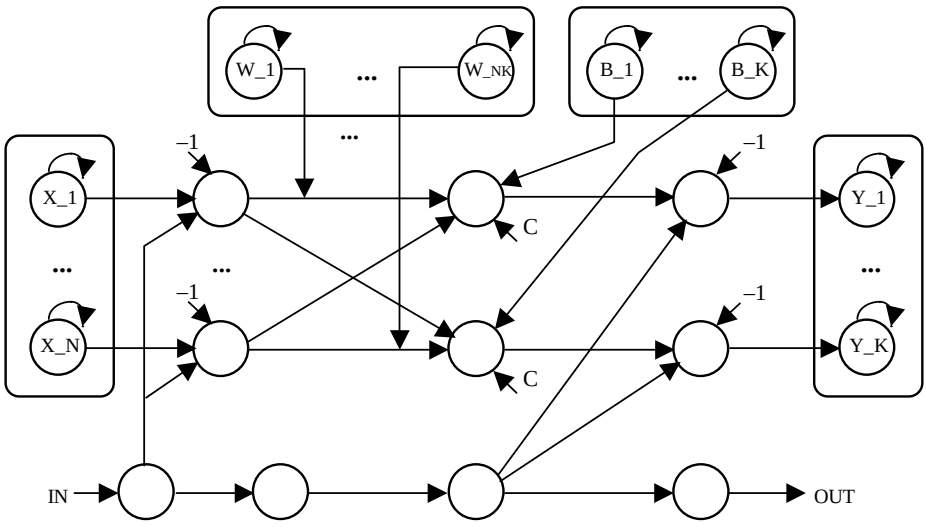


fig. 5.5.8 : Classificação da informação de entrada X.

Para isso, é possível usar a estrutura apresentada na parte da aprendizagem, tendo de incluir somente um novo conjunto de neurónios que gerem o processo de sincronização da tarefa da classificação.

5.6. Lei de Hebb

Em 1949, Donald Hebb apresentou a seguinte descrição do que seria um modelo explicativo para o comportamento, ao nível sináptico:

> *"When an axon of cell A is near enough to excite cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."* [Hebb 49], página 50.

Esta afirmação, não sendo um modelo matemático, foi suficiente explicita para a comunidade científica construir modelos de aprendizagem baseados nela. A aprendizagem baseada na ideia de Hebb diz-nos que na sinapse w_{ji}, que conecta a entrada u_i ao neurónio x_j, a variação sináptica é dada pela seguinte expressão:

$$\Delta w_{ju_i} = \eta \, u_i \, y_j \qquad (27)$$

sendo η uma constante de aprendizagem entre]0,1], necessária ao processo de estabilização do sistema, com vista à convergência dos pesos finais.

Esta regra tem um problema, que também surge na formulação inicial da lei, que é de sempre fortalecer a ligação sináptica, não havendo uma forma de diminuição do respectivo peso. Uma outra variante da lei, focando neste ponto, assume papeis assimétricos aos papeis da entrada e da saída. Ela é designada por lei de Oja (cf. [Oja 82]), e é descrita pela seguinte expressão:

$$\Delta w_{ju_i} = \eta \, (u_i \, y_j - y_j^2 \, w_{ju_i}) \qquad (28)$$

Esta lei também pode ser aplicada em casos não-supervisionados, considerando o y_j como a resposta linear do neurónio x_j à amostra dada, i.e.,

$$y_j = \sum_{i=1}^{N} w_{ju_i} u_i$$

Usamos na seguinte codificação a regra de Oja.

```
LFUNC F IS LAYERED
  X_SIZE := N;
  Y_SIZE := M;
  ETA    := 0.1;
  W_RULE := W + ETA * (X*D – D*D*W);
  B_RULE := B + ETA * (D – D*D*B);
ENDLFUNC;
```

para aprendizagem supervisionada (i.e., quando é fornecido previamente o vector D das respostas desejadas), e:

```
LFUNC F IS LAYERED
  X_SIZE := N;
  Y_SIZE := M;
  ETA    := 0.1;
  W_RULE := W + ETA * (X*Y – Y*Y*W);
  B_RULE := B + ETA * (Y – Y*Y*B);
ENDLFUNC;
```

para aprendizagem não supervisionada (i.e., quando não se sabe o vector D das respostas desejadas).

A estrutura (simplificada aqui por motivos de legibilidade) do módulo que calcula a diferença do peso sináptico para o caso supervisionado, é dada na figura 5.6.1.

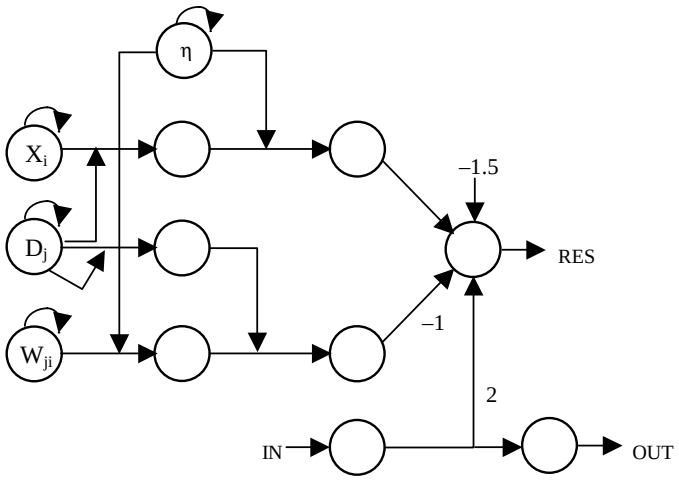


fig. 5.6.1 : Rede para cálculo do ΔW_{ji} na aprendizagem Hebbiana supervisionada.

A rede que efectua o calculo do delta para a regra não supervisionada, tem estrutura semelhante, mas em vez de receber o resultado desejado D_j, recebe o resultado Y_j efectivamente obtido pela rede.

Para a obtenção da variação do pendor, a rede é ligeiramente mais simples, dado não ser necessário recorrer a nenhum elemento do vector X.

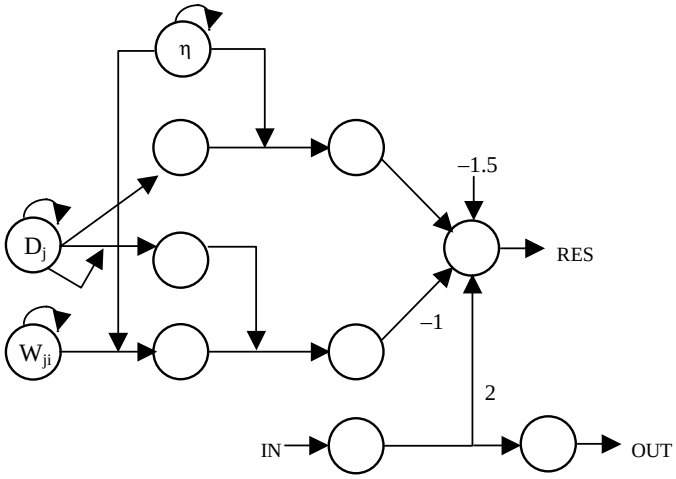


fig. 5.6.2 : Rede para cálculo do ΔB_j na aprendizagem Hebbiana supervisionada.

Em termos de neurónios, cada cálculo de um peso sináptico (e também para os pendores) requer 8 neurónios (neste caso da rede optimizada, que pode ser definido como padrão pelo processo de compilação, para redes utilizando esta aprendizagem Hebbiana). Já o tempo de execução requer um atraso de somente duas unidades de tempo.

5.7. Lei de Widrow-Hoff

O modelo proposto inicialmente por Bernard Widrow e Marcian Hoff (cf. [Widrow e Hoff 60]), e denominado por ADALINE, é basicamente uma variante do modelo de perceptrão, que procura solucionar alguns dos problemas deste último. A função de saída do neurónio é simplesmente a aplicação linear das suas entradas,

$$y = \sum_{i=1}^{M} w_{u_i} u_i - c \qquad (29)$$

A regra tende a minimizar o erro E entre a diferença de *y* (o valor computado) e *d* (o valor desejado), i.e., $E(y,d) = (y - d)^2$.

A variação sináptica do peso w_i é dada pela expressão:

$$\Delta w_{u_i} = \eta \, (d - y) \, u_i \qquad (30)$$

onde y é a saída actual do neurónio, dado pela resposta linear do vector de entrada com os pesos respectivos menos o pendor, aplicando a função de activação ϕ_H dada pela função (2) descrita na secção 2.2,

$$y = \phi_H(\sum_{i=1}^{M} w_{u_i} u_i - c) \qquad (31)$$

Por isso, esta lei também é conhecida por lei *LMS*. Este tipo de lei de aprendizagem é designada por lei de *descida do gradiente*, na medida que cada correcção efectuada pela rede (procurando melhorar o seu desempenho) é realizada através de uma descida ao longo do gradiente da superfície definida pela função de erro. A influência, por menor que seja a diferença entre y e d, é levada em conta (dado que a função degrau não é aplicada, não se perdendo uma possível informação relevante).

ing., Least Mean Square

A característica desta lei de aprendizagem é a convergência para o mínimo da função erro. No entanto, não é garantida a separação de todas as classes linearmente separáveis.

A regra de aprendizagem de Widrow-Hoff é calculada pela seguinte descrição:

```
LFUNC F IS LAYERED
  X_SIZE := N;
  Y_SIZE := M;
  ETA    := 0.1;
  W_RULE := W + ETA * (D - Y) * X;
  B_RULE := B + ETA * (D - Y);
ENDLFUNC;
```

A estrutura (simplificada por motivos de legibilidade) do módulo que calcula a diferença do peso sináptico para a regra de Widrow-Hoff, é dada na seguinte figura.

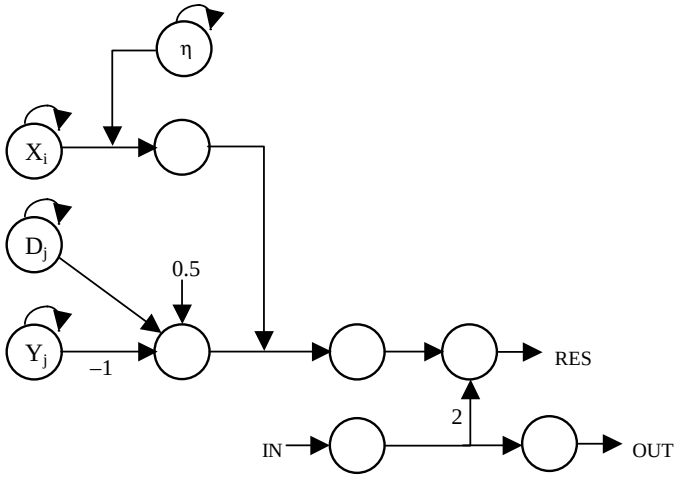


fig. 5.7.1 : Rede para cálculo do ΔW_{ji} na aprendizagem de Widrow-Hoff.

Para o cálculo da variação do pendor, a única alteração em relação à ultima rede apresentada, seria a ligação directa do neurónio da taxa de aprendizagem ao neurónio que devolve a diferença ($Y_j - D_j$) através de uma sinapse neuro-sináptica.

5.8. Aprendizagem Competitiva

Na aprendizagem competitiva, dado um vector de entrada, existe um conjunto de neurónios (conectado sinapticamente a esse vector de entrada) que vai ser activado ao mesmo tempo. Dessas activações, haverá uma maior que as outras que fará com que o respectivo neurónio seja declarado o "vencedor", e ele e apenas ele terá as suas ligações sinápticas alteradas segundo uma regra de adaptação bem definida.

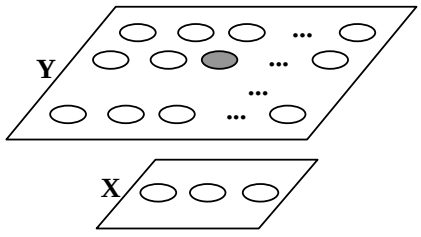


fig. 5.8.1 : Estrutura neuronal na aprendizagem competitiva.

Na figura, a camada inferior de neurónios (no nosso caso, o vector X) recebe a informação proveniente dos canais de entrada e envia esses valores para todos

os neurónios da camada superior (vector Y), ponderados pelos respectivos pesos sinápticos (vector W). O maior é seleccionado (representado na figura pela coloração cinzenta).

Aprendizagem

Seja um dado vector de entrada, $\vec{u} = (u_1, ..., u_M)$. Existe um neurónio x_k (em caso de empate, escolher um ao acaso) que possuí a maior actividade, i.e.,

$$\sum_{i=1}^{M} w_{ku_i} u_i \geq \sum_{i=1}^{M} w_{ju_i} u_i \ , \ \forall_{j \neq k}$$

Se a amostra estiver a ser classificada, a rede neuronal informa que lhe atribuiu a classe representada pelo neurónio x_k. Se, por outro lado, a rede estiver em processo de aprendizagem, os pesos sinápticos w_{ku_i} são modificados segundo a expressão seguinte[16] (os pesos dos outros neurónios são mantidos inalterados),

$$\Delta w_{ku_i} = \eta \ (u_i - w_{ku_i} \ ) \qquad (32)$$

ing., Self-Organizing Map

No algoritmo SOM (baseado nesta lei) de Teuvo Kohonen (cf. [Kohonen 87, 97]), os neurónios vizinhos do neurónio escolhido têm também os seus pesos alterados com a mesma regra, mas com um coeficiente η menor (tanto menor quanto mais distante o vizinho). O objectivo é correlacionar neurónios vizinhos, com classes estruturalmente próximas no espaço de características definido pelo conjunto das amostras apresentadas à rede. Deste modo, aproxima-se a estrutura da resolução, fornecida pela rede neuronal, da própria estrutura do problema. Como resultado da aplicação desta lei, o vector sináptico de cada neurónio ajusta-se ao centro de gravidade de uma região de decisão, identificando cada neurónio com uma certa classe de amostras (sendo as classes definidas pelo próprio processo de aprendizagem, sendo por isso, um algoritmo não supervisionado). No entanto, a rede apresentada nesta secção não implementa o SOM, mas simplesmente o algoritmo competitivo padrão.

O primeiro passo é calcular todas as activações da camada de neurónios que recebe da informação de entrada. De seguida, essas activações passam por um processo de selecção para detectar qual delas possui a activação máxima, a qual será usada para alterar os pesos desse neurónio vencedor.

Nesta secção apresentaremos apenas as redes da estrutura total que apresentam diferenças em relação às redes apresentadas anteriormente. Deste modo, tanto a rede inicial apresentada na figura 5.5.1, as redes de actualização das variáveis internas das figuras 5.5.5 e 5.5.6, e a rede que executa os procedimentos finais da figura 5.5.7, são idênticas em ambos os casos.

Da mesma forma que foi efectuado nas secções anteriores, também aqui as computações serão executadas em paralelo. Apesar da estrutura neuronal poder ser complexa, o processamento mantém a sua propriedade de ser executado em tempo constante.

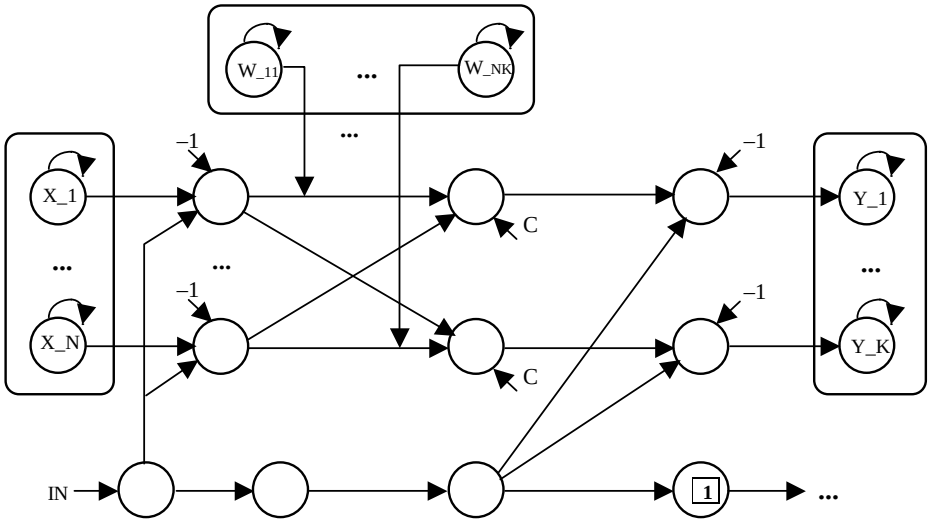


fig. 5.8.2 : Calculo das activações neuronais dado o vector X.

O valor C descrito na estrutura, tem o valor necessário para compensar a soma simultânea de N valores (igual à dimensão do vector de entrada X), ou seja C = – 0.5*(N–1).

---

[16] Em alguns algoritmos baseados nesta lei, depois da aprendizagem de uma amostra, existe a normalização dos pesos sinápticos do neurónio escolhido de modo a que sua soma seja 1.

A rede seguinte mostra o esquema geral de como encontrar o valor máximo de activação, dados os novos valores guardados em Y. Apresentaremos apenas a rede que envia o sinal 1 pelo canal de saída de resultado, se um dado neurónio Y_j obteve a maior das activações, e 0 caso contrário. Cada neurónio do vector Y tem associado uma rede destas.

Nesta rede, a activação de Y_j é comparada com a de cada um dos outros neurónios do vector Y (excluindo ele próprio). Se a sua activação não for a maior, haverá restos da subtracção em alguma das activações, que será magnificada pela multiplicação com o majorante M. Assim, essa multiplicação anulará o valor 1 transportado pelo sinal de sincronização, e a saída da rede será zero.

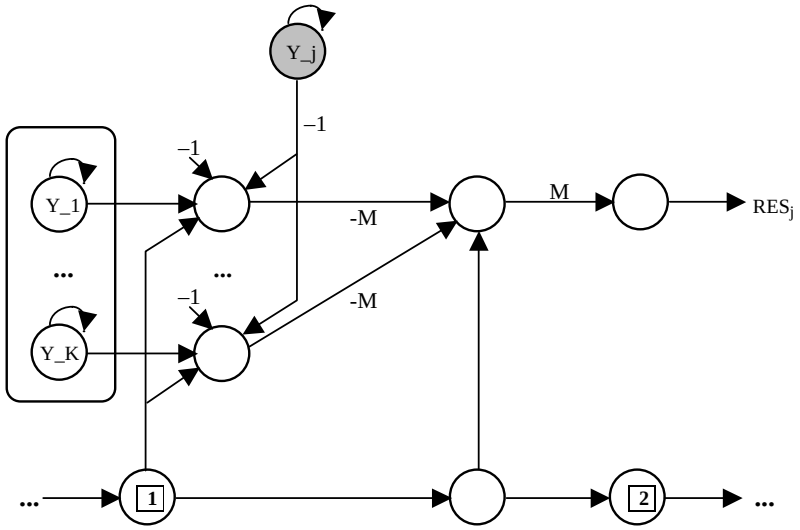


fig. 5.8.3 : Rede que detecta se o neurónio Y_j é o de maior activação.

