Uma Especificação Formal Orientada a Objetos de Redes Neurais Artificiais [1]

Patrícia Duarte de Lima Machado
Silvio Romero de Lemos Meira *Edson Costa de Barros Carvalho Filho*
Herman Martins Gomes

Universidade Federal de Pernambuco
Departamento de Informática
Caixa Postal 7851, 50.732-970, Recife, PE, Brasil
Tel.: +55 (81) 271-8430, Fax: +55 (81) 271-4925
E-mail: {pdlm, srlm, ecdbcf, hmg}@di.ufpe.br

Sumário

Este artigo apresenta uma especificação de redes neurais artificiais usando a linguagem MooZ. *Por ser uma linguagem de especificação formal orientada a objetos,* MooZ *possibilita a elaboração de uma descrição gradual e precisa das propriedades de redes neurais, baseada na reusabilidade de definições. Reutilizando classes abstratas, subclasses são definidas a medida que novas características precisam ser incorporadas. A especificação produzida pode ser aplicada no processo de desenvolvimento de software para simulação de redes neurais.*

Palavras-chave: *Redes neurais, especificações formais, orientação a objetos.*

Abstract

This paper shows a specification of artificial neural networks by using the MooZ *language. As it is an object oriented formal specification language,* MooZ *makes possible the elaboration of a precise and gradual description of neural networks properties, based on the reusability of definitions. By reusing abstract classes, subclasses are defined as new features of neural networks should be added. The produced specification may be applied to the development process of neural network simulation software.*

Keywords: *Neural networks, formal specifications, object orientation.*

1 Introdução

Redes neurais artificiais são modelos abstratos da atividade do cérebro humano. São compostas por um conjunto de elementos, os neurônios artificiais, organizados em uma rede de conexões ou topologia. Alguns métodos são utilizados para codificar e recuperar informações da rede. A codificação corresponde a um processo conhecido como aprendizagem, cujo objetivo é fazer com que a rede passe a ter um comportamento desejado.

Simulação de redes neurais é uma atividade essencial em pesquisas na área. Permite a análise do comportamento das redes, que é feita através da geração de estatísticas, do acompanhamento de sua evolução durante o processo de treinamento e das suas respostas quando são fornecidas entradas externas. A simulação pode ser feita através de

[1] Este trabalho foi desenvolvido com o apoio do CNPq.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

um software que é executado em um computador convencional ou de um dispositivo de hardware específico.

Redes neurais são tolerantes a falhas. no sentido de que a remoção de um ou mais neurônios ou conexões da rede pode não resultar em uma perda de conhecimento. A-pesar disso. software para redes neurais deve ser confiável. visto que mínimos erros de programação podem compromenter. por exemplo. o comportamento de um algoritmo de aprendizagem. Geralmente. é de grande porte e de alta complexidade: algumas aplicações podem possuir mais de 10000 neurônios interconectados. o que tende a crescer de uma ordem de magnitude em um futuro próximo. Ambientes de simulação são pacotes de software que dão suporte à construção e análise de redes neurais usando diferentes paradigmas. Têm. como principal objetivo, diminuir o tempo e o esforço envolvidos no processo de simulação. Assim. a área de redes neurais pode ser um dos pontos focais de desenvolvimento de software em larga escala. o que a torna um alvo óbvio para aplicação de teorias, métodos. técnicas e ferramentas associadas à Engenharia de Software como um todo.

A teoria de redes neurais é derivada de muitas disciplinas: psicologia, matemática. neuro-fisiologia, física. biologia e lingüística. Representa uma universalização destas ciências em busca de um objetivo comum - os sistemas inteligentes modelados a partir de estruturas computacionais simples: os neurônios. Devido a esta interdicisplinaridade, a aplicação de métodos e técnicas para o desenvolvimento de software confiável e que possa ser reutilizado em outros experimentos é reduzida. Normalmente, apenas a fase de programação é considerada. tornando o processo de construção de software um processo puramente artesanal [6]. Em alguns simuladores como o RCS [3], a simulação é descrita usando a linguagem de programação C.