No outro caso, ou seja, se o valor de Y_j for realmente o maior, ele eliminará na subtracção todas as outras actividades que não afectarão a passagem do valor da sincronização. Assim, ele passará pelo canal de resultado, informando que é este o neurónio vencedor. Tendo Y dimensão K, esta rede que detecta individualmente qual dos neurónios é o máximo, necessita de 4K+2 neurónios, já que os dois neurónios de sincronização são partilhados por todos.

Em princípio não haverá situações de igualdade, dado que o vector W é iniciado com valores aleatórios, que excluem à partida essa situação. No

entanto, se isso acontecer, a rede simplesmente não aprende essa amostra naquela iteração do processo de aprendizagem.

O próximo passo é actualizar apenas os pesos sinápticos de W_{ji}, que conectam as entradas X_i com o neurónio vencedor Y_j. Já aqui não é necessário criar múltiplas redes idênticas para cada Y_j, dado que a actualização será efectuada apenas para um dos neurónios de Y. É possível que todos eles partilhem a mesma estrutura que executa a lei de aprendizagem, como veremos nas redes seguintes:

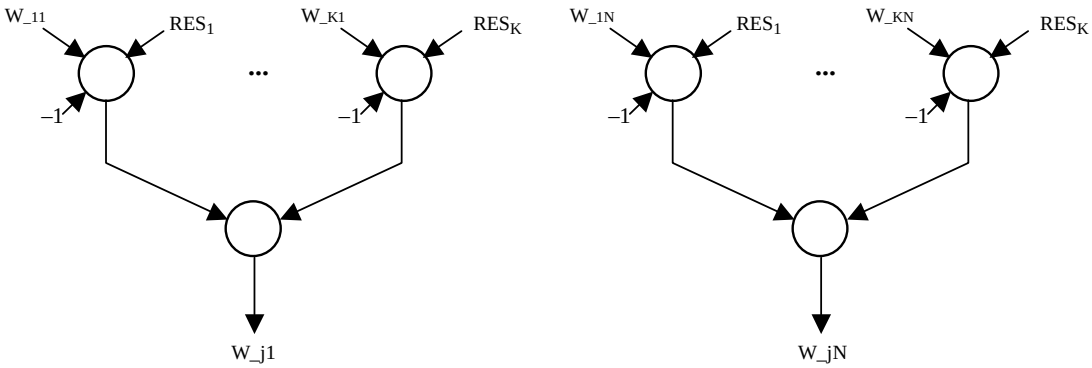


fig. 5.8.4 : Rede para obtenção dos pesos W a modificar, relativas ao neurónio vencedor Y_j.

Uma vez seleccionados quais os pesos que serão alterados, executa-se a rede que calcula $\Delta w_{ki} = \eta (u_i - w_{ki})$.

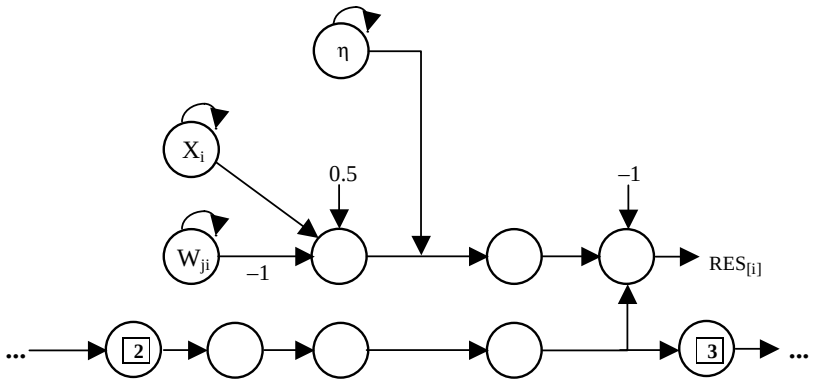


fig. 5.8.5 : Rede que calcula ΔW_{ji}.

Calculadas as N variações dos pesos, ΔW_{j1} a ΔW_{jn}, é necessário actualizar os respectivos valores. Para isso, é usado um esquema semelhante ao utilizado na rede apresentada na figura 5.8.4. Os valores das saídas RES_j servem para abrir apenas os canais respectivos, dado que apenas o neurónio vencedor foi activado com o valor 1.

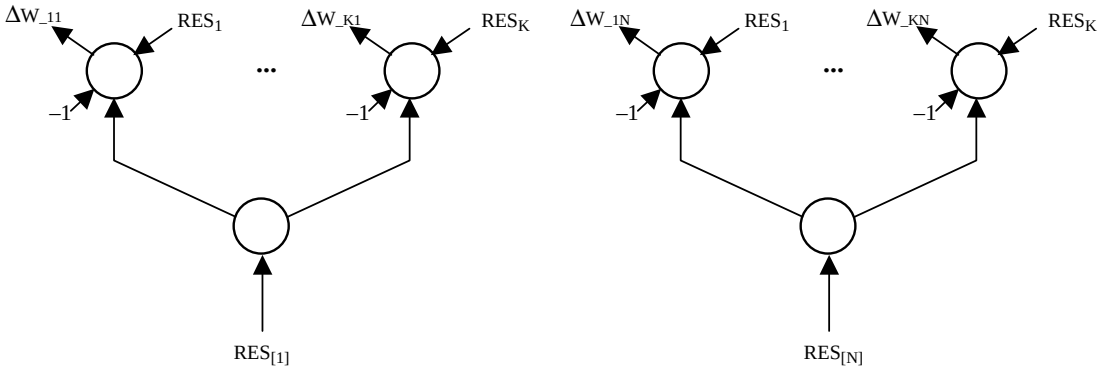


fig. 5.8.6 : Actualização dos pesos W_{j1}, ..., W_{jn}.

Esta rede é apresentada na sua forma simplificada, eliminando os pormenores necessários para adicionar a variação ao valor da iteração anterior do peso sináptico, bem como a sincronização de RES_i e $RES_{[i]}$. A dimensão total destas três redes é (K+1)N, 7N e (K+1)N neurónios respectivamente, totalizando 2N(K+1)+7N neurónios na estrutura de adaptação sináptica.

Uma das extensões do algoritmo competitivo atribui ao coeficiente de aprendizagem η um valor dependente do número de iterações já efectuadas. A justificação para este procedimento, tem a ver com o facto de uma rede cuja aprendizagem tenha começado há pouco seja mais flexível e maleável do que uma rede cujo processo de aprendizagem esteja bem avançado. Assim, η deixa de ser uma constante e passa a ser uma função do número de iterações. A fórmula utilizada aqui é dada pela seguinte definição da função $\eta(t)$:

$$\eta(t) = 0.1 - 0.00001\ t$$

Isso poderia ser feita por programação, usando um código como o seguinte:

```
LFUNC F IS COMPETITIVE
  ...
  ALPHA := 0.1;
  W_RULE := W + ALPHA * (D – Y) * X;
  ...
FINAL
  ALPHA := 0.1 – T * 0.00001;
ENDLFUNC;
```

Classificação

O processo de classificação é semelhante ao apresentado para as redes de uma camada. A estrutura utilizada é a mesma: aplicar os valores da amostra a classificar sobre o vector X, ponderar esses valores com os valores de W, e obter o resultado em Y.

A diferença, neste método, é que os neurónios em Y vão devolver activações proporcionais da sua representatividade em relação à amostra dada. O tratamento dessa informação depende do problema a resolver e deve ser programado depois da chamada de classificação sobre a rede de aprendizagem competitiva. Por exemplo, seja F a função de aprendizagem competitiva utilizada, com um vector X de 3 entradas, e um vector Y de saída de 6 saídas (simulando uma matriz de resposta de dimensão 2x3):

```
LOCK CALLF;
  F_X := [1.0, 0.0, 0.0];
  CLASSIFY F;
  resultado := F_Y;
ENDLOCK;
```

Suponhamos que o resultado do vector de saída Y é igual a [0.432, 0.0, 0.03, 0.86, 0.0, 0.0]. Se quisermos obter o neurónio vencedor, bastaria percorrer o vector resposta para obter o máximo e determinar qual a sua posição na matriz (o 4^{o} elemento representaria o elemento $Y_{2,1}$ da matriz de resposta). Outros requerimentos seriam igualmente programados através do NETDEF.

5.9. Utilização de sistemas caóticos

É possível aproveitar características caóticas de certas funções não lineares. A ideia desta secção é aproveitar algumas das características destes sistemas para desenvolver ferramentas para a linguagem NETDEF+.

Apesar de não existir ainda um consenso na comunidade científica sobre como caracterizar um sistema dinâmico como caótico, vamos utilizar uma definição dada em [Devaney 95]. Mas primeiro alguns conceitos:

órbita
ponto periódico
ponto fixo

Definição 5.9.1: Seja f : A→A uma aplicação no conjunto real A. A *órbita* de x, denotada por O(x), é o conjunto de pontos x, f(x), $f^2(x)$, $f^3(x)$, ... Um ponto x é *periódico* (ou de período n), se existe um n≥0 tal que $x=f^n(x)$; neste caso, diz-se que a órbita é periódica com período n. Um ponto x é um *ponto fixo* se x=f(x). ◁

topologicamente transitiva

Definição 5.9.2: Seja f : A→A uma aplicação no conjunto real A. A função f diz-se *topologicamente transitiva* se para qualquer par de conjuntos abertos U,V⊂A, existe um k>0 tal que $f^k(U)\cap V\neq\emptyset$. ◁

ponto limite
conjunto aberto
conjunto fechado
fecho
conj. denso

Definição 5.9.3: Seja os conjuntos reais B ⊂ A. Um ponto x∈A é um *ponto limite* de B, se existe uma sequência de pontos $x_n\in B$ que se aproximam arbitrariamente de x. B é *fechado* se contém todos os seus pontos limite, caso contrário, B é *aberto*. O *fecho* de um conjunto B, denotado por $\overline{B}$, é a união de B com todos os seus pontos limite. O subconjunto B é *denso* em A, se e só se, $\overline{B} = A$. ◁

Observação 5.9.4: Funções topologicamente transitivas possuem órbitas com pontos que se movem numa vizinhança arbitrariamente pequena de qualquer outra. Logo, os sistemas dinâmicos definido por elas não podem ser decompostos em dois conjuntos disjuntos independentes. Uma função topologicamente transitiva possui uma órbita densa no seu domínio (sendo que o inverso também se verifica, cf. [Devaney 95]).

sensibilidade às condições iniciais

Definição 5.9.5: Seja f : A→A uma aplicação no conjunto real A. A função f possui *sensibilidade às condições iniciais* se existe δ>0, tal que, para qualquer x∈A e para qualquer vizinhança B de x, existe um y∈B e k≥0, tal que, $|f^k(x) - f^k(y)|>\delta$. ◁

Observação 5.9.6: Uma função com esta característica possui para cada ponto x, um outro ponto y numa vizinhança arbitrariamente próxima (não necessariamente todos os pontos da vizinhança!) que se afastará pelo menos δ ao fim de um número finito de iterações. Estas funções limitam à partida o seu cálculo computacional, dado que mesmo o mais pequeno erro é ampliado pela iteração da função. Assim, a órbita computada poderá não ser semelhante à órbita real.

Chegamos assim à definição de Devaney para um sistema caótico unário:

função caótica

Definição 5.9.7: Seja $f : A \rightarrow A$ um aplicação no conjunto real A. A função f é *caótica* se:

- possui sensibilidade às condições iniciais;
- é topologicamente transitiva;
- se os seus pontos periódicos são densos em A. ◁

Este tipo de função unária caótica tem três ingredientes: não é previsível, não pode ser decomposta em subsistemas mais simples e possui um ingrediente de regularidade (dado pelos seus pontos periódicos) que é denso no domínio da função.

família das funções quadráticas $F_\mu(x) = \mu x(1-x)$

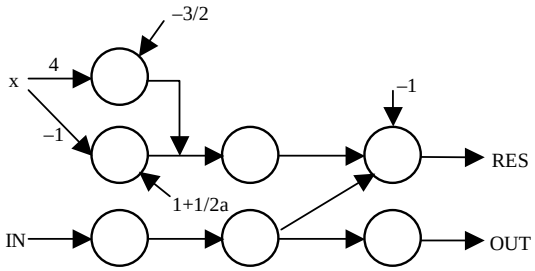
Exemplo 5.9.8: A função $F_4(x) = 4x(1–x)$ é caótica no intervalo [0,1]. A demonstração pode ser encontrada em [Devaney 95].

Vamos utilizar a função F_4 na construção de processos de busca e na criação de números pseudo-aleatórios. Antes de entrar nos pormenores mais específicos de cada ferramenta, vamos construir a rede neuronal capaz de computar F_4. Dividimos a função em duas parcelas, $f(x) = 4x$ e $g(x) = (1–x)$ para multiplicação posterior. As funções a serem computadas são dadas pelas composições $\alpha_R(f(\alpha_R^{-1}(x)))$ e $\alpha_R(g(\alpha_R^{-1}(x)))$.

$\alpha_R = (x+a) / 2a$ *para* $R = [-a, a]$

$$\alpha_R(f(\alpha_R^{-1}(x))) = 4x – 3/2$$

$$\alpha_R(g(\alpha_R^{-1}(x))) = – x + (1 + 1/2a)$$



cálculo de $F_4(x)$

fig. 5.9.9 : Estrutura da função $F_4(x) = 4x(1-x)$.

Do mesmo modo que na rede da multiplicação, a última camada de neurónios (nomeadamente o pendor –1) é necessária para evitar que a rede envie sinais não nulos em momentos de inactividade.

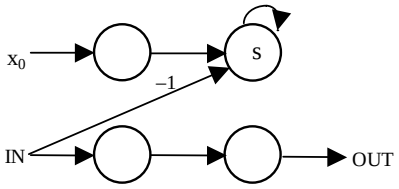
Pesquisadores

Sendo F_4 topologicamente transitiva, é garantida a existência de pelo menos uma órbita densa sobre [0,1], ou seja, uma órbita que passa arbitrariamente próxima de qualquer ponto do intervalo. Seja um sistema dinâmico doravante designado sistema pesquisador, definido pela seguinte equação:

$$s(t) = F_4^{t+1}(x_0) \qquad (33)$$

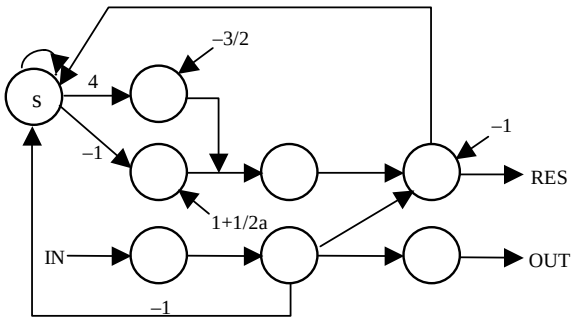
A trajectória do sistema s descreve as diversas composições da função F_4.

O objectivo deste sistema é pesquisar no intervalo [0,1] um determinado valor que satisfaça uma certa condição, quando nada mais se conhece sobre o processo de procura (seja de optimização, decifração ou outro qualquer). O sistema não é eficiente, mas tende a descobrir um valor aproximado dentro do contínuo dos valores reais (o que à partida não é óbvio, partindo da hipótese inicial de nada se conhecer sobre a estrutura do problema). Para poder aproveitar este mecanismo, são acrescentados à linguagem, dois novos processos: SEED e NEXT. O processo SEED introduz directamente um valor no módulo pesquisa, servindo principalmente para iniciar a iteração do módulo. O processo NEXT itera a função, calcula, armazena e devolve o novo valor calculado.



módulo para inserção da valor inicial da órbita

fig. 5.9.10 : Processo SEED(s, x_0).



módulo para cálculo do novo valor da órbita

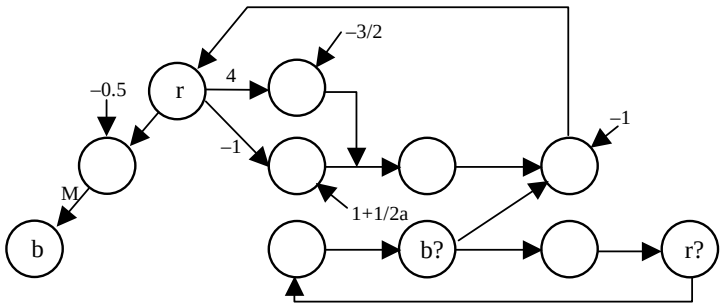
fig. 5.9.11 : Função NEXT(s).

Foram realizadas experiências com 200 mil pontos dentro do intervalo]0,1[. A vizinhança de cada ponto foi visitada pela função, pelo menos uma vez, durante uma iteração de um milhão de passos. A demora média para a função passar na vizinhança de um ponto foi cerca de 119 mil iterações. Esta forma de pesquisa de valores é cerca de 19% mais lenta, se comparada com os casos em que conhecemos a dimensão da partição na qual o valor desejado se encontra (onde é suficiente uma pesquisa linear pelas partições existentes). No entanto, este tipo de pesquisa é mais geral, dado não ser necessário conhecer previamente a partição de busca.

Números Aleatórios

A função F_4 dá-nos igualmente um processo de obtenção de sequências de números aleatórios[17]. O módulo correspondente será responsável pela aplicação constante da função, disponibilizando o último número computado numa variável específica.

[17] Ou pseudo-aleatórios para ser exacto, na medida que os métodos usados para a construção de tais números são através de fórmulas matemáticas iteradas, existindo sempre uma correlação na sequência dos números produzidos.



módulo produtor de sequências reais e booleanas aleatórias

fig. 5.9.12 : Módulo produtor da sequência de números aleatórios.

Este módulo possui dois canais de resultados que devolvem um real em]0,1[e um valor booleano (representado pelos valores de activação 0 e 1), respectivamente os neurónios designados por ‘r’ e ‘b’. Os neurónios ‘r?’ e ‘b?’ indicam quando esses valores estão disponíveis. É por isso que se torna necessário o mecanismo de espera bloqueante dos seguintes módulos:

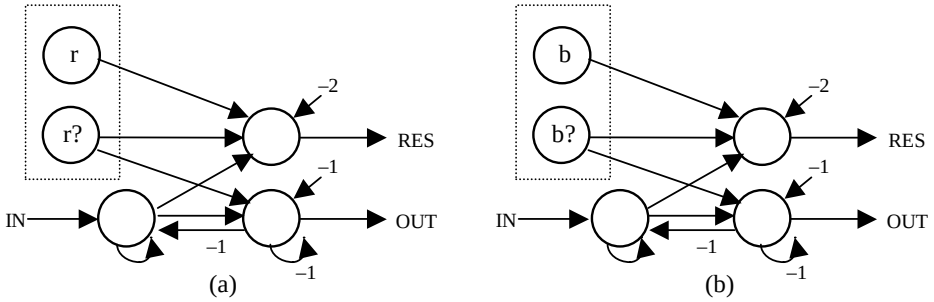


fig. 5.9.13 : Processos aleatórios: (a) RANDR e (b) RANDB.