Neste trabalho. é dada uma especificação formal orientada a objetos de redes neurais artificiais, utilizando a linguagem MooZ. O objetivo é que esta especificação possa ser utilizada em processos de desenvolvimento rigoroso de software para simulação de redes neurais artificiais, tal que produtos confiáveis e evolutíveis possam ser desenvolvidos. A especificação produzida está sendo utilizada no desenvolvimento de um ambiente para simulação de redes neurais artificiais [7].

A seção seguinte apresenta uma breve descrição da linguagem MooZ. Na seção posterior, é mostrada a estrutura da especificação produzida e a especificação de algumas classes. Por fim, é dada a conclusão sobre o trabalho realizado.

2 A Linguagem MooZ

MooZ (*Modular Object-Oriented Z*) [8] é uma extensão orientada a objetos da linguagem de especificação formal baseada em modelos Z [12]. A especificação de um sistema em MooZ é representada por uma classe cuja definição possivelmente pode depender da especificação de subsistemas ou subclasses. A definição completa da linguagem é dada em [9].

A estrutura geral da especificação de uma classe em MooZ é mostrada na figura 1. Todas as cláusulas são opcionais. mas a ordem é obrigatória. A cláusula givensets

compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

introduz os tipos indefinidos. que servem para representar um objeto que não precisa de um modelo em um certo nível de abstração. Os tipos indefinidos também podem ser usados para parametrizar uma classe. tornando-a genérica. A estrutura hierárquica entre as classes é estabelecida na cláusula superclasses. Definições públicas e privadas para os clientes da classe são introduzidas a partir da cláusula private ou public. A cláusula constants é usada para definir constantes globais. Os componentes de estado e os invariantes de uma classe são os da superclasse mais aqueles introduzidos na cláusula state. Na cláusula initialstates. são definidas as operações de inicialização e. na cláusula operations, as operações da classe.

Comentários podem ser incluídos em qualquer ponto da especificação. Qualquer parágrafo de Z (*given sets*. esquemas. descrições axiomáticas. abreviações. *free types*. restrições) pode ser utilizado em uma especificação de classe. MooZ ainda inclui duas novas estruturas sintáticas. o esquema anônimo e a operação semântica.

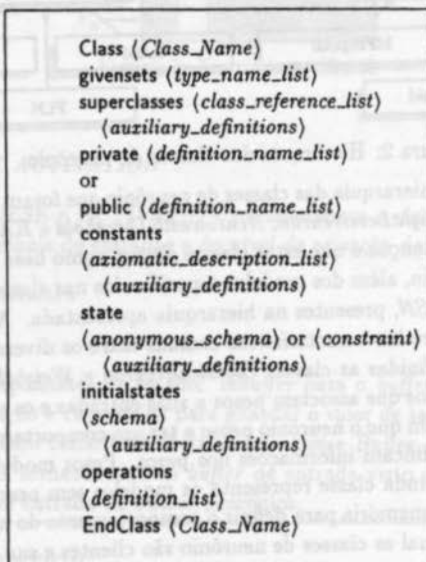
Class ⟨*Class_Name*⟩
givensets ⟨*type_name_list*⟩
superclasses ⟨*class_reference_list*⟩
⟨*auxiliary_definitions*⟩
private ⟨*definition_name_list*⟩
or
public ⟨*definition_name_list*⟩
constants
⟨*axiomatic_description_list*⟩
⟨*auxiliary_definitions*⟩
state
⟨*anonymous_schema*⟩ **or** ⟨*constraint*⟩
⟨*auxiliary_definitions*⟩
initialstates
⟨*schema*⟩
⟨*auxiliary_definitions*⟩
operations
⟨*definition_list*⟩
EndClass ⟨*Class_Name*⟩

Figura 1: Estrutura de uma classe em MooZ

3 A Especificação

A especificação está dividida em duas partes: especificação de neurônios artificiais. dada na subseção 3.1. e especificação de redes neurais artificiais. dada na subseção 3.2. São especificadas apenas algumas das classes mencionadas. Algumas especificações de esquemas e descrições axiomáticas também são omitidas por questão de espaço. A especificação completa de todas as classes pode ser encontrada em [4]. Não sendo um tutorial. este texto assume conhecimento prévio em especificações formais. orientação a objetos e redes neurais. sem deixar de ser auto-contido. onde é estritamente necessário que tal aconteça.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

3.1 Neurônios Artificiais

Os neurônios artificiais [14, 11, 5] são modelos que tentam imitar o comportamento do neurônio biológico [13] a fim de exibir algumas de suas propriedades. Por serem unidades computacionais simples, não levam em consideração a complexidade envolvida na atividade do sistema biológico, assegurando seu *status* como modelos e tornando possível seu fácil entendimento.

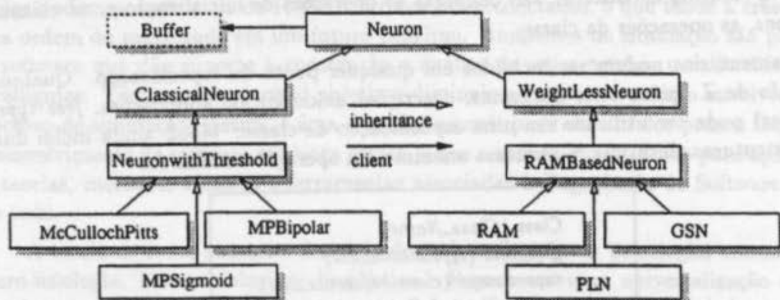

Figura 2: Hierarquia das classes de neurônio.

A figura 2 mostra a hierarquia das classes de neurônio que foram especificadas. *Neuron*, *ClassicalNeuron*, *WeightLessNeuron*, *NeuronwithThreshold* e *RAMBasedNeuron* são as classes abstratas. A intenção é que estas classes sirvam como base para a definição de outros modelos de neurônio, além dos modelos especificados nas classes *McCulloch-Pitts*, *Sigmoid*, *RAM*, *PLN* e *GSN*, presentes na hierarquia apresentada. A classe *Neuron* é a classe mais abstrata e introduz características comuns entre os diversos modelos. Como subclasses diretas, são definidas as classes *ClassicalNeuron* e *WeightLessNeuron*. A primeira representa os modelos que associam pesos a suas entradas e os valores destes pesos são alterados para fazer com que o neurônio passe a ter um comportamento desejado [14], ou seja, estes modelos codificam informações nos pesos. Pesos modelam as sinapses do neurônio biológico. A segunda classe representa os modelos sem peso, os quais armazenam valores em células de memória para definir o comportamento do neurônio [1]. *Buffer* é uma classe auxiliar da qual as classes de neurônio são clientes e sua especificação pode ser encontrada em [4].

3.1.1 Classe *Neuron*

A classe *Neuron* define um modelo abstrato de neurônio artificial o qual engloba características comuns encontradas em modelos reais. Este modelo possui uma seqüência de terminais de entrada, realiza um processamento interno e gera valores de saída. A figura 3 ilustra um neurônio artificial.

A geração de um valor de saída ou ativação do neurônio é ordenada por uma entidade do mundo externo. A função que determina o valor de saída utiliza apenas informações locais, como o nível de ativação (representado por *activation* na figura 3). Este é computado utilizando os valores dos terminais de entrada durante a ativação do neurônio.

Compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

Os valores dos terminais de entrada nem sempre são recebidos ao mesmo tempo, por ocasião da ativação do neurônio. É possível que apenas os valores de alguns terminais sejam recebidos entre duas ativações consecutivas. Por este motivo, estes valores não podem ser modelados como a entrada externa desta operação. Deve existir um *buffer* para armazenar todo valor que for sendo recebido. Quando for necessário gerar a saída do neurônio, são utilizados os valores armazenados neste *buffer*. O valor de saída também deve ser mantido no estado da classe, visto que precisa ser constantemente consultado e, entre duas consultas, alguma informação local pode ser alterada. Isto pode fazer com que dois valores diferentes de saída sejam devolvidos antes da próxima ativação do neurônio.

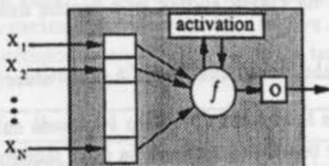

Figura 3: Neurônio artificial.