Para aceder a estes valores, existem dois processos, RANDR e RANDB, que devolvem, respectivamente, um valor real aleatório entre [0,1] e um valor booleano aleatório. O canal booleano é utilizado para fornecer informação ao canal GUESS do processo CHOOSE para simular a escolha não determinista de um dos seus processos.

RANDR ∈]0,1[
RANDB ∈ {0,1}

O módulo produtor da sequência aleatória calcula um novo número a cada 4 iterações da rede. Para obter uma eficiência óptima, ou seja, um novo valor aleatório a cada instante, é possível construir 4 diferentes geradores, desfasados uniformemente pelo intervalo de 4 iterações para obter o desejado. Além disso, existe a vantagem de o gerador ser menos regular, dado ser o resultado de 4 iterações paralelas não relacionadas. Para isso ser possível, definem-se quatro pontos iniciais $x_{0,i}$ e ao i-ésimo gerador é dada a semente $f(x_{0,i})$. Os quatro

geradores são acoplados directamente ao neurónio denominado GUESSR que serve como um canal que disponibiliza a sequência de números aleatórios em]0,1[. É fácil, a partir de GUESSR, criar também um canal booleano designado por GUESSB. Este canal disponibiliza uma sequência aleatória de valores booleanos, ou seja, pertencentes ao conjunto {0, 1}.

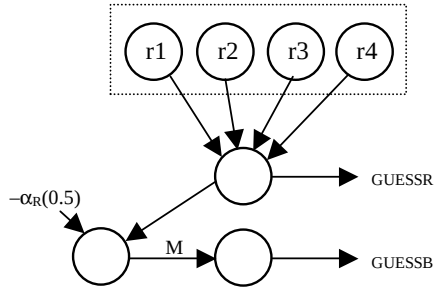


fig. 5.9.14 : Definição do canal GUESSR e GUESSB.

Por exemplo, o processo RANDR é construído da seguinte forma:

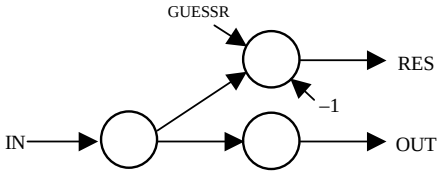


fig. 5.9.15 : Processo RANDR.

Neste caso, não é necessária a consulta aos neurónios ‘r?’, dado que em cada instante um e apenas um dos geradores terá um valor diferente de zero.

6. Como executar redes-σ eficientemente?

Apesar deste trabalho não ter como objectivo a definição ou a construção de sistemas físicos em *hardware*, apresentamos neste capítulo, uma proposta de arquitectura capaz de executar em paralelo e eficientemente, as computações necessárias para suportar a dinâmica das redes propostas nos capítulos anteriores.

6.1. Trabalhos Relacionados

Os processadores actuais são capazes de simular qualquer algoritmo de aprendizagem relativo a qualquer arquitectura finita de rede neuronal. Esta é a forma mais fácil e barata de executar este tipo de aplicação e, sem dúvida, a mais utilizada na área. Porém, em certas aplicações o uso de sistemas específicos de computação neuronal pode ser essencial. Enquanto o computador usual processa dados sequencialmente, um sistema de *hardware* neuronal, permite a execução maciçamente paralela de uma grande quantidade de dados melhorando, assim, o desempenho em várias ordens de magnitude.

Existem diversos produtos comerciais que executam, com grande eficiência, um conjunto de algoritmos de aprendizagem para redes neuronais. A forma

mais comum passa por construir uma placa de simulação neuronal, que ao conectar-se com o processador digital, recebe os dados e devolve os resultados obtidos. As tarefas realizadas por estes sistemas são: reproduzir a topologia da rede a emular, carregar as sinapses a partir dos dados, e classificar através do mapa de entrada-saída resultante da aprendizagem. Um trabalho detalhado sobre estratégias de construção de *hardware* neuronal em VLSI pode ser lido em [Duong *et al.* 96].

A maioria dos sistemas apenas permite a execução de algoritmos sub-simbólicos (os mais estudados na teoria e usados na prática). Entre eles, destaca-se o sistema da SIEMENS, *Synapse-3*, um neurocomputador (uma ou várias placas de conexão a um computador digital), capaz de emular uma grande quantidade de algoritmos de aprendizagem. Este sistema também possui uma linguagem de programação capaz de controlar as suas estruturas matriciais cujo desempenho atinge os 2 mil milhões de multiplicações por segundo (cf. [Ramacher 91, 94], [SIEMENS 98]). Este sistema não utiliza representações neuronais para guardar a topologia da rede, dado que existem numerosos problemas no que respeita a construir, literalmente, neurónios e sinapses em unidades de silício. Um dos principais problemas é o grau de intersecção necessário entre as várias ligações sinápticas da rede.

Outros trabalhos são [Bessière 91], [Wei-Ling 91], [Moore *et al.* 98], [Card 98]. Uma arquitectura que utiliza a função de activação σ é mostrada em [Chevtchenko *et al.* 97]. Em [Lighttower *et al.* 99] é apresentada uma máquina digital paralela capaz de executar o algoritmo SOM com 256 neurónios. Para encontrar resumos de trabalhos relacionados, ler [Aybay *et al*. 96] e [Ienne *et al.* 95]. Um estudo comparativo dos limites das representações físicas bidimensionais, comparativamente com o sistema tridimensional do sistema nervoso central pode ser lido em [Bein 98]. Para obter informações mais recentes, pode ser visitada a página *www1.cern.ch/NeuralNets/nnwInHepHard.html*.

6.2. Modelos Teóricos

Os primeiros modelos de máquinas teóricas eram máquinas de processamento sequencial. Apesar de não ser possível aumentar o seu poder computacional, é viável estudar formas de aumentar o seu desempenho, ou seja, diminuir o tempo gasto na execução de certos algoritmos, seja pela diminuição da complexidade do algoritmo em questão (que está fora do contexto desta Tese), seja pela possibilidade da máquina operar em paralelo. Quanto mais instruções de um algoritmo forem passíveis de ser executadas independentemente, mais rápida é a execução desse algoritmo numa máquina paralela. Restam assim, dois problemas gerais:

a) Obter um algoritmo paralelo para o problema em questão, e
b) Construir uma máquina paralela com o melhor desempenho possível.

O ponto a) foi tratado, com bastante pormenor, nos capítulos 4 e 5, onde foi demonstrado ser possível traduzir um algoritmo numa rede neuronal computacionalmente equivalente. O ponto b) será tratado neste capítulo, primeiro revisitando alguns modelos relevantes e, depois, apresentando um esboço de uma máquina que, à partida, se mostra adequada para emular o funcionamento de uma rede neuronal de tamanho arbitrário, capaz de alterar os seus próprios pesos sinápticos, facto essencial para a integração das computações simbólica e sub-simbólica.

Máquina Vectorial

Uma máquina vectorial processa vectores de bits por unidade de tempo. A característica principal deste modelo teórico é a capacidade de processar um vector de tamanho arbitrário, em cada instante.

Esta máquina possui um conjunto de instruções que pode executar, nomeadamente, atribuição de um vector a uma posição de memória, complemento de um vector, conjunção lógica de dois vectores, translação (*shift*) de N bits de um vector para a esquerda ou para a direita e ainda uma

instrução condicional de salto, que permite saber qual é a próxima instrução do programa.

Com este conjunto simples de operações é possível, por exemplo, construir grandes estruturas vectoriais de forma muito rápida, manipular estruturas matriciais ou realizar operações aritméticas (para mais informações, ler volume II de [Balcázar *et al*. 95]).

SIMDAG

SIMDAG
Single Instruction Stream, Multiple Data Stream, Global Memory

Esta máquina, baseada na máquina RAM de von Neumann, é constituída por uma unidade central, uma memória global e por um conjunto infinito de processadores. Cada processador possui uma memória local, onde lhe é possível guardar e alterar informação. Não sendo razoável ter um número infinito de processos a ser executados ao mesmo tempo, a máquina possui estruturas de controle que restringem a execução paralela a um número finito de processadores em cada instante.

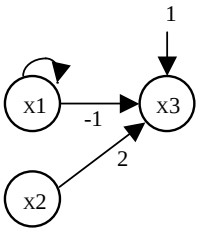
Existem dois tipos de instruções, as sequenciais, executadas pelo processador central, para acesso à memória global, e as paralelas que são executadas pelos processadores secundários (estes, porém, não incluem instruções condicionais nem de salto). O acesso à memória global é hierarquizado, sendo que o processador com índice menor tem a prioridade sobre os outros (que estão bloqueados enquanto esperam).

6.3. Emulação da Dinâmica Neuronal com Matrizes

Uma rede neuronal pode ser representada por uma matriz. Para uma rede R com N neurónios, a matriz M_R com $N\times(N+1)$ elementos, representa cada sinapse e pendor possível que pode ir de um neurónio da rede neuronal para qualquer outro.

O exemplo seguinte, exemplifica esta ideia. Existe uma rede com 3 neurónios, que é representada por uma matriz de 4 linhas e 4 colunas, sendo a coluna extra

(por definição, a 1ª coluna) reservada para os pendores dos neurónios, e a 1ª linha para que a multiplicação matricial produza um vector de dimensões iguais às do vector de activação (ver os próximos parágrafos).



R	b	x1	x2	x3
1	1	0	0	0
x1	0	1	0	0
x2	0	0	0	0
x3	1	-1	2	0

fig. 6.3.1 : Uma rede neuronal R e a sua matriz M_R correspondente.

Com esta representação e assumindo que os últimos valores de activação dos neurónios estariam num vector X de dimensão N+1 (a primeira componente do vector é sempre igual a um para multiplicar pelo valor do pendor). A dinâmica da rede, ou seja, o conjunto dos próximos valores de activação, é equivalente à multiplicação de M_R por X, obtendo-se um vector X^+ contendo essas novas activações (que têm de ser processadas pela função de activação σ, antes que ocorra uma nova iteração do processo).

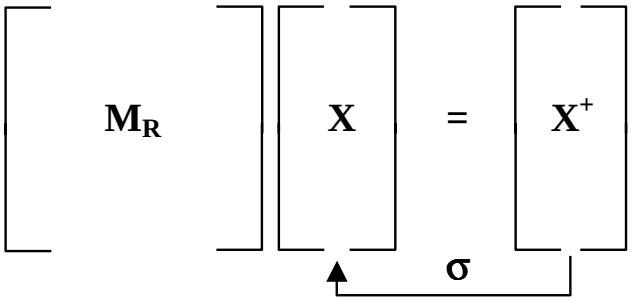


fig. 6.3.2 : Multiplicação de M_R por X.

Este esquema tem como vantagens a rapidez e simplicidade das computações necessárias. Apesar das ferramentas matemáticas estarem bem definidas no domínio das operações com matrizes esparsas (ou seja, matrizes cuja maioria de seus elementos é zero, cf. [Sangiovanni 76], [Rose 76], [Pan 82]), as desvantagens são relevantes. Para além do gasto excessivo de memória (uma rede com 1000 neurónios, necessita de uma matriz com mais de 1 milhão de elementos), não permite a modificação da própria matriz M_R (essencial para representar e executar processos de aprendizagem).

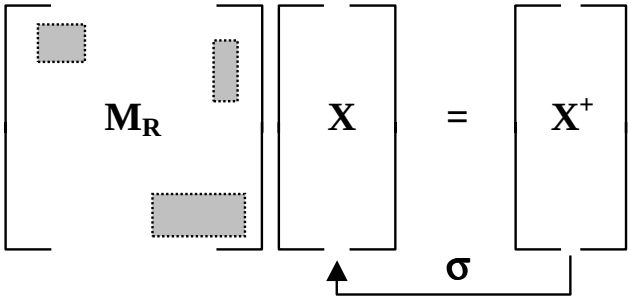


fig. 6.3.3 : Áreas de modificação de M_R representando processos de aprendizagem.

Por isso, esta arquitectura mantém as vantagens da representação matricial, mas reduz ao mesmo tempo os problemas decorrentes dessa abordagem. O primeiro passo estipula que a descrição da matriz seja optimizada para eliminar a esmagadora maioria de elementos nulos (ou seja, não gastar memória com sinapses que não existem). Já o segundo passo, a modificação da matriz, requer que certos cálculos produzam efeitos secundários na estrutura de representação da matriz (estes cálculos são, basicamente, as ligações neuro-sinápticas introduzidas no capítulo 5).

6.4. Representação da topologia da rede

Para representar de forma eficaz a topologia da rede, codificada nos elementos de uma matriz, é necessário codificar toda a ligação sináptica existente na rede e ao mesmo tempo eliminar os zeros da representação matricial, responsáveis por parte substancial da memória necessária (já que a maior parte dos neurónios da rede não são conectados entre si).

Existem três tipos de ligação sináptica: neurónio-neurónio, neuro-sinápticas e pendores. Cada uma delas será representada por um triplo que guarda a informação necessária sobre essa ligação. Graficamente, temos:

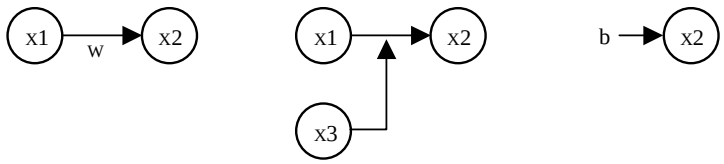


fig. 6.4.1 : Tipos de sinapse e seus respectivos triplos.

O tipo de informação dos argumentos determina o tipo de sinapse. Se o segundo argumento for real, o triplo identifica uma sinapse padrão entre dois neurónios, se o segundo argumento for o nome de um neurónio, o triplo indica uma sinapse neuro-sináptica, finalmente, se o primeiro e o segundo argumentos forem respectivamente 1 e um real, trata-se de um pendor.

Assim, uma rede neuronal em vez de uma representação matricial é definida por uma lista de triplos. Se a rede tiver S sinapses e pendores, a lista é constituída por S triplos. Esta lista é guardada num vector de triplos, denominado aqui por $\mathbf{S_R}$. De reparar que não é necessário representar os neurónios, dado que eles são implicitamente nomeados na lista. Esta lista possui um tamanho fixo, dado que a rede não pode criar ou destruir neurónios e sinapses em tempo de execução[18].

No entanto, é necessário guardar os valores de activação da rede, ou seja, o estado actual do processo da computação neuronal efectuado pela rede. Essa informação é guardada no vector $\mathbf{X_R}$. Do mesmo modo, o vector $\mathbf{X^+_R}$ guardará os valores da próxima activação, enquanto estes estão sendo calculados. O processo de actualização das várias activações, uma vez calculado $\mathbf{X^+_R}$, transforma cada elemento usando a função de activação σ, transferindo os novos valores para $\mathbf{X_R}$, completando assim, uma iteração.

$$\sigma(x) = \begin{cases} 0, & x<0 \\ x, & 0\leq x\leq 1 \\ 1, & x>1 \end{cases}$$

Para a execução e paragem desta máquina, definiu-se que o primeiro elemento de $\mathbf{X_R}$ representa o neurónio de activação da rede, e que o segundo elemento de $\mathbf{X_R}$ representa o neurónio de paragem. Assim, a máquina inicia a sua execução quando $\mathbf{X_R}[1]$ é igual a 1, e termina quando $\mathbf{X_R}[2]$ é igual a 1.

As estruturas vectoriais $\mathbf{X_R}$ e $\mathbf{S_R}$ guardam toda a informação relevante ao processo de computação efectuado pela rede neuronal. Falta a capacidade de

[18] No entanto, se existisse um processo de criação neuronal (como por exemplo, inserindo mecanismos de recursão), esta estrutura de dados não teria de ser alterada, pois poder-se-ia sempre incluir novos tuplos em S_R, enquanto houvesse espaço disponível em memória.

manipular a informação contida em $\mathbf{X_R}$ para calcular $\mathbf{X^+_R}$ e de processar a função de activação σ.

O processo de inicialização é definido nos seguintes passos:

- Carregar a topologia da rede em $\mathbf{S_R}$.
- Inicializar todos os elementos de $\mathbf{X_R}$ e $\mathbf{X^+_R}$ a zero (ou seja, a rede está inactiva).
- Atribuir uma lista de triplos a cada processador P_i.
- $\mathbf{X_R}[1] = 1$

Consideramos a hipótese de existência de P processadores em paralelo. Este número de processadores pode ser aumentado ou diminuído de forma transparente, apenas afectando o desempenho geral da execução e não alterando o funcionamento interno da máquina. Se P for igual a 1, então estamos perante uma máquina sequencial; para P maior que 1, a máquina executa P triplos em paralelo.

Sendo que a dimensão de $\mathbf{S_R}$ é igual a S cada processador executará S/P triplos. Para ser exacto, o processador P_i executará a lista de triplos desde $\lceil (S/P) \rceil (i-1) + 1$ até $\lceil (S/P) \rceil i$. Deste modo, o funcionamento da máquina não depende do número de processadores, sendo que a única mudança que ocorre reside no número de triplos que cada um tem de executar em cada iteração.

Cada triplo a executar representa uma soma de multiplicações, cujo resultado é adicionado ao conteúdo de um registo do vector $\mathbf{X^+_R}$. O maior problema (e o maior problema de todas as implementações paralelas) é o da comunicação, nomeadamente, no momento da escrita simultânea de dois processos num mesmo registo. A questão é solucionável com mecanismos de estafeta (como o proposto na figura 4.5.3), mas pode causar atrasos consideráveis na computação.

Porém, devido ao carácter extremamente modular da tradução da linguagem NETDEF em redes-σ, é possível organizar os triplos do mesmo módulo no mesmo processador, reduzindo assim, o tempo gasto na exclusão mútua das

instruções de escrita. Existindo um *hardware* como este, o próprio programador pode tomar atenção no estilo de programação, como não aceder às variáveis globais quando elas não são necessárias.[19]

O algoritmo executado pelos processadores, em cada ciclo, é o seguinte:

- Buscar o próximo triplo <a, b, c>
- Caso o triplo seja:
  - Neurónio-neurónio $\Rightarrow \mathbf{X^+_R}[c] = \mathbf{X^+_R}[c] + b * \mathbf{X_R}[a]$
  - Neuro-sinaptico $\Rightarrow \mathbf{X^+_R}[c] = \mathbf{X^+_R}[c] + \mathbf{X_R}[a] * \mathbf{X_R}[b]$
  - Pendor $\Rightarrow \mathbf{X^+_R}[c] = \mathbf{X^+_R}[c] + b$

Quando todos os processadores completarem os seus cálculos, é executado um ciclo que percorre cada elemento do vector $\mathbf{X^+_R}$ executando a atribuição $\mathbf{X_R}[i] = \sigma(\mathbf{X^+_R}[i])$. Deste modo, o vector $\mathbf{X_R}$ fica preparado para a próxima iteração.