Class *Neuron*

givensets *IMPULSE*, *ACTIVATION*

Os *given sets IMPULSE* e *ACTIVATION* parametrizam a classe com relação aos tipos dos valores dos terminais de entradas e do nível de ativação do neurônio.

private *ComputeActivation*

state

São definidos 3 componentes de estado: *inbuffer* para o *buffer* de entradas, *activation* para o nível de ativação e *currentOut* para guardar o valor de saída corrente do neurônio. O primeiro é modelado como uma instância da classe *Buffer*. O tipo de *currentOut* é o mesmo dos valores armazenados no *buffer* de entrada visto que pode ser usado como valor de terminais de entrada de outros neurônios.

inbuffer : *Impulse*
activation : *ACTIVATION*
currentOut : *IMPULSE*

$Impulse == Buffer(VALUE \backslash IMPULSE)$

initialstates

<u>*CreateNeuron*</u> utiliza a operação *CreateBuffer* da classe *Buffer* para dar o valor inicial de *inbuffer*. *activation* e *currentOut* são inicializados por entradas externas.

CreateNeuron
$\Delta(inbuffer, activation, currentOut)$
inbuffer CreateBuffer(*values*?*inbuffervalues*?)
activationvalue? : *ACTIVATION*
currentOutvalue? : *IMPULSE*
$activation' = activationvalue? \wedge currentOut' = currentOutvalue?$

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

InitNeuron especifica o estado inicial onde o ***buffer*** de entradas está vazio.

InitNeuron
$CreateNeuron(inbuffervalues? \setminus inbufferempty)$
$inbufferempty : Seq\ IMPULSE$

$inbufferempty = \langle\rangle$

operations

ChangeInbuffer muda os valores de algumas posições do ***buffer*** de entrada. É usada a operação *ChangeValues* da classe *Buffer*, que recebe um mapeamento de posições do *buffer* para valores.

$$ChangeInbuffer \mathrel{\hat=} \Delta(inbuffer) \wedge inbuffer\ ChangeValues(values? \setminus inbuffervalues?)$$

IncInBuffer adiciona novas locações ao final do ***buffer*** de entrada. É utilizada a operação *Insert* da classe *Buffer* que recebe como entrada uma seqüência de valores para inicializar as novas locações.

$$IncInBuffer \mathrel{\hat=} \Delta(inbuffer) \wedge inbuffer\ Insert(values? \setminus inbuffervalues?)$$

DecInBuffer remove locações do ***buffer*** de entrada especificadas por uma entrada externa. É utilizada a operação *Remove* da classe *Buffer*.

$$DecInBuffer \mathrel{\hat=} \Delta(inbuffer) \wedge inbuffer\ Remove(locations? \setminus inbufferlocations?)$$

ComputeActivation dá um novo valor para o nível de ativação do neurônio.

ComputeActivation
$\Delta(activation)$
$a : ACTIVATION$

$activation' = a$

ActivateNeuron define a ativação de um neurônio onde novos valores para o nível de ativação e para a saída são computados.

ActivateNeuron
$\Delta(currentOut)$
$ComputeActivation$
$o : IMPULSE$

$currentOut' = o$

EndClass *Neuron*.

3.1.2 Classe *ClassicalNeuron*

A classe *ClassicalNeuron* representa os modelos de neurônio no qual as informações de aprendizagem são armazenadas nos pesos de suas conexões com outros neurônios. Um neurônio desta classe possui um peso associado a cada terminal de entrada. Quando o neurônio é ativado, é calculada a soma ponderada dos valores dos terminais de entrada pelos pesos, produzindo o *nível de ativação* corrente. Este é processado por uma função de ativação para produzir o valor de saída do neurônio.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

Class *ClassicalNeuron*

superclasses $Neuron(IMPULSE \setminus \mathbb{R}, ACTIVATION \setminus \mathbb{R})$

Os *given sets* da superclasse *Neuron* são instanciados para o tipo $\mathbb{R}$.

private *WeightedSum*

state

É incluído o componente *weightbuffer* para modelar o *buffer* de pesos do neurônio. Este deve ter o mesmo tamanho do *buffer* de entradas e existe uma correspondência implícita dos elementos que ocupam a mesma posição nos dois *buffers*. Por exemplo, a primeira posição do *buffer* de pesos corresponde ao peso do primeiro terminal de entrada.

weightbuffer : Weights
inbuffer length = weightbuffer length

$$Weights == Buffer(VALUE \setminus \mathbb{R})$$

initialstates

CreateNeuron passa a incluir a definição do valor inicial de *weightbuffer*.