As iterações ocorrem enquanto o valor de $\mathbf{X^+_R}[2]$ for igual a zero. Caso contrário, a operação termina. Os resultados da execução, ou foram convenientemente enviados pelos canais de saída (representados por alguns dos elementos de $\mathbf{X_R}$) ou estão guardados em um ou mais dos neurónios que guardam as activações das variáveis.

Uma optimização do processo apresentado seria eliminar todos os triplos referentes aos pendores. Em cada iteração, actualizar-se-ia directamente em $\mathbf{X^+_R}$ os valores dos pendores para cada neurónio. Por exemplo, se o neurónio de índice 10 possuísse um pendor igual a 0,5 no fim de uma iteração $\mathbf{X^+_R}[10]$ seria igual a 0.5 e não igual a zero, como apresentado anteriormente. Deste modo, todo o tempo gasto para executar os triplos referentes aos pendores seria reduzido a zero. A contrapartida é a necessidade de mais memória,

[19] Um caso semelhante é apresentado em [Yang e Ahuja 99], onde usando um método de particionar as amostras para efectuar um treino paralelo num algoritmo competivivo (ou seja, duas amostras activam dois neurónios diferentes), obteve-se uma melhoria de desempenho de 315% usando-se 4 processadores em simultâneo.

nomeadamente na criação de um novo vector **B** para guardar os pendores da rede, para poder actualizar correctamente, em cada iteração, os valores de $\mathbf{X^{+}_{R}}$.

Esta arquitectura, com o seu funcionamento simples e modular no que respeita à adição ou remoção de processadores, bem como do tipo de operações que realiza (adições, multiplicações e a função de activação σ) mostra que é possível construir estruturas computacionais em *hardware*, que executam de forma paralela e eficaz, as complexas redes-σ construídas ao longo desta Tese, nos vários algoritmos simbólicos e/ou sub-simbólicos.

7. Aplicação com Agentes

Neste capítulo apresentaremos um projecto de aplicação do sistema de computação em redes-σ, para verificar a funcionalidade dos processos da linguagem NETDEF em problemas onde o uso do paralelismo seja relevante. O capítulo centrar-se-á numa aplicação experimental de multiagentes, cujo objectivo é o de transferir pacotes de informação dentro de uma complexa rede de agentes para comunicação distribuída.

7.1. Introdução

O assunto que iremos abordar neste capítulo refere-se à questão de como enviar informação de um local A para um local B, sendo que ambos estão ligados através de uma rede de transmissão com dimensão e complexidade desconhecida. A informação que eles retêm é local (nenhum ponto da rede possuí uma descrição completa da rede – eventualmente, a sua própria estrutura pode ser dinâmica e alterar-se com o tempo) e é com essa informação que o emissor decide enviar a informação para o vizinho seguinte, o qual por sua vez toma uma nova decisão de enviar para um outro seu vizinho até que a informação chegue ao receptor desejado.

O objectivo desta secção é mostrar um breve conjunto de questões relacionadas com a área da Inteligência Artificial Distribuída (IAD), nomeadamente dos sistemas multiagentes, para contextualizar o exemplo de aplicação a ser apresentado (cf. [Russel e Norvig 95], [Álvares e Sichman 98] para maior informação sobre agentes e suas capacidades).

Dificuldade Teórica

A questão de encontrar um determinado caminho entre dois nós num dado grafo pode ser resolvido usando o algoritmo padrão de Dijkstra (cf. [Dijkstra 59]) que possuí uma baixa complexidade polinomial, nomeadamente, $O(n^2)+O(m)$, onde n é o número de nós e m o número de arcos; ou o algoritmo de Floyd de complexidade $\Theta(n^3)$ que produz melhores resultados para redes de maior conectividade. O algoritmo de Floyd encontra o caminho mais curto (dado um determinado critério) entre todos os vértices do grafo segundo o seguinte método: Os cálculos são executados sobre uma matriz bidimensional que representa os custos entre os arcos existentes. Se um nó *i* tem uma ligação com *j* de custo *k*, então nas posições (*i*,*j*) e (*j*,*i*) da matriz será inserido o valor *k*. Para todos os outros elementos da matriz (i.e., todos os pares de nós que não estão ligados directamente) insere-se um valor suficientemente alto para designar um custo 'infinito'. Tendo a matriz construída, o seguinte algoritmo é utilizado para calcular o custo mínimo entre todos os pares de pontos do grafo[20].

graph[i,j] = k
graph[j,i] = k

```
FOR (k = 1; k <= n; k++)
  FOR (i = 1; i <= n; i++)
    FOR (j = 1; j <= n; j++)
      IF (graph[i,j] > (graph[i,k] + graph[k,j])) {
          graph[i,j] = graph[i,k] + graph[k,j];
          path[i,j] = k;
      }
```

No entanto, existem duas questões relevantes. A primeira é que o critério para a definição de qual caminho (ou seja, as métricas a serem usadas) pode alterar a

[20] Esta descrição não foi representada em NETDEF por motivos de clareza, dado que o mecanismo de matrizes bidimensional não foi implementado no compilador. O programa seria executado usando vectores unidimensionais. Porém, nesse caso, o algoritmo perderia legibilidade e prejudicaria a exposição.

complexidade do problema. Nos sistemas ATM de movimento bancário, por exemplo, a preocupação em relação à taxa de transmissão, atraso na comunicação e percentagem de perda de pacotes de informação, transforma o problema em NP-completo colocando o algoritmo, anteriormente polinomial, no domínio da classe exponencial. Infelizmente, dadas as necessidades das aplicações actuais em relação às redes de comunicação, este é o caso geral e mais comum (cf. [Zhan 97] e [Wang e Crowcroft 96] para um estudo destas questões). A segunda questão é que a resolução do algoritmo implica que toda a informação fique concentrada numa memória única, ou seja, que a topologia da rede, a informação relevante de cada arco (por exemplo, o atraso de cada ligação), e todos os detalhes relevantes têm de se concentrar num determinado local. Isto pode ser impossível para determinados problemas (por exemplo, a estrutura da Internet não pode ser mapeada e centralizada em tempo útil, dada a sua dimensão, dinamismo e ainda os objectivos divergentes das companhias que a suportam).

Abordagem Centralizada vs. Abordagem Distribuída

Num ambiente centralizado, onde existe um servidor capaz de guardar e gerir toda a topologia da rede, cada unidade de endereçamento recebe actualizações da parte do servidor, para reenviar apropriadamente os seus pacotes de informação. Esta abordagem tem como vantagens, a simplificação dos protocolos de organização da rede, dado que basta o simples algoritmo de Floyd para determinar os caminhos mais curtos do grafo; e o controle explicito pelo servidor do fluxo da rede e dos seus atrasos de transmissão. No entanto, para além do referido na secção anterior, este sistema possui desvantagens relevantes:

- No caso da rede ser bastante dinâmica (taxa elevada de criação e remoção de arcos e nós) é criado um fluxo muito intenso de acesso ao servidor, sobrecarregando-o e introduzindo uma componente de atraso pelas potenciais filas de espera que serão criadas pelo facto de ter de avisar todos agentes da rede das alterações ocorridas.

- Os algoritmos de optimização do caminho mais curto são de complexidade polinomial (como o de Floyd visto anteriormente), mas para um determinado número de agentes, o tempo gasto no novo cálculo da matriz não é desprezível. Por exemplo, para mil agentes são necessárias duas matrizes de um milhão de elementos, ocorrendo mil milhões de comparações entre os elementos da matriz. Se forem dez mil agentes, o número de comparações sobe para um milhão de milhões.

Em relação à abordagem distribuída, o uso de agentes paralelos e independentes é uma solução especificamente dirigida às propriedades intrínsecas do problema apresentado. A grande quantidade de informação requerida para a decisão de transmissão é distribuída por natureza (ao longo dos vários pontos da rede de comunicação) e pode alterar-se ao longo do tempo (locais e canais de comunicação são criados ou desactivados). Isto implica que a organização necessária para a gestão dos recursos e sua subsequente distribuição tornam-se muito complexas de controlar se não for efectuada uma divisão de trabalho, ou seja, é necessário transformar todas as decisões de transmissão e cálculo de caminhos em assuntos restritos dos agentes relacionados. Para ser implementado, tem de existir um controle local capaz de distribuir a informação juntamente com as múltiplas decisões de retransmissão que ocorrem ao longo da rede. Os controladores locais têm de ser capazes de resolver os problemas a eles propostos com o seu conhecimento local, mesmo apesar dos eventuais e inesperados problemas que possam ocorrer (por exemplo, a desactivação de um controlador usado num determinado conjunto de caminhos). Quanto mais restrito esse controle local for (somente os agentes estritamente necessários devem ser sobrecarregados com a tarefa de retransmissão), melhor as condições gerais de tráfego na rede. O uso de *broadcasts* (mensagens direccionadas para todos os controladores) não é uma ferramenta apropriada neste contexto.

A distribuição de controle também evita os problemas intrínsecos da sincronização, que além de ser um mecanismo de *broadcast*, implica que todo o sistema precisa esperar a execução dos seus elementos mais lentos, gastando

demasiado tempo na manutenção da consistência global, ao contrário da necessidade bem mais simples de coerência local entre unidades vizinhas. Um exemplo simples deste ponto pode ser observado quando uma determinada unidade é desactivada. Essa informação percorre a rede de vizinho em vizinho de forma local. De certa forma, é como se a rede (do ponto de vista global) procurasse obter uma coerência ao longo do tempo, através de mecanismos locais e não centralizados. Seria um tipo de comportamento emergente da soma dos comportamentos individuais, e não algo programado à priori para resolver o problema (cf. [Willmott *et al.* 99], [Willmott e Faltings 99], [Kramer *et al.* 99], [Jennings *et al.* 99] para maiores discussões sobre o assunto).

Algumas propostas no domínio da IAD para resolver a questão de endereçamento numa rede de comunicação, incluem:

- Modelos emergentes de agentes reactivos (cf. [White e Pagurek 98, 99], [Subramanian *et al.* 97], [Schoonderwoerd *et al.* 97]). Uma das inspirações destes modelos são os comportamentos observados entre os insectos sociais, como o uso de ferormonas pelas formigas na definição dos seus caminhos.
- Uso de conceitos económicos como a noção de mercado e dos seus mecanismos próprios, tais como, recursos, capacidade de alocação desses recursos e ferramentas de troca e/ou preços (cf. [Prouskas *et al.* 2000], [Wellman 93], [Patel *et al.* 2000], [Gibney e Jennings 97]).
- Redes activas onde cada pacote deixa de ser visto como um conjunto inerte de informação, transformando-se num processo capaz de decidir, em tempo real, qual a sua própria rota até ao receptor final. Aqui, cada nó da rede de comunicação é somente uma passagem e um reservatório de informação necessária aos pacotes, para eles próprios tomarem as decisões do caminho a tomar (cf. [Kramer *et al.* 99], e ainda [Tennenhouse *et al.* 97] para um resumo destes trabalhos).

O caminho adaptado neste capítulo é diferente destas três visões. Um agente (representando um determinado nó da rede) não é um colector de informação (como nas abordagens emergentes), ou um comprador de um certo caminho

preestabelecido (como nas abordagens de mercado), mas é capaz de tomar uma decisão baseado no estado actual da rede onde se insere (ou melhor, no estado da rede que ele conhece localmente), mudando as suas decisões consoante as alterações que lhe forem sendo comunicadas por processos internos (e.g., eliminação de uma das suas ligações) ou por notícias provenientes dos seus vizinhos (e.g., um agente não vizinho foi desactivado).

7.2. Topologia da Rede de Transmissão

A topologia da rede de transmissão segue o padrão básico dos grafos bidireccionais etiquetados, onde um conjunto de nós (ou locais) é conectado por um conjunto de arcos (ou ligações), e onde cada arco possuí uma etiqueta (sendo um número racional positivo) que representa o atraso da informação (em unidades de tempo) dessa mesma ligação. Um exemplo possível encontra-se na seguinte figura:

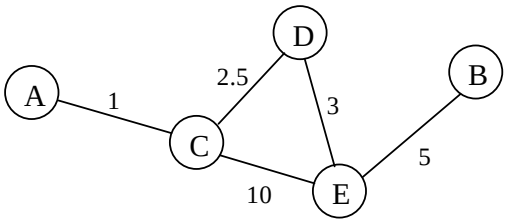


fig. 7.2.1 : Uma rede de transmissão.

Neste exemplo, existem dois caminhos possíveis para ir do local A para o local B. Dependendo do critério, os agentes que recebem o pacote tentarão determinar qual a melhor forma de retransmiti-lo para que chegue ao destino maximizando o critério pedido. Vejamos dois critérios distintos: a) obter o caminho mais rápido; e b) obter o caminho mais curto.

No caso a) quando o pacote chega a C, este decide enviar por D para minimizar o tempo de atraso (caminho mais rápido A→C→D→E→B com atraso total de 11.5 unidades de tempo). No caso b) quando o mesmo pacote chega a C, este decide enviar por E para minimizar o caminho percorrido (caminho mais curto A→C→E→B com uma passagem por 2 nós intermédios).

7.3. Algoritmo de Encaminhamento

O algoritmo possuí algumas propriedades que definem o nosso problema de retransmissão:

- Os caminhos de transmissão não são previstos à priori, mas sim calculados e substituídos quando ocorre uma determinada alteração na rede. A decisão de qual caminho a tomar é assim, uma consequência do estado da rede quando é pedido o envio de informação. O processo de retransmissão adapta-se ao estado da rede.
- O caminho não é calculado na sua totalidade por qualquer agente. Cada agente possuí na sua memória local apenas uma parte desse caminho, nomeadamente, qual o próximo agente a enviar a informação.
- O sistema é proactivo e não reactivo: os problemas têm de ser evitados antes que possam ocorrer. Isso implica que os agentes são, pelo menos até um determinado nível de complexidade, cognitivos e não reactivos.

Os agentes, ou seja, os nossos controladores locais têm de possuir dois elementos essenciais:

- Informação (mesmo que incompleta) sobre como a rede ou parte da rede está definida. Esta conhecimento é necessário para a decisão de qual caminho o pacote de informação deve tomar;
- Distribuição do controle e da decisão dos caminhos. A forma como é encontrado o caminho não depende somente de um controlador, mas é definido ao longo do percurso desde o emissor até ao destino desejado.

Usando os termos de [Willmott e Faltings 2000], uma *coordenação* é definida por um conjunto de agentes trabalhando em conjunto para resolver uma determinada tarefa (ou um conjunto limitado de tarefas); enquanto uma *organização* é um conjunto mais durável de protocolos de comunicação entre um grupo de agentes para várias tarefas não necessariamente relacionadas.

No nosso problema, a organização que reúne os vários agentes da rede é representada pelo conjunto de mensagens reconhecidas entre os agentes para comunicação e actualização da topologia da rede (e.g., ‘informo que tenho um

novo vizinho'). Já uma coordenação é um episódio temporário de transferência de informação entre o agente emissor e o receptor final, mais todos os agentes intermédios percorridos pelo pacote de informação. Veremos em seguida estes dois aspectos que definem o algoritmo de encaminhamento.

Organização na Rede

A organização dos agentes relacionados dentro de uma mesma rede de comunicação é definida pelo conjunto de mensagens que todos são capazes de reconhecer, ou seja, um dado protocolo de comunicação. Esse protocolo trata essencialmente da topologia da rede. As mensagens que os agentes recebem do ambiente são as seguintes:

- Desactivação (remoção do próprio nó);
- Criação de uma ligação com um novo vizinho (inserção de um arco);
- Desactivação de uma ligação com um novo vizinho (remoção de um arco);
- Actualização de um tempo de transmissão para um determinado local de um vizinho (relevante para o critério do caminho mais rápido);
- Pergunta sobre o atraso e o caminho de um dado local.

vector de envio

Esta informação é actualizada num vector, designado doravante por vector de envio, que guarda os seguintes tuplos <*local de destino, tempo de transmissão, local para envio no tempo mais curto, dimensão do percurso, local para envio no caminho mais curto*>. Ou seja, para enviar uma mensagem para um dado destino, o agente reconhece num tuplo apropriado qual o local de envio e qual o tempo de transmissão ou dimensão do percurso estimado para chegar ao destino. Com este vector, a tarefa dos agentes de retransmissão de informação fica limitada a percorrer o vector até encontrar o tuplo indicado (cada agente guarda a informação de todos os destinos possíveis), uma tarefa de complexidade linear. É na manutenção deste vector que reside a dificuldade do algoritmo, e é nele que se encontra o seu carácter proactivo: quando o agente recebe um aviso de alteração da rede, procura a informação necessária (interna, consultando as suas estruturas de dados, ou externa, perguntando aos seus vizinhos) e prepara uma solução para a eventualidade de um pacote de informação precisar desse caminho no futuro.

Veremos primeiro como se efectua o envio de um pacote de informação pela rede, para depois estudarmos os mecanismos de resolução prévia dos caminhos disponíveis para o fluxo de envio de pacotes.

Coordenação na Rede

A coordenação sendo um episódio temporário onde a transmissão de informação entre dois locais é efectuada, é activada num dado agente quando este recebe um pedido de reenvio de informação. O tipo da mensagem de recepção de informação é única, e vem com a identificação do receptor final (o qual pode ser o próprio terminando aí o seu trajecto, ou para outro diferente, reenviando a informação para um terceiro agente) juntamente com o critério a adoptar na retransmissão (como por exemplo, usar o caminho mais rápido). Tendo em conta o que foi dito sobre o vector de envio, este problema é de resolução trivial: Determinar se o destino desejado é ele próprio, terminando a coordenação actual, ou caso contrário, encontrar o próximo agente a reenviar a mensagem procurando no vector de envio com o critério apropriado.

Resolução proactiva dos caminhos possíveis

Sendo a rede uma estrutura dinâmica, acontece que pode não existir uma coerência global em relação aos vectores de envio de todos os agentes (por exemplo, se um agente considerar um determinado caminho no qual desapareceu um outro agente intermédio, cuja notícia ainda não foi recebida devido à distância entre os dois). O objectivo é fazer com que os vectores de envio de todos os agentes sejam usados localmente para uma convergência mútua entre pares de agentes, a qual terá como consequência a convergência global do sistema (quanto menos dinâmico o ambiente, menor será a pressão sobre este processo de actualização). Vejamos cada evento separadamente:

- Criação de um novo local – Enquanto um agente estiver desconectado da rede qualquer actividade de envio/recepção de informação é impossível. Não altera em nada o estado interno dos outros agentes.
- Criação de uma nova ligação directa – Os dois agentes referidos nesta ligação são informados pelo controlador. Cada agente informa o outro

sobre o seu próprio vector de envio. A partir daqui, ambos informam todos os outros vizinhos da nova ligação e dos seus atrasos intrínsecos, e das alterações ocorridas pela análise do vector de envio do vizinho.

- Criação de uma nova ligação indirecta – Um agente recebe de um vizinho um novo caminho para um dado agente mais o atraso respectivo. Ele analisa o vector de envio para determinar se esta nova opção é mais rápida que a actual. No caso afirmativo, actualiza o vector e envia para os outros vizinhos a informação desta nova actualização. Isto não cria um ciclo infinito de mensagens enviadas, dado que a cada reenvio, o atraso da conexão aumenta até se tornar indesejável para todos os agentes receptores.

Vejamos um simples exemplo da execução deste protocolo de comunicação para a inserção de conexões entre pares de agentes. Seja a rede representada na figura 7.3.1(a):

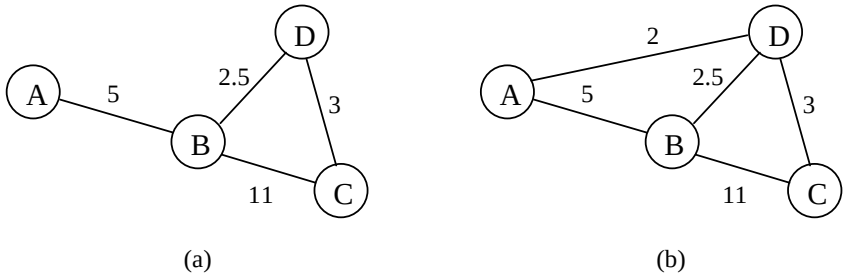


fig. 7.3.1 : Inserção de uma nova ligação em uma rede de comunicação.

Como foi referido anteriormente, cada tuplo do vector de envio é constituído por 5 elementos: O agente destino, qual o menor tempo e por onde continuar, qual o menor percurso e por onde continuar. Por exemplo, o vector do agente A é igual a [<B,5,B,1,B>, <C,10.5,B,2,B>, <D,7.5,B,2,B>]. Todos os caminhos de A passam por B dado ser esse o único caminho possível. Se inserirmos uma nova ligação (figura 7.3.1(b)) entre os locais A e D, um conjunto de mensagens será enviado para os seus respectivos vizinhos. O protocolo construirá uma conversa semelhante à seguinte[21]:

[21] Semelhante dado que cada agente é um processo paralelo e assim, a ordem de execução pode ser ligeiramente diferente. No entanto, isso não prejudica ou invabiliza a convergência para o resultado desejado, i.e., a optimização dos vectores de envio.

```
[A] Actualiza ligação para D com atraso 2 via D
[D] Actualiza ligação para A com atraso 2 via A
[A→B] Nova ligação para D com atraso 2
[A→D] Ligação para B com atraso 5
[A→D] Ligação para C com atraso 10.5
[D→B] Nova ligação para A com atraso 2
[D→C] Nova ligação para A com atraso 2
[D→A] Ligação para B com atraso 2.5
[D→A] Ligação para C com atraso 3
[B] Actualiza ligação para A com atraso 4.5 via D
[B→C] Nova ligação para A com atraso 4.5
[C] Actualiza ligação para A com atraso 5 via D
[C→B] Nova ligação para A com atraso 5
[A] Actualiza ligação para B com atraso 4.5 via D
[A] Actualiza ligação para C com atraso 5 via D
[A→B] Nova ligação para C com atraso 5
```

Nesse momento, os agentes que receberam as últimas mensagens não têm necessidade de alterar os seus vectores de envio (os tempos recebidos não são melhores) e assim, não enviam mais mensagens. Isso termina o processo de interacção. O vector de envio de A ficou com o seguinte conteúdo: [<B,4.5,D,2,D>,<C,5,D,2,D>,<D,2,D,1,D>]. Foram enviadas 10 mensagens no total, o que resulta em uma média de 2.5 mensagens por agente.

A outra metade dos eventos incluídos no protocolo de organização refere-se à eliminação de arcos e nós:

- Desactivação de uma ligação directa – Os dois agentes da ligação informam os outros seus vizinhos deste facto. Em seguida, voltam a perguntar aos mesmos vizinhos se algum deles possuí uma outra ligação alternativa.
- Desactivação de uma ligação indirecta – O agente ao receber essa informação verifica o seu vector de envio. Se possui uma opção alternativa, informa o emissor sobre essa opção. Se não, pergunta aos seus vizinhos se existe alternativa. Se receber uma alternativa, guarda-a no vector e envia aos vizinhos como se se tratasse da criação de uma nova ligação indirecta.
- Desactivação de um local – O agente a ser desactivado envia a todos os seus vizinhos uma informação de desactivação directa da respectiva ligação que os une.

- Pergunta sobre uma ligação indirecta – O agente analisa a sua estrutura interna (nomeadamente no vector de envio) à procura da resposta. Em caso afirmativo, responde desencadeando um evento de criação de uma ligação indirecta. Em caso negativo, retransmite a pergunta aos seus outros vizinhos.

Vejamos um exemplo deste tipo de acontecimento, usando novamente a rede da figura 7.3.1. Consideramos que a rede actual é a 7.3.1(b) e que é eliminada a conexão que une os agentes A e D, obtendo-se, deste modo, a rede 7.3.1(a). A comunicação entre os agentes da rede seria a seguinte:

```
[A] Desactivar ligação com D
[D] Desactivar ligação com A
[A→B] Remoção de ligação indirecta com D
[D→C] Remoção de ligação indirecta com A
[A→B] Remoção de ligação indirecta com C
[D→B] Remoção de ligação indirecta com A
[B] Substituindo ligação para A por ligação directa com atraso 5
[B→A] Nova ligação para C com atraso 5.5
[B→D] Nova ligação para A com atraso 5
[A] Criando nova ligação para C desde B com atraso total 10.5
[D→C] Você tem uma conexão com A? [não, mas vou procurar...]
[D→B] Você tem uma conexão com A? [sim, usando A]
[D] Criando nova ligação para A desde B com atraso total 7.5
[B→A] Nova ligação para D com atraso 2.5
[B→C] Remoção de ligação indirecta com A
[A→B] Você tem uma conexão com C? [sim, usando D]
[B→D] Nova ligação para A com atraso 5
[A] Criando nova ligação para D desde B com atraso total 7.5
[B→C] Nova ligação para A com atraso 5
[A→B] Você tem uma conexão com D? [sim, usando D]
[C] Criando nova ligação para A desde B com atraso total 10.5
[B→A] Nova ligação para C com atraso 5.5
[D→C] Nova ligação para A com atraso 7.5
[C→D] Nova ligação para A com atraso 10.5
[A] Actualizando ligação para C desde B com atraso total 10.5
[C→B] Remoção de ligação indirecta com A
[C→B] Você tem uma conexão com A? [sim, usando A]
[D→C] Nova ligação para A com atraso 7.5
[B→C] Nova ligação para A com atraso 5
[B→A] Nova ligação para D com atraso 2.5
```

Esta troca de 22 mensagens (média de 5.5 por agente) revela a maior complexidade reservada à eliminação de conexões existentes na rede. Isto deriva do facto que a remoção de uma conexão útil para um dado caminho, tem de ser compensada proactivamente para resolver o problema de pedidos futuros que usariam essa conexão (agora inexistente). Os vários elementos da rede têm de comunicar uns com outros, para encontrar a próxima melhor opção. Neste

exemplo, depois de terminada a comunicação, o vector de envio do agente A passou a ser igual a [<B,5,B,1,B>, <C,10.5,B,2,B>, <D,7.5,B,2,B>], precisamente o estado em que se encontrava antes da inserção e consequente remoção da ligação A↔D.

A estrutura neuronal

Um agente espera um pedido dentro de um conjunto possível de pedidos, e desencadeia os procedimentos necessários à sua correcta execução. Cada um destes pedidos tem uma determinada independência que pode ser aproveitada no uso de módulos paralelos. A estrutura da um agente neuronal é a seguinte:

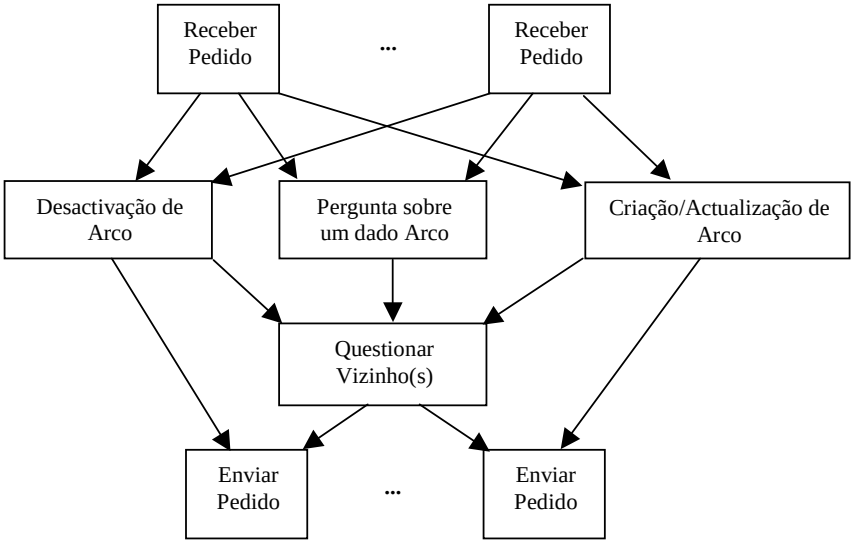


fig. 7.3.2 : Fluxo de informação de organização na Rede.

Cada caixa representa um processo paralelo. Os processos de receber e enviar pedidos formam a estrutura de comunicação entre agentes vizinhos. Cada conexão da rede possui dois processos exclusivos (um em cada agente) que tratam da comunicação exclusiva a essa conexão sem sobrecarregar as outras ligações (pressupõem-se um número máximo de ligações por agente)[22].

Os processos de entrada – as caixas Receber Pedido – requerem a execução do pedido aos outros processos do seu agente. Quando o pedido é processado, este

[22] Se não existir um número máximo, este conjunto de processos são substituídos por um único com uma fila de espera única que recebe os pedidos do exterior.

pode desencadear a activação de outros processos até que (eventualmente) sejam activados os processos de comunicação externa, enviando pedidos ou informações para os seus agentes vizinhos. Para tratar os episódios de coordenação, ou seja, a transferência de informação entre os vários agentes da rede, a estrutura é bem mais simples sendo efectuada por um sistema de comunicação que interliga todos os processos de entrada e saída do agente. O processo que recebe o pedido analisa o vector de envio para descobrir qual o próximo agente a receber o pacote. Deste modo, é possível remeter para o processo responsável de enviar informação ao próximo agente.

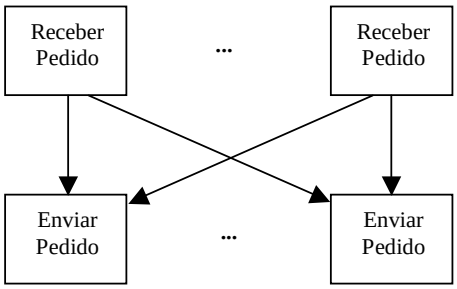


fig. 7.3.3 : Fluxo de informação de coordenação na Rede.

Assim, o agente é transformado num conjunto de processos paralelos que interagem uns com os outros para maximizar o uso das suas próprias potencialidades quando duas ou mais requisições de serviços dos seus vizinhos ocorrem simultaneamente.

7.4. Simulação

Na falta de uma máquina capaz de executar em paralelo as redes propostas nesta Tese (conferir o próximo capítulo sobre esta problemática), foi construído um programa capaz de simular num computador as redes produzidas pelo compilador NETDEF. No entanto, o nosso programa de compilação e simulação não possuí capacidade para executar processos concorrentes (neste caso, os módulos de redes neuronais a serem executados paralelamente).

O ambiente SACI

Tendo isto em conta, mais o facto de estarmos a realizar uma simulação num ambiente multiagente, decidimos utilizar uma ferramenta orientada a agentes e

apropriada para esta aplicação. A ferramenta, designada por SACI, é desenvolvida[23] pela equipe do Prof. Jaime Simão Sichman, nomeadamente pelo aluno de doutoramento Jomi Fred Hübner (cf. [Hübner e Sichman 2000]). O SACI visa facilitar a programação de sistemas de comunicação entre conjuntos de agentes distribuídos. O SACI disponibiliza um conjunto de bibliotecas em Java que aceleram o desenvolvimento de aplicações multiagentes, com a vantagem da independência de plataforma física que a linguagem desenvolvida pela SUN possui como característica central da sua filosofia de compilação.

Simple Agent Communication Infrastructure

A estrutura da simulação

Foi decidido programar um controlador cuja tarefa é criar e manipular a topologia da rede de comunicação. Este controlador é capaz de inicializar novas instâncias de agentes e, de seguida, ligar esses novos agentes à rede através da criação de conexões apropriadas. Para além disso, o controlador também informa pedidos de envio de pacotes de informação para um determinado agente, indicando-lhe o destino e o critério desejado. Os critérios programados nesta simulação são: a) caminho mais rápido; b) caminho mais curto; e c) caminho através de um local intermediário. O agente é um sistema independente capaz de processar os eventos externos provenientes ou do controlador ou dos seus agentes vizinhos. O agente possui um conjunto de informação interna (como o já referido vector de envio) que aliado ao algoritmo descrito dão uma dimensão cognitiva à sua capacidade de resolução.

Comparação de Resultados

Nesta secção são apresentados dados que mostram algumas das características da abordagem centralizada e da abordagem distribuída. Os dados seguintes focam sobre os episódios de organização da rede, dado que é nesse ponto que se encontram as diferenças entre os dois métodos. Em relação aos episódios de coordenação (simples envio de informação entre agentes sem alteração da

[23] Conferir `http://www.lti.pcs.usp.br/saci` onde é possível obter a versão mais actual desta aplicação.

topologia) os sistemas têm um comportamento semelhante, dado que os vectores de envio que possuem são iguais (o critério a maximizar é o mesmo).

A questão principal foca na complexidade dos dois algoritmos e no tráfego da rede necessário para resolver o problema do caminho mais curto. Por isso, foram consideradas para análise as seguintes componentes: o tempo de processamento interno e o número de mensagens para cada adaptação correspondente a uma alteração da topologia da rede. Seria trivial pensar que um servidor central possa estar ligado directamente a todos os agentes de retransmissão (se assim fosse, bastava enviar a mensagem para o servidor e este endereçava directamente para o agente destino...). Logo, um agente quando envia ou recebe um pedido do servidor tem de enviá-lo pela rede até que chegue a um agente com contacto directo com o servidor. Façamos uma estimativa optimista de $\log_q N$ transferências desde e até ao servidor, sendo q o número de ligações média por agente. O número de mensagens trocadas é aproximadamente de $\log_q N$ para a comunicação do servidor com cada agente, ou seja, $N. \log_q N$.

Na tabela 7.4.1, o par de números da primeira coluna significam o número de agentes na rede e o número médio de ligações por agente. Por exemplo, a primeira linha refere-se a uma rede com 3 agentes com uma conexão média de duas ligações por agente. A segunda e quarta coluna dizem respeito ao processamento interno do servidor necessário à adaptação da rede a uma alteração nas ligações. No caso centralizado, o cálculo diz respeito à complexidade polinomial do algoritmo de Floyd, e no caso distribuído, à complexidade linear das operações efectuadas por cada agente[24].

As restantes colunas indicam o número médio de mensagens na rede. No caso centralizado é o número total de mensagens. No caso distribuído, o primeiro

[24] Apresentamos o número estimado em relação à complexidade de cada um dos algoritmos. Haveria a necessidade de multiplicar por uma constante dependente dos pormenores de cada um dos programas, que não é relevante já que a sua influência (no que diz respeito à comparação dos resultados) é progressivamente menor, à medida que a dimensão do problema aumenta.

elemento da quinta coluna refere-se igualmente ao número total de mensagens e o segundo elemento representa a média de mensagens enviadas por agente. No caso dos agentes distribuídos foram executados dez exemplos com topologias diferentes sendo apresentada a média resultante.

Topologia da Rede	Centralizada		Distribuída	
	Processamento	Mensagens	Processamento	Mensagens
3 – 2	27	4,8	3	4,3 – 1,4
4 – 2	64	8,0	4	7,2 – 1,8
5 – 2	125	11,6	5	15.8 – 3,2
8 – 3	512	15,1	8	65,4 – 8,2
10 – 3	1000	21,0	10	143,8 – 14,4
15 – 4	3375	29,3	15	389,4 – 26,0

fig. 7.4.1 : Comparação entre as duas abordagens.

Graficamente, podemos comparar o desempenho das duas abordagens, tanto do ponto de vista do crescimento do processamento, como da troca de mensagens ocorridas no interior da rede de comunicação. No próximo gráfico, é mostrado o número de mensagens usando os dois sistemas, apresentando para o caso distribuído os valores totais e o valor por agente.

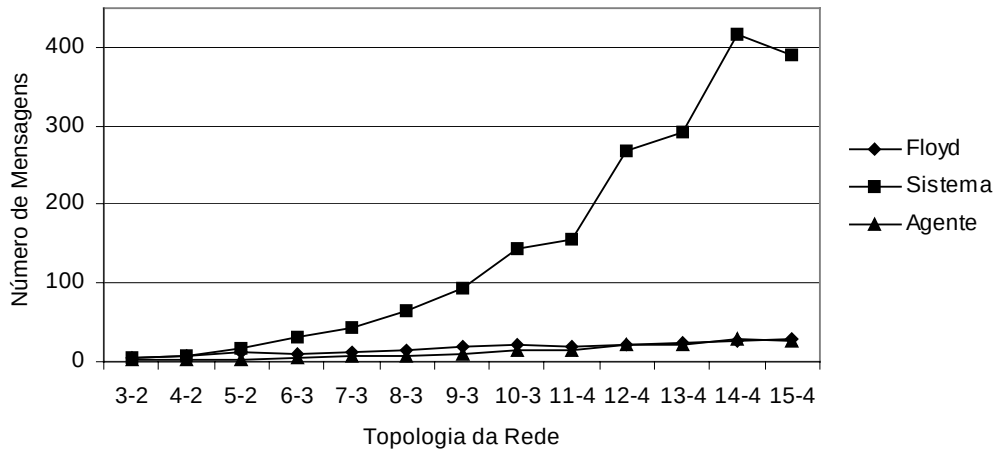


fig. 7.4.2 : Número de mensagens na Rede e por Agente.

Verificou-se que a inserção de um arco mais rápido do que os seus 'concorrentes' implicou numa maior troca de informação entre os agentes interessados. Já um arco que não oferece um serviço melhor que os existentes, provoca um reduzido número de mensagens, especificamente entre o par de

agentes desse arco e os seus respectivos vizinhos. Foi também observado durante a fase de recolha de dados, que as operações de inserção de ligações requerem menos mensagens que as operações de remoção. Para os casos com 15 agentes foram necessárias quatro a seis vezes mais mensagens nos exemplos de remoção de um arco importante para a comunicação de duas áreas distintas da rede. Já a remoção de arcos menos usados provocou um muito menor número de mensagens trocadas, sendo semelhante às inserções de arcos 'lentos'. Como seria de esperar, e de acordo com os dados obtidos, quanto maior a importância de um arco ou de um nó na rede de comunicação, maior é a actividade de convergência da própria rede como consequência dessa mudança. Neste próximo gráfico é mostrado a diferença de complexidade entre o trabalho executado pelo algoritmo centralizado e o paralelismo da solução multiagente.

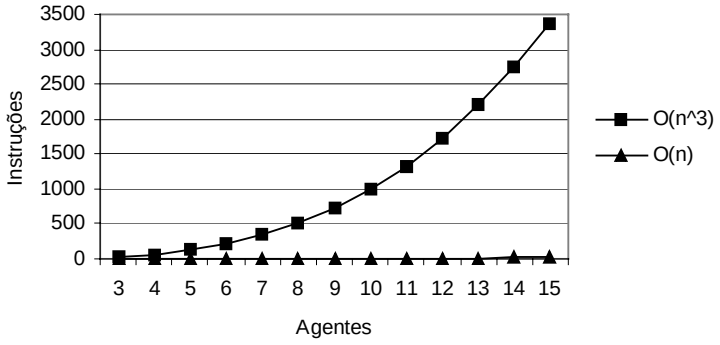


fig. 7.4.3 : Complexidade polinomial *vs.* complexidade linear.

Verifica-se que, apesar das duas complexidades pertencerem ao domínio polinomial, o sistema multiagente apresentado diminuiu a complexidade intrínseca do algoritmo padrão de escolha do caminho mais curto para um dado grafo. Isso foi conseguido à custa do paralelismo da solução que tem como principal consequência o aumento das mensagens trocadas ao longo da rede. Houve por isso uma troca na sobrecarga dos recursos, ou seja, entre um menor número de cálculos necessários e um subsequente aumento no tráfego da rede de comunicação.

No entanto, este facto não é uma desvantagem porque maximiza um recurso distribuído – a capacidade de transmissão definida pelo conjunto dos agentes e das suas conexões que definem a topologia da rede – dividindo a complexidade da tarefa por todos os agentes da rede com as vantagens referidas no decorrer deste capítulo, nomeadamente, um tratamento local para alterações locais, uma adaptação implícita do sistema em relação a um ambiente dinâmico, a característica de não precisar de um mecanismo de coerência global, e ainda o facto de não ser necessário um mecanismo centralizador por onde o fluxo da informação e o controle do seu conteúdo seriam subordinados.

7.5. Conclusão

Este capítulo pretendeu mostrar uma aplicação prática do uso do paralelismo para tratamento de uma tarefa complexa, nomeadamente na transferência de informação dentro de uma rede de agentes distribuídos. Estes agentes são capazes, isoladamente, de tomar decisões sobre para onde enviar informação, tendo em conta apenas a análise da informação local (e portanto, incompleta).

Cada agente pode ser visto como um conjunto de processos mais simples, a actuar em paralelo, recebendo e enviando informação de e para os outros processos internos ao agente ou para o exterior (i.e., para os seus agentes vizinhos). Estes processos, por sua vez, são definidos a partir de algoritmos específicos que podem ser implementados em redes neuronais, usando a linguagem NETDEF. Logo, é possível ter agentes dentro de um sistema multiagente, construídos a partir de uma arquitectura neuronal, que para além do aspecto de controle descrito nas secções anteriores, poderiam ter incluídos processos de aprendizagem naturalmente integráveis na arquitectura apresentada, como foi descrito no capítulo 5.

Ou seja, esta aplicação procurou mostrar que o conjunto de ferramentas apresentadas no decorrer desta dissertação, serve para construir sistemas de processamento relativamente complexos executados exclusivamente por redes neuronais.

8. Conclusão

Será apresentada uma breve discussão dos resultados obtidos neste trabalho e de quais as contribuições relevantes do mesmo para a área cientifica em que se insere. De seguida, alguns aspectos passíveis de investigação futura serão revistos, juntamente com sugestões que justifiquem a sua inclusão. Conclui-se com algumas considerações finais.

8.1. Discussão

No desenrolar desta Dissertação, apresentamos várias visões possíveis da computação, com diferentes formas de expressar e descrever soluções e algoritmos. Estas visões ou paradigmas da computação, passam pelas funções parciais recursivas, pelas máquinas abstractas como máquinas de Turing, ou redes neuronais artificiais inspiradas na neurobiologia. Elas possuem o mesmo poder computacional, sendo capazes de executar o mesmo conjunto de problemas. No entanto, algumas delas mostram-se mais eficientes se fornecidos os recursos apropriados. Uma dessas formas passa pelo uso do paralelismo e é nesse espírito que as redes neuronais têm um papel interessante e mesmo relevante.

Até há relativamente pouco tempo (início dos anos 90), as redes neuronais tinham sido observadas como ferramentas de aprendizagem. No entanto, vários trabalhos, no qual se inclui esta Tese, procuraram mostrar que o cenário não é tão restritivo e que existem caminhos abertos na área da computação simbólica (cf. [Pollack 87], [Maass 96], [Siegelmann e Sontag 91, 95], [Sun 95], [Siegelmann 99]).

Foi nesse sentido que orientámos o trabalho. Ao mostrar no capítulo 3 que o poder computacional de um determinado modelo de rede é equivalente ao das máquina de Turing, usando um sistema modular, foi aberto um caminho para a procura de procedimentos eficazes. Apesar desta demonstração não ser a primeira, dado já os trabalhos de Siegelmann e Sontag terem mostrado a universalidade destes sistemas (através da simulação de uma máquina de Turing universal), ela é relevante, primeiro porque o carácter modular da sua estrutura, segundo porque fornece um mecanismo operacional de construção de problemas específicos, e terceiro porque simplifica extremamente as estruturas neuronais capazes de controlar a execução de algoritmos.

Outra porta foi aberta, quando se mostrou no capítulo seguinte, que não era necessário construir redes específicas para problemas específicos (questão, aliás, bastante complexa). Era possível apresentar uma descrição em muito maior nível de abstracção, através de uma linguagem de programação com uma sintaxe e semântica muito semelhante às linguagens de programação comuns, que fosse capaz de produzir essas mesmas redes. Isso implica uma simplificação operacional extremamente relevante e fornece uma ferramenta capaz de tornar a tarefa em algo simples e, principalmente, automático.

A porta final (do ponto de vista do trabalho presente, dado que o conhecimento passa por abrir um número ilimitado de novas e desconhecidas portas) foi aberta com a criação de uma ponte entre o sistema computacional proposto para integração da aprendizagem em módulos de uma arquitectura semelhante (semelhante mas não igual, dado serem usadas funções de activação de segundo grau).

Esta ligação mostra que é possível criar sistemas neuronais complexos capazes de executar os dois tipos de computação, a computação simbólica e a computação sub-simbólica. Que a representação local (onde uma unidade de memória representa um objecto, um conceito ou uma hipótese) e a distribuída (onde cada unidade de memória participa da representação de um ou mais objectos, podendo intersectar-se e mesmo degradar o conhecimento assim sobreposto) podem coexistir na mesma arquitectura neuronal. Este aspecto de integração é considerado relevante pela comunidade, como se pode verificar em trabalhos recentes (cf. [Browne 98], [Li e Elliman 98], [Appoloni e Zoppis 99], [Medler *et al.* 99], [Taha e Ghosh 99], [Zang 99]), ou mesma numa edição de artigos especialmente dedicada a esta temática (cf. [Sun 95]).

Todos estes passos foram dados dentro de uma arquitectura neuronal, relativamente simples, reconhecida pela comunidade científica (as redes com funções de activação σ). Um dos principais pontos da originalidade do trabalho passou pela conexão destes sistemas com o aspecto plenamente modular das soluções propostas. Esta modularidade permite controlar a complexidade inerente e inevitável dos grandes sistemas, como expressar e salientar a sua correcção. A reutilização de módulos é também algo permissível, dado que as interacções deles com o ambiente externo e os outros módulos, é efectuado por canais de comunicação bem definidos e de fácil controle e gestão.

8.2. Trabalho Futuro

Podemos inferir, da experiência acumulada durante a preparação e execução desta Tese, que os seguintes pontos podem conter sementes de trabalho futuro e de resultados potencialmente interessantes.

Formalização e Integração

Quando foi realizada, no capítulo 5, a integração da computação simbólica e sub-simbólica sobre as redes-σ, ficou aberta a possibilidade de um salto qualitativo deste assunto para uma esfera mais formal, mais matemática.

A execução de uma rede pode ser vista como a iteração, ou evolução de uma órbita no espaço de fases definido pela arquitectura neuronal. O início da computação inicia-se em um determinado ponto desse espaço, o vector de activações (que define o estado actual da rede) evolui a cada instante para uma nova posição até atingir um dos estados finais (ou seja, entrar no intervalo de terminação do programa).

Como ambas as computações (simbólica e sub-simbólica) estão neste momento integradas nesta arquitectura neuronal, será possível, em principio, estudar estes dois conceitos aparentemente diversos numa estrutura matemática comum. Essa formalização poderia fornecer informação relevante tanto na comparação de programas nos dois conceitos de computação, bem como na classificação de vários algoritmos de aprendizagem tendo em conta o seu comportamento dinâmico dentro da estrutura neuronal considerada.

Unidades Heterogéneas de Activação

Todo o sistema de construção automática neuronal apresentado, utiliza redes homogéneas no que respeita à função de activação: todos os neurónios usam redes-σ. No entanto, não existe restrição ao uso de outras funções de activação dentro da mesma rede neuronal, criando assim redes heterogéneas de computação. O uso mais directo desta extensão do modelo, seria a implementação de algoritmos de aprendizagem mais complexos, como o algoritmo da retropropagação. Para isso, seriam utilizados neurónios com função de activação sigmoidal para realizar as partes dos cálculos onde a função sigmoidal é necessária.

No que respeita à activação, outras extensões são possíveis. Uma mais complexa, com o objectivo de diminuir o número de neurónios utilizados (procurando diminuir assim os problemas e demoras na comunicação sináptica de grandes redes), seria o uso de autómatos finitos na activação neuronal, onde consoante o valor da entrada sináptica, seria considerado um determinado comportamento, que mudaria, eventualmente, ao longo do tempo. Esta ideia, sem nenhuma inspiração biológica, procura ver cada neurónio como um

processador de informação sensível ao seu contexto, tanto em relação ao seu estado actual, bem como aos acontecimentos externos do seu passado mais recente (que determinariam a passagem para um novo estado e logo para uma nova função de activação). Existe algum trabalho realizado neste domínio. O modelo neuronal designado por *neuroid* é um neurónio programável com um mecanismo de alteração da sua estrutura interna, podendo assim, mudar o seu comportamento computacional, cf. [Wiedermann 99]. Em [Lopin 99] é estudado um modelo neuronal a partir do conjunto dos seus estados discretos possíveis.

Estruturas Neuronais Dinâmicas

Uma questão pertinente é a possibilidade de uso de redes de dimensão dinâmica, ou seja, capazes de alterar a sua arquitectura em tempo real de execução. Isso não acontece com o sistema apresentado, em que apenas é possível alterar a topologia da rede, através da modificação dos próprios pesos sinápticos. Estes mecanismos controlados apenas pelas próprias redes parecem-nos demasiado complexos para obter alguma efectividade no seu potencial uso. Porém, existindo algum suporte externo, nomeadamente nas capacidades de um *hardware* paralelo que executaria as redes compiladas, seria talvez admissível introduzir mecanismos de duplicação de redes existentes, para comportar múltiplas execuções do mesmo código ao mesmo tempo. Por exemplo, dado um módulo B, a execução de um comando FORK B, cria uma nova instância do módulo, que uma vez atribuídas as entradas e saídas específicas, executa de forma paralela. Isto permitiria uma implementação de mecanismos de recursão típicos das linguagens de programação tradicionais. Outros comandos relacionados incluiriam um MERGE, que tentaria fundir duas instâncias de um mesmo módulo, eliminando um dos conjuntos de entrada/saída ou mesmo partilhar o módulo resultado, usando um mecanismo de exclusão mútua, como existe no NETDEF; e um DELETE para destruir unidades neuronais já não utilizáveis.

8.3. Considerações Finais

Pela construção de propriedades e métodos modulares, pela construção recursiva de módulos individualmente simples, colectivamente poderosos e flexíveis, pela criação de processos específicos para tratamento e procura de soluções em ambientes de computação maciçamente paralela, foi obtido um sistema que é capaz de organizar aplicação entre descrições algorítmicas de problemas arbitrários e redes neuronais relativamente simples e homogéneas. Por tudo isto, pode afirmar-se que ficou diminuída a distância que separa a clássica Teoria da Computação do corpo de conhecimento agregado à Neurodinâmica.

Anexo A – Esquema da rede de uma função NETDEF

Como exemplo, apresentaremos a rede criada para a função booleana seguinte:

```
FUNCTION F (VAR x1 : INTEGER, x2 : INTEGER) IS BOOLEAN
  B;
```

em que B é um processo qualquer, e a chamada em questão F(a1, a2), é a 2ª chamada da função detectada no programa pelo compilador.

Cada chamada da função espera pela sua vez. Quando esse momento é chegado (F_2 é activado a 1), adquire o testemunho e retira-o do sistema (através do neurónio *begin*). Em seguida, a rede da função transfere os valores da chamada para os argumentos da função, e inicia o seu processo principal.

Quando termina, actualiza as variáveis associadas aos argumentos por variável (neste exemplo, somente o primeiro argumento), envia o resultado para o processo que a activou, e recoloca o testemunho no próximo neurónio (neurónio *end*) do sistema de estafetas. O neurónio *work* deixa passar a informação no fim do processamento da função.

Os neurónios dentro do tracejado mais o sistema de estafetas, fazem parte da estrutura da função e são únicos. O que está fora é repetido para cada chamada

da função, o que implica um custo de N+M+3 neurónios por chamada, sendo N o número de argumentos e M o número desses argumentos que são passados por variável.

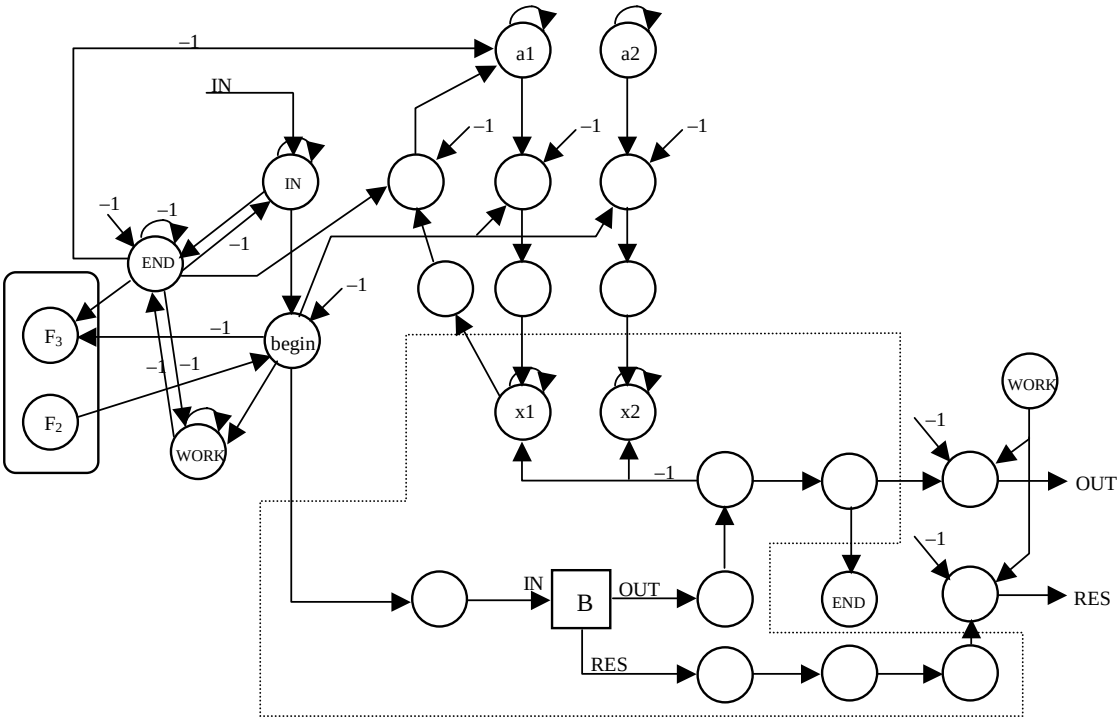


fig. A1 – Rede padrão para execução de funções.

Anexo B – Outros Processos NETDEF

Não Determinismo

O processo CHOOSE introduz não determinismo no NETDEF. CHOOSE escolhe de forma não determinista, um dos seus processos para executar.

choose ::= "CHOOSE" process ";" process { ";" process } "ENDCHOOSE".

O NETDEF possui um canal especial, denominado por GUESSB, que devolve ou zero ou um de forma aleatória. Tendo N processos, necessitamos de $K = \lceil \log_2 N \rceil$ bits de informação para decidir qual dos processos a escolher. Para isso, CHOOSE obtém K bits de GUESS, $M = b_1, ..., b_K$, para activar o processo $b_1 b_2 \ldots b_K$ (em notação binária). Se $M > N$, então CHOOSE tenta outra vez.

Seja o seguinte exemplo,

```
CHOOSE
  I1;
  I2;
  I3;
ENDCHOOSE;
```

A rede resultante seria a seguinte,

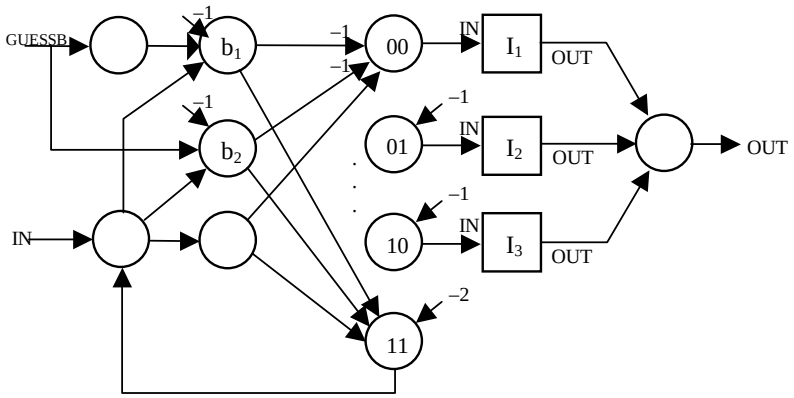


fig. B1 : escolha não determinista entre três processos.

Os pesos entre os neurónios b_i e os neurónios 01 e 10 não são mostrados por razões de clareza. Para cada processo, existe apenas uma sequência binária que o activa. É isso que determina o pendor dos neurónios marcados com as sequências binárias e os pesos sinápticos que os ligam aos neurónios b_i.

Inserção Directa de Redes-σ

Todas as redes apresentadas derivam do funcionamento dos seus comandos respectivos. No entanto, em certas situações, outras redes ou comandos podem ser importantes, tanto pela poupança no número de neurónios, como na rapidez de execução de uma dada rede neuronal.

O NETDEF permite a inserção de redes arbitrariamente complexas construídas pelo programador, desde que estas respeitem as regras de sincronização existentes em todas as redes da linguagem.

Os comandos ASM e ENDASM indicam ao compilador que o que está entre eles, são equações que descrevem uma rede neuronal. O formato EBNF de rede inserida desta forma é a seguinte:

Inserção directa de redes

net ::= “ASM” equation “;” { equation “;” } “ENDASM”.
equation ::= id “=” real “” id { { “+” | “-” } real [“*” id] }.*

De reparar que não existe controlo se o sistema está sincronizado ou não, isso é uma tarefa do programador. O que o sistema garante, é que o sinal de entrada é enviado para o X_{IN} e quando o processo termina, é enviado o sinal de terminação para o X_{OUT}. Isso é realizado, ligando a primeira equação ao neurónio entrada e a última equação ao neurónio saída. É possível atribuir e ler o conteúdo das variáveis que forem visíveis no contexto do sistema de equações. Por exemplo,

```
ASM
  XB(t+1)    = SIG( );
  XA(t+1)    = SIG( 1.0 * A + 1.0 * XB(t) – 0.97 );
  XE(t+1)    = SIG( 1.0 * XA(t) );
ENDASM;
```

Em termos gráficos,

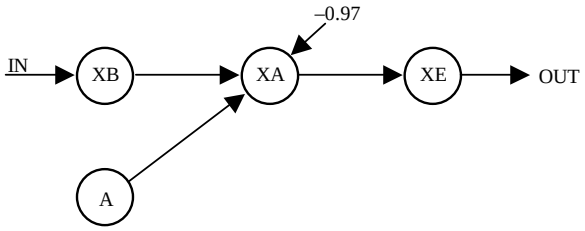


fig. B2 : rede construída pelo utilizador utilizando a variável A.

Comandos Guardados

guarda

Na medida em que cada módulo pode ser visto como uma entidade independente do resto da rede, é possível criar certos módulos que só têm a sua actividade iniciada quando um dado acontecimento ocorrer. Dir-se-á que esse módulo reagiu a um dado acontecimento, executando uma tarefa específica. Essa reacção pode provocar outras, e assim sucessivamente. Para isso, o NETDEF tem um novo tipo de variável denominada de guarda, que quando activada provoca a execução de uma ou mais redes neuronais.

guarded_process ::= "GUARD" process "WITH" id.

Uma variável guarda possuí dois neurónios, um para enviar o sinal de activação e outro para informar se todas as redes guardadas por ela estão inactivas ou não. Se houver pelo menos uma rede que está activa, diz-se que a guarda está

ocupada. A rede resultado de um processo guardado é a seguinte (os neurónios dentro do tracejado representam a guarda),

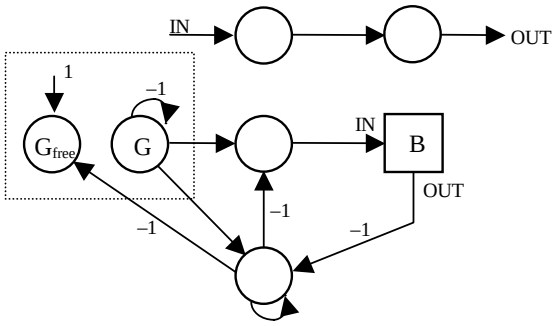


fig. B3 : rede do comando GUARD B WITH G.

Para iniciar as redes guardadas por uma determinada guarda G, utiliza-se o processo START G. A função booleana BUSY(G) indica se existe pelo menos um dos módulos guardados por G, que esteja activo.

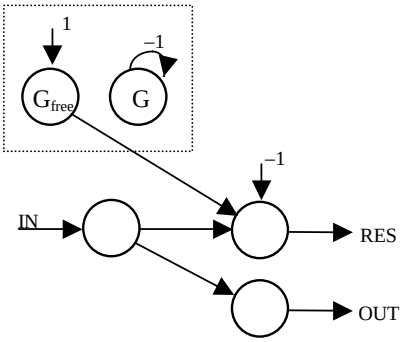


fig. B4 : rede do comando BUSY(G).

Se pretendermos esperar que os módulos, a que guarda diz respeito, terminem, utiliza-se o processo WAIT(G).

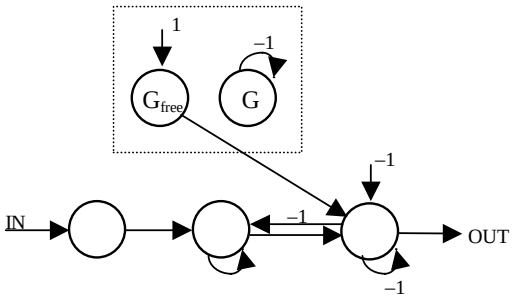


fig. B5 : rede do comando WAIT G.

Bibliografia

Alexander, J. and Mozer, M., *Template-based procedures for neural network interpretation*, **Neural Networks**, [12] 3, 1999, 479-498.

Alon, N., Dewdney, A., and Ott, T., *Efficient Simulation of Finite Automata by Neural Nets*, **Journal of the Association of Computing Machinery**, [38] 2, 1991, 495-514.

Alvares, L. and Sichman, J. – *Introdução aos Sistemas Multiagentes – XVI Jornada de Actualização em Informática,* In **XVII Congresso da Sociedade Brasileira de Computação**, 1998.

Andersen, A. and Rosenfeld, E., **Neurocomputing: Foundations of Research**, MIT Press, 1988.

Andersen, J., *A simple neural network generating an interactive memory*, **Mathematical Biosciences**, [14], 1972, 197-220.

Andonie, R., *The New Computational Power of Neural Networks*, **Neural Network World**, [4], 1996, 469-475.

Apolloni, B. and Zoppis, I., *Subsymbolically Managing Pieces of Symbolical Functions for Sorting*, **IEEE Transactions of Neural Networks**, [10] 5, 1999, 1099-1122.

Aybay, I, Cetinkaya, S., and Halici U., *Classification of Neural Network Hardware*, **Neural Network World**, [1], 1996, 11-27.

Balcázar, J., Díaz, J., and Gabarró, J., **Structural Complexity I**, Springer Verlag, 1995.

Balcázar, J., Díaz, J., and Gabarró, J., **Structural Complexity II**, Springer Verlag, 1995.

Barbosa, V., **Massively Parallel Models of Computation**, Elis Horwood, 1993.

Bein, V., *2D neural hardware versus 3D biological ones*, **ICSC/IFAC Symposyum**, Academic Press, 1998, 36-42.

Bennani, Y., *A Modular and Hybrid Connectionism System for Speaker Identification*, **Neural Computation**, [7], 1995, 791-798.

Bessière, P., Chams, A., and Chol, P., *MENTAL: A virtual machine approach to artificial neural networks programming*, In NERVES, ESPRIT B.R.A. project no 3049, 1991.

Block, H., *The Perceptron: a model for brain functioning. I*, **Reviews of Modern Physics**, [34], 1962, 123-135.

Boas, P., *Machine Models and Simulations*, **Algorithms and Complexity**, Elsevier, 1990, 1-65.

Boers, E., Kuiper, H., Happel, B., and Kuyper, I., *Designing Modular Artificial Neural Networks*, Technical Report 93-24, Leiden University, 1993.

Brooks, R., *How to build complete creatures rather than isolated cognitive simulators*, in K. VanLehn (ed.), **Architectures for Intelligence**, Lawrence Erlbaum Associates, Hillsdale, NJ, 1991, 225-239.

Browne, A., *Performing a symbolic inference step on distributed representations*, **Neurocomputing**, [19] 1-3, 1998, 23-34.

Caelli, T., Guan, L., and Wen, W., *Modularity in Neural Computing*, **Proceedings of the IEEE**, [87] 9, 1999, 1497-1518.

Card, H., Rosendahl, G., McNeill, D., and McLeod, R., *Competitive Learning Algorithms and Neurocomputer Architecture*, **Transactions on Computers**, [47] 8, 1998, 847-858.

Carrasco, R., Forcada, M., Valdés-Muñoz, M., and Ñeco, R., *Stable Encoding of Finite-State Machines in Discrete-Time Recurrent Neural Nets with Sigmoid Units*, **Neural Computation**, [12], 2000, 2129-2174.

Carreira, P., Rosa, M., Neto, J., and Costa, J., *Building a Neural Computer: A Compiler and Simulator for Partial Recursive Functions over Neural Networks*, Technical Report DI-98-8, Computer Science Department, University of Lisbon, 1998.

Casey, M., *The Dynamics of Discrete-Time Computation, with Application to Recurrent Neural Networks and Finite State Machines Extraction*, **Neural Computation**, [8] 6, 1996, 1135-1178.

Casey, M., *Correction to Proof that Recurrent Neural Networks can Robustrly Recognize only Regular Languages*, **Neural Computation**, [10], 1998, 1067-1069.

Chalmers, D., *A Computational Foundation for the Study of Cognition*, Department of Philosophy, University of California, 1997.

Chen, C., Honavar, V., *A Neural Network Architecture for Syntax Analysis*, **IEEE Transactions on Neural Networks**, [10] 1, 1999, 94-114.

Chester, M., **Neural Networks, A Tutorial**, Prentice Hall, 1993.

Chevtchenko, P., Fomine, D., Tchernikov, V., and Vixne, P., *Using of microprocessor NM6403 for neural net emulation*, disponível na Internet, 1997.

Codd, E. F., **Cellular Automata**, Academic Press, NY, 1968.

Cutland, N., **Computability, an Introduction to Recursive Function Theory**, Cambridge University Press, 1980.

DasGupta, B, Siegelmann, H., and Sontag, E., *On the Intractability of Loading Neural Networks*, **Theoretical Advances in Neural Computation and Learning**, Chowdhury, V., Siu, K., Orlitsky, A. (eds.), Kluwer Academic Publishers, 1994, 357-389.

Devaney, R., **An Introduction to Chaotic Dynamical Systems**, 2nd Ed., Addison-Wesley, 1989.

Dijkstra, E., *A Note on Two Problems in Connection with Graphs*, **Numerical Mathematics**, [1], 1959, 269-271.

Duong, T., Eberhardt, S., Daud, T., and Thakoor, A., *Learning in Neural Networks: VLSI Implementation Strategies*, **Fuzzy Logic and Neural Network Handbook**, Chen, C. (ed.), McGraw-Hill, 1996

Eberhart, R. and Dobbins, R., **Neural Networks PC Tools**, Academic Press, 1990.

Fahlman, S. and Lebiere, C., *The cascade-correlation learning architecture*, Carnegie Mellon Un., Technical Report CMU-CS-90-100, 1990.

Frasconi, P., Gori, M., and Sperduti, A., *A General Framework for Adaptive Processing of Data Structures*, **IEEE Transactions on Neural Networks**, [9] 5, 1998, 768-786.

Garzon, M. and Franklin, S., *Neural Computability*, **Proc. Third International Joint Conference on Neural Networks**, [2], 1989, 631-637.

Garzon, M., **Models of Massive Parallelism**, Springer, 1995.

Garzon, M. and Botelho, F., *Dynamical approximation by recurrent neural networks*, **Neurocomputing**, [29] 1-3, 1999, 25-46.

Gavaldà, R. and Siegelmann, H., *Discontinuities in Recurrent Neural Networks*, **Neural Computation**, [11] 3, 1999, 715-745.

Gibney, M. and Jennings, N., *Market Based Multi-Agent Systems for ATM Network Management*, **Proc. 4th Communications Networks Symposium**, Manchester, UK., 1997.

Giles, C., Miller, C., Chen, D., Chen, H., Sun, G., and Lee, Y., *Learning and extracting finite state automata with second-order recurrent neural networks*, **Neural Computation**, [4] 3, 1992, 393-405.

Goser, K., *Basic VLSI Circuits for Neural Networks*, **Neurocomputing**, NATO ASI Series [F68], Springer Verlag, 1990, 131-140.

Goudreau, M., Giles, C., Chakradhar, S., and Chen, D., *First-Order Versus Second-Order Single-Layer Recurrent Neural Networks*, **IEEE Transactions of Neural Networks**, [5] 3, 1994, 511-513.

Grossberg, S., *Adaptative pattern classification and universal recoding: I. Parallel development and coding of neural feature detectors*, **Biological Cybernetics**, [23], 1976, 121-134.

Gruau, F., Ratajszczak, J., and Wiber, G., *A neural compiler*, **Theoretical Computer Science**, [141] (1-2), 1995, 1-52.

Gunter, C. and Scott, D., *Semantic Domains*, **Formal Models and Semantics**, Elsevier, 1990, 633-674.

Guerra, H., **Poder Computacional das Redes Neuronais**, Dissertação apresentada na Faculdade de Ciências da Universidade de Lisboa para obtenção do grau de Mestre em Informática, 1997.

Handelman, D., Lane, S., and Gelfand, J., *Integrating Neural Networks and Knowledge-Based Systems for Intelligent Robotic Control*, **IEEE Control Systems Magazine**, April 1990, 77-87.

Happel, B. and Murre, J., *The Design and Evolution of Modular Neural Networks Architectures*, **Neural Networks**, [7], 1994, 985-1004.

Haykin, S., **Neural Networks – A Comprehensive Foundation**, 2nd ed., Prentice Hall, 1999.

Hebb, D., **The Organization of Behavior**, Wiley, 1949.

Hecht-Nielsen, R., **Neurocomputing**, Addison-Wesley, 1990.

Hein, J., **Theory of Computation: An Introduction**, James and Bartlett Publishers, 1996.

Hendler, J. and Dickens, L., *Integrating neural and expert reasoning: An example*, **Proceedings of AISB-91**, Leeds, UK, 1991.

Hrycej, T., **Modular Learning in Neural Networks**, John Wiley & Sons, 1992.

Hoare, C., *Communicating Sequential Processes*, **Communications of the ACM**, [21] 8, 1978, 666-676.

Honavar, V., *Symbolic Artificial Intelligence and Numeric Artificial Neural Networks: Towards a Resolution of the Dichotomy*, **Computational Architectures Integrating Neural and Symbolic Processes**, Kluwer Academic Publishers, 1995, 351-388.

Hopcroft, J. and Ullman, J., **Introduction to Automata Theory, Languages and Computation**, Addison-Wesley, 1979.

Hopfield, J., *Neurons with graded response have collective computational properties like those of two-state neurons*, **Proceedings of the National Academy of Sciences**, [81], 1984, 3088-3092.

Hopfield, J., *Neural networks and physical systems with emergent collective computational abilities*, **Proceedings of the National Academy of Sciences**, [79], 1982, 2554-2558.

Hübner, J. and Sichman, J., *SACI : Uma Ferramenta para Implementação e Monitoração da Comunicação entre Agentes*, **International Joint Conference IBERAMIA/SBIA 2000**, 2000, .

Hussain, T., *Modularity within Neural Networks*, Ph.D. Proposal, Queens Univ. – Ontario, Canada, 1995, 41 pgs.

Interactive Software Engineering Inc., **Eiffel®: the Language**, TR-EI 17/RM, 1989.

Jennings, B., Brennan, R., Gustavsson, R., Feldt, R., Pitt, J., Prouskas, K., and Quantz, J., *FIPA-compliant agents for real-time control of Intelligent Network traffic*, **Computer Networks**, **[**31] 19, 1999, 2017-2036.

Kandell, A. and Langholz, G., (eds.) **Hybrid Architectures for Intelligent Systems**, CRC Press, Boca Raton, Florida, 1992.

Karp, R. and Ramachandran, V., *Parallel Algorithms for Shared-Memory Machines*, **Algorithms and Complexity**, Elsevier, 1990, 869-941.

Knuth, D., **The Art of Computer Programming, Vol. 2 Seminumerical Algorithms**, Addison-Wesley, Reading Mass, 1981.

Koiran, P., **Puissance de calcul des réseaux de neurones artificiels**, Thèse présentée pour l'obtention du Diplôme de Doctorat, specialité de Informatique, 1993.

Koiran, P., Cosnard, M., and Garzon, M., *Computability with Low-Dimensional Dynamical Systems*, **Theoretical Computer Science**, [132], Elsevier, 1994, 113-128.

Koiran, P., *A family of universal recurrent networks*, **Theoretical Computer Science**, [168], 2, 1996, 473-480.

Koikkalainen, P., *MIND: A Specification Formalism for Neural Networks*, **Artificial Neural Networks**, Kohonen *et al*. (eds.), Elsevier, 1991, 579-584.

Kohonen, T., *Correlation matrix memories*, **IEEE Transaction on Computers**, [C-21], 1972, 353-359.

Kohonen, T., **Self-Organization and Associative Memory**, Springer-Verlag, 1987.

Kohonen, T., **Self-Organizing Maps – 2nd ed.**, Springer-Verlag, 1997.

Kramer, K., Minar, N., and Maes, P., *Tutorial: Mobile Software Agents for Dynamic Routing*, `http://www.media.mit.edu/nelson/research/routes/`, 1999.

Lacher, R. and Nguyen, K., *Hierarchical Architectures for Reasoning*, **Computational Architectures Integrating Neural and Symbolic Processes**, Kluwer Academic Publishers, 1995, 117-150.

Lamport, L. and Lynch, N., *Distributed Computing: Models and Methods*, **Formal Models and Semantics**, Elsevier, 1990, 1157-1199.

Lange, T., *A Structured Connectionist Approach to Inferencing and Retrieval*, **Computational Architectures Integrating Neural and Symbolic Processes**, Kluwer Academic Publishers, 1995, 69-115.

Lewis, H., and Papadimitriou, C., **Elements of the Theory of Computation**, Prentice Hall, 1981.

Leeuwen, J., **Handbook of Theoretical Computer Science - Formal Models and Semantics**, Elsevier, 1990.

Leeuwen, J., **Handbook of Theoretical Computer Science - Algorithms and Complexity**, Elsevier, 1990.

Lester, B., **The Art of Parallel Programming**, 1993, Prentice-Hall.

Li, B. and Elliman, D., *Fuzzy Rules Induction with Associative Memory Networks*, **ICSC/IFAC Symposyum**, Academic Press, 1998, 61-66.

Li, H., and Chen, C., *The Equivalence Between Fuzzy Logic Systems and Feedforward Neural Networks*, **IEEE Transactions on Neural Networks**, [11] 2, 2000, 356-365.

Lighttower, N., Allen, A., and Grant, H., *The Modular Map*, **Int. Joint Conference on Neural Networks**, [2], 1999, 851-856.

Lin, T., Horne, B., and Giles, C., *How embedded memory in recurrent neural network architectures helps learning long-term temporal dependencies*, **Neural Networks**, [11] 5, 1998, 861-868.

Lopin, V., *Analysis of a formal-logic model of neuron by means of a discrete state space*, **Automate Control for Computer Science**, [33] 6, 1999, 44-47.

Maass, W., *Lower Bounds for the Computational Power of Networks of Spiking Neurons*, **Neural Computation**, [8], 1996, 1-40.

Maass, W. and Orponen, P., *On the effect of Analog Noise in Discrete-Time Analog Computations*, **Neural Computation**, [10], 1998, 1071-1095.

Maass, W. and Sontag, E., *Analog Neural Nets with Gaussian or Other Common Noise Distributions Cannot Recognize Arbitrary Regular Languages*, **Neural Computation**, [11], 1999, 771-782.

Mahoney, J., J., *Combining Symbolic and Neural Learning to Revise Probabilistic Rule Bases*, **Artificial Intelligence**, [92], 1992, 189.

McCulloch, W. and Pitts, W., *A logical calculus of the ideas immanent in nervous activity*, **Bulletin of Mathematical Biophysics**, 5, 1943, 115-133.

Medler, D., McCaughan, D., and Dawson, M., *When local isn't enough: Extracting Distributed Rules from Networks*, **Int. Joint Conference on Neural Networks**, [2], 1999, 1174-1179.

Miikkulainen, R., Subsymbolic Parsing of Embedded Structures, **Computational Architectures Integrating Neural and Symbolic Processes**, Kluwer Academic Publishers, 1995, 153-186.

Minsky, M. and Papert, S., **Perceptrons**, MIT Press, 1969.

Mitra, S. and Hayashi, Y., *Neuro-Fuzzy Rule Generation Survey in Soft Computing Framework*, **IEEE Transactions on Neural Networks**, [11] 3, 2000, 748-768.

Moore, C., *Unpredictability and Undecidability in Dynamical Systems*, **Physical Review Letters**, Vol. 64, 20, American Physical Society, 1990, 2354-2357.

Moore, C., *Generalized shifts: unpredictability and undecidability in dynamical systems*, **Nonlinearity**, 4, 1991, 199-230.

Moore, R., Klauner, B., Waldschmidt, K., *Compiler Technology for Two Novel Computer Architectures*, disponível na Internet, 1998.

Moses, P., *Denotational Semantics*, **Formal Models and Semantics**, Elsevier, 1990, 575-632.

Neto, J., Costa, J., and Coelho, H., *Lower Bounds of a Computational Power of a Synaptic Calculus*, **Lecture Notes in Computer Science – 1240**, Springer-Verlag, 1997, 340-348.

Neto, J., Siegelmann, H., Costa, J., and Araujo, C., *Turing Universality of Neural Nets (revisited)*, **Lecture Notes in Computer Science – 1333**, Springer-Verlag, 1997, 361-366.

Neto, J., Siegelmann, H., and Costa, J., *On the Implementation of Programming Languages with Neural Nets*, **First International Conference on Computing Anticipatory Systems,** CHAOS, [1], 1998, 201-208.

Neto, J., Siegelmann, H., and Costa, J., *Information Coding and Neural Computing*, **Advances in A.I. and Engineering Cybernetics, Vol. IV: Systems Logic & Neural Networks**, IIAS, [10], 1998, 76-80.

Neto, J. and Costa, J., *Building Neural Net Software*, Technical Report DI-99-05, Computer Science Department, University of Lisbon, 1999. Disponível em *www.di.fc.ul.pt/biblioteca/tech-reports*.

Neto, J., Costa, J., and Ferreira, A., *Merging Sub-symbolic and Symbolic Computation*, In H. Bothe and R. Rojas (eds), **Proceedings of the Second International ICSC Symposium on Neural Computation (NC'2000)**, 329-335. ICSC Academic Press, 2000.

Neto, J., Siegelmann, H., and Costa, J., *Symbolic Processing in Neural Networks*, aceito para publicação no **Journal of Brasilian Computer Society**, 2001.

Neto, J., Coelho, H., and Ferreira, A., *Competitive Learning and Symbolic Computation Integration*, **5º Congresso Brasileiro de Redes Neurais**, 19-24, 2001.

Neto, J., Costa, J., Carreira, P., and Rosa, M., *A compiler and simulator for partial recursive functions over neural networks*, artigo a submeter, 2001.

von Neumann, J., *First draft of a report on the EDVAC*, **The Origins of Digital Computers: Selected Papers**, B. Randall (ed.), Springer, 1945/82. Citado em Anderson, 88.

von Neumann, J., **The Computer and the Brain**, Yale Univ. Press, 1958.

von Neumann, J., **Theory of Self-reproducing Automata**, Illinois Univ. Press, 1966.

Newell, A., **Unified Theories of Cognition**. Harvard, 1990.

Oja, E., *A simplified Neuron Model as a Principal Component Analyser*, **Journal of Math. Biology**, [15], 1982, 267-273.

Oja, E. and Kohonen, T., *The subspace learning algorithm as a formalism for pattern recognition and neural networks*, **Proc. IEEE International Conference of Neural Networks**, San Diego, CA, 1988, 277-284.

Omlin, C. and Giles, C., *Extraction of Rules from Discrete-time Recurrent Neural Networks*, **Neural Networks**, [9] 1, 1996, 41-52.

Omlin, C. and Giles, C., *Constructing Deterministic Finite-State Automata in Recurrent Neural Networks*, **Journal of the Association of Computing Machinery**, [43] 6, 1996, 937-972.

Omlin, C., Thornber, K., and Giles, C., *Fuzzy Finite-State Automata can be Deterministically Encoded into Recurrent Neural Networks*, **IEEE Transactions of Fuzzy Systems**, [6] 1, 1998, 76-89.

Orponen, P., *Computational Complexity of Neural Networks: A Survey*, **on Proceedings of the 17th International Symposium on Mathematical Foundations of Computer Science, Lectures Notes on Computer Science 692**, 1992, 50-61.

Osório, F. and Amy, B., *INSS: A hybrid system for constructive machine learning*, **Neurocomputing**, [28], 1999, 191-205.

Pan, V., *How to Multiply Matrices Faster*, **Lecture Notes in Computer Science 179**, Springer Verlag, 1982, 31-45.

Paolo, I. and Kuhn, G., *Digital Systems for Neural Networks*, in Papamichalis, P. and Kerwin, R., eds., **Digital Signal Processing Technology**, Critical Reviews Series CR57 Orlando, FL: SPIE Optical Engineering, 1995, 314-45.

Patel, A., Prouskas, K., Barria, J., and Pitt, J., *Load Control Using a Competitive Market-Based Multi-agent System*, **Lecture Notes in Computer Science 1774**, 2000, 239-254 ,

Pippenger, N., *Communication Networks*, **Algorithms and Complexity**, Elsevier, 1990, 805-833.

Pollack, J., *On Connectionist Models of Natural Language Processing*, Ph.D. Thesis, Computer Science Dept., Univ. Illinois, Urbana, 1987. Citado em Siegelmann, 95.

Pollack, J., *Recursive Distributed Representations*, **Artificial Intelligence**, [46], 1990, 77-106.

Post, E., *Formal Reductions of the General Combinatorial Decision Problem*, **American Journal of Mathematics**, [65], 1943, 197-268.

Prouskas, K., Patel, A., Pitt, J., and Barria, J., *A Multi-agent System for Intelligent Network Load Control Using a Market-based Approach*, **The Third International Conference on Multi-Agent Systems (ICMAS '98)**, 1998, 231-238.

Ramacher, U. *et al*., *Design of a First Generation Neurocomputer*, in **VLSI Design of Neural Networks**, Kluwer Academic Publishers, 1991.

Ramacher, U., **SYNAPSE-1 -- A General Purpose Neurocomputer**, SIEMENS AG, 1994.

Rosenblatt, F., *The perceptron: a probabilistic model for information storage and organization in the brain*, **Psychological Review**, [65], 1958, 368-408.

Rojas, R., **Neural Networks: A Systematic Introduction**, Springer, 1995.

Rose, D. and Whitten, G., *A Recursive Analysis of Dissection Strategies*, **Sparse Matrix Computations**, D. Rose (ed.), Academic Press, 1976, 59-84.

Rumelhart, D., Hinton, G., and Williams, R., *Learning internal representation by error propagation*, **Parallel Distributed Processing: Explorations in the Microstructures of Cognition**, MIT Press, 1986, 318-362.

Rumelhart, D., Hinton, G., and Williams, R., *Learning representations by back-propagation errors*, **Nature**, [323], 1986, 533-536.

Rumelhart, D. and McClelland, J., **Parallel Distributed Processing: Explorations in the Microstructures of Cognition**, [1], MIT Press, 1986.

Russel, S. and Norvig., P., **Artificial Intelligence - A Modern Approach**. Prentice Hall Series in AI, 1995.

Sangiovanni-Venticelli, A., *Optimization for Sparse Systems*, **Sparse Matrix Computations**, D. Rose (ed.), Academic Press, 1976, 97-110.

Santos, E. and Zuben, F., *Improved 2nd order training algorithm for globally and partially recurrent neural networks*, **Int. Joint Conference on Neural Networks**, [1], 1999, 502-505.

Schoonderwoerd, R., Holland, O., Bruten, J., and Rothkrantz, L., *Ant-based Load Balancing in Telecommunications Networks*, **Adaptive Behavior**, [5] 2, 1997, 169–207.

Schutter, E., *A consumer guide to neuronal modelling software*, **Trends in Neurosciences**, [15], Elsevier Science Publishers, 1992, 462-464.

Setiono, R. and Liu, H., *Symbolic Representation of Neural Networks*, **Computer**, Março 1996.

Setiono, R., *Extracting M-of-N Rules from Trained Neural Networks*, **IEEE Transactions on Neural Networks**, [11] 2, 2000, 512-519.

SGS-THOMSON, **Occam® 2.1 Reference Manual**, 1995.

Sharkey, N. and Jackson, S., An Internal Report for Connectionists, **Computational Architectures Integrating Neural and Symbolic Processes**, Kluwer Academic Publishers, 1995, 223-244.

Shepherd, G., **Neurobiology**, 3rd ed., Oxford University Press, 1994.

Sheperdson, J. and Sturgis, H., *Computability of Recursive Functions*, **J. Assoc. Computing Machinery**, [10], 1963, 217-255.

Siegelmann, H., *Foundations of Recurrent Neural Networks*, Technical Report DCS-TR-306, Rutgers University, 1993.

Siegelmann, H. and Sontag, E., *Analog Computation via Neural Networks*, **Theoretical Computer Science**, [131], Elsevier, 1994, 331-360.

Siegelmann, H. and Sontag, E., *On the Computational Power of Neural Nets*, **Journal of Computer and System Sciences**, [50] 1, Academic Press, 1995, 132-150.

Siegelmann, H., *On NIL: The Software Constructor of Neural Networks*, **Parallel Processing Letters**, [6] 4, World Scientific Publishing Company, 1996, 575-582.

Siegelmann, H. and Giles, C., *The complexity of language recognition by neural networks*, **Neurocomputing**, [15] 3-4, 1997, 327-345.

Siegelmann, H., Horne, B., and Giles, C., *Computational capabilities of recurrent NARX neural networks*, **IEEE Transactions on Systems, Man and Cybernetics**, [27] 2, 1997, 208-227.

Siegelmann, H., **Neural Networks and Analog Computation, Beyond the Turing Limit**, Birkhäuser, 1999.

SIEMENS, **SYNAPSE3•PC: SynUse•Base Programming Handbook**, MediaInterface Dresden GmbH, 1998.

Smolensky, P., *On the proper treatment of connectionism*, **Behavioral and Brain Sciences**, [11], Cambridge Univ. Press, 1988, 1-74.

Smullyan, R., **Diagonalization and Self-Reference**, Oxford Science Publishers, 1994.

Soucek, B. and Soucek, M., **Neural and Massively Parallel Computers: The 6th Generation**, John Wiley & Sons, 1988.

Sperduti, A., *On the Computational Power of Recurrent Neural Networks for Structures*, **Neural Networks**, [10] 3, 1997, 395-400.

Subramanian, D., Druschel, P., and Chen, J., *Ants and Reinforcement Learning: A Case Study in Routing in Dynamic Networks*, **Proc. of IJCAI'97**, 1997, 832-838.

Sun, G., Chen, H., Lee, Y., and Giles, C., *Turing equivalence of neural networks with second-order connections weights*, **Proc. International Joint Conference on Neural Networks**, IEEE, 1991.

Sun, R., *A Two-Level Hybrid Architecture for Structuring Knowledge for Commonsense Reasoning*, **Computational Architectures Integrating Neural and Symbolic Processes**, Kluwer Academic Publishers, 1995, 247-281.

Sun, R. and Bookman, L. (eds.), **Computational Architectures Integrating Neural and Symbolic Processes: A perspective on the State of the Art**, Kluwer Academic Publishers, 1995.

Sun, R., *Knowledge Extraction from Reinforcement Learning*, **Int. Joint Conference on Neural Networks**, [4], 1999, 2555-2559.

Taha, I. and Ghosh, J., *Symbolic Interpretation of Artificial Neural Networks*, **IEEE Transactions on Knowledge and Data Engineering**, [11] 3, 1999, 448-463.

Tan, A., *Cascade ARTMAP: Integrating Neural Computation and Symbolic Knowledge Processing*, **IEEE Transactions on Neural Networks**, [8] 2, 1997, 237-250.

Tennenhouse, D., Smith, J., Sincoskie, W., Wetherall, D., and Minden, G., *A Survey of Active Network Research*, **IEEE Communications Magazine**, [35] 1, 1997, 80-86.

Tiño, P. and Sadja, J., *Learning and Extracting Initial Mealy Automata with a Modular Neural Network Model*, **Neural Computation**, [7], 1995, 822-844.

Towell, G, *Symbolic Knowledge and Neural Networks: Insertion, refinement and extraction*, Ph.D. Thesis, Univ. of Wisconsin, Madison, Comp. Science Dept., 1991.

Tsukimoto, H., *Extracting Rules from Trained Neural Networks*, **IEEE Transactions on Neural Networks**, [11] 2, 2000, 377-389.

United States Dept. of Defence, **Reference Manual for the Ada® Programming Language**, American National Standards Institute Inc., 1983.

Valiant, L., *General Purpose Parallel Architectures*, **Algorithms and Complexity**, Elsevier, 1990, 943-971.

Verleysen, M. and Jespers, P., *An Analog VLSI Architecture for Large Neural Networks*, **Neurocomputing**, NATO ASI Series [F68], Springer Verlag, 1990, 141-144.

Wang, D., *Stable Neurodynamics and Symbolic Computation*, **RISC-Linz Report Series**, 90-38, RISC-Linz Faculty, 1990.

Wang, D., *Computer Algebra and Neurodynamics*, **RISC-Linz Report Series**, 91-56, RISC-Linz Faculty, 1991.

Wang. D., *Project Report on Symbolic Computation for Neural Networks*, **RISC-Linz Report Series**, 92-60, RISC-Linz Faculty, 1992.

Wang, Z. and Crowcroft, J., *Quality-of-Service Routing for Supporting Multimedia Applications*, **IEEE Journal on Selected Areas in Communications,** [14] 7, 1996.

Watt, D., **Programming Language Concepts and Paradigms**, Prentice-Hall International Series in Computer Science, 1990.

Wei-Ling L., Prasanna, L., and Przytula, K., *Algorithmic Mapping of Neural Network Models onto Parallel SIMD Machines*, In **IEEE Transactions on Computers**, [40] 12, 1991, 1390-1401.

Wellman, M., *A Market-Oriented Programming Environment and its Applications to Distributed Multicommodity Flow Problems*, **Journal of Artificial Intelligence Research**, [1] 1, 1993, 1-22.

Werbos, P., *Beyond regression: new tools for prediction and analysis in the behavioral sciences*, Ph.D. Thesis, Harvard Univ., Cambridge, 1974. Citado em Anderson, 88.

White, T. and Pagurek, B., *Towards Multi-Swarm Problem Solving in Networks*, **The Third International Conference on Multi-Agent Systems (ICMAS '98)**, 1998, 333-340.

White, T. and Pagurek, B., *Distributed Fault Location in Networks Using Learning Mobile Agents*, **Lecture Notes in Computer Science 1733**, 1999, 182-196

Widrow, B. and Hoff, M., *Adaptative switching circuits*, **IRE WESCON Convention Record**, IRE, 1960, 96-104.

Wiedermann, J., *Computational Power of Neuroidal Nets*, **SOFSEM'99: Theory and Practice of Informatics, Lecture Notes on Computer Science 1725**, Springer Verlag, 1999, 479-487.

Willmott, S. and Faltings, B., *Active Organisations for Routing*, **Lecture Notes in Computer Science Series 1653**, 1999, 262-273.

Willmott, S. and Faltings, B., *The Benefits of Environment Adaptive Organisations for Agent Coordination and Network Routing Problems*, **The Fourth International Conference on Multi-Agent Systems (ICMAS-2000)**, 2000, 333-340.

Willmott, S., Frei, C., Faltings, B., and Calisti, M., *Organisation and Coordination for On-line Routing in Communications Networks*, **Software Agents for Future Communication Systems**, Springer Verlag, 1999, 130-159.

Wilson, A. and Hendler, J., *Linking Symbolic and Subsymbolic Computing*, Technical Report, Dept. of Computer Science, University of Maryland, 1993.

Wolpert, D., *A computationally universal field computer which is purely linear*, Technical Report LA-UR-91-2937, Los Alamos National Laboratory, Los Alamos, NM, 1991.

Yang, M. and Ahuja, N., *A data partition method for parallel SOM*, **Int. Joint Conference on Neural Networks**, [3], 1999, 1929-1933.

Zang, Y., *On logical semantics of hybrid symbolic neural nets for commonsense reasoning*, **Int. Joint Conference on Neural Networks**, [1], 1999, 502-505.

Zhan, F., *Three Fastest Shortest Path Algorithms on Real Road Networks: Data Structures and Procedures*, **Journal of Geographic Information and Decision Analysis**, [1] 1, 1997, 69-82.