CreateNeuron
$\Delta(weightbuffer)$
weightbuffer CreateBuffer(values? \ weightbuffervalues?)
inbuffervalues? len = weightbuffervalues? len

operations

ChangeWeight muda os valores de alguns pesos de *weightbuffer*. É utilizada a operação *ChangeValues* da classe *Buffer* que recebe um mapeamento de posições do *buffer* para os novos valores.

$$ChangeWeight \mathrel{\hat=} \Delta(weightbuffer) \wedge weightbuffer\ ChangeValues(values? \setminus wbvalues?)$$

IncInBuffer é redefinida para especificar que *weightbuffer* também deve ser incrementado.

IncInBuffer
$\Delta(weightbuffer)$
IncInBuffer
weightBuffer Insert(values?\weightbuffervalues?)
inbuffervalues? len = weightbuffervalues? len

DecInBuffer é redefinida para que *weightbuffer* seja decrementado das mesmas posições a serem retiradas de *inbuffer*.

ComputeActivation é redefinida para incluir a função para o cálculo de *activation*.

ComputeActivation
$\Xi(inbuffer, weightbuffer)$
ComputeActivation
a = WeightedSum(inbuffer buffer, weightbuffer buffer)

$$ComputeActivation \mathrel{\hat=} ComputeActivation \setminus (a)$$

WeightedSum calcula a soma ponderada de duas seqüências de números reais.

$$WeightedSum : \quad Seq\ \mathbb{R} \times Seq\ \mathbb{R} \rightarrow \mathbb{R}$$

$$\forall i : Seq\ \mathbb{R};\ w : Seq\ \mathbb{R} \bullet (i = \langle\rangle \wedge w = \langle\rangle \Rightarrow WeightedSum(i, w) = 0) \wedge$$
$$(i \neq \langle\rangle \wedge w \neq \langle\rangle \Rightarrow WeightedSum(i, w) =$$
$$i\ head * w\ head + WeightedSum(i\ tail, w\ tail))$$

EndClass *ClassicalNeuron*.

3.1.3 Classe *WeightLessNeuron*

Alguns modelos de neurônio são caracterizados por não armazenar informações em pesos, mas em células de memória. Nestes neurônios, cada célula de memória é endereçada por uma função aplicada a uma certa combinação dos valores dos terminais de entrada do neurônio. Cada célula pode armazenar a saída do neurônio, ou alguma outra informação que concorra para a definição deste valor. Neurônios deste tipo são comumente denominados *neurônios sem peso*. A classe *WeightLessNeuron* é definida para incluir esta característica.

Class *WeightLessNeuron*

givensets *CELL*

CELL parametriza a classe com relação ao tipo dos objetos armazenados nas células.

superclasses *Neuron*(*activation* \ *address*, *ACTIVATION* \ *ADDRESS*,
computeActivation \ *computeAddress*)

state

Um novo componente é definido: *cells*, um mapeamento de endereços para valores de células. O componente *address* passa a representar um endereço de célula gerado em função dos valores de entrada durante a ativação do neurônio.

$$cells : Cells$$
$$address \in cells\ dom$$

$$Cells == MAP\ ADDRESS\ CELL$$

initialstates

<u>*CreateNeuron*</u> dá o valor inicial do componente *cells*, usando as entradas externas: *addresses?*, o conjunto de endereços das células, e *cellvalue?*, o valor inicial das células. O esquema *CreateNeuron* da superclasse é implicitamente incluído no esquema abaixo.

CreateNeuron

$$\Delta(cells)$$
$$addresses? : \mathbb{P}\ ADDRESS$$
$$cellvalue? : CELL$$
$$address' \in addresses? \wedge cells' = \{a : addresses? \bullet a \mapsto cellvalue?\}$$

operations

<u>*SetCell*</u> muda o conteúdo de uma célula.

$$\boxed{\begin{array}{l} \textit{SetCell} \\ \hline \Delta(cells) \\ add? : ADDRESS \\ value? : CELL \\ \hline (add? \in cells\ dom) \wedge (cells'(add?) = value?) \wedge (cells' \ntriangleleft \{add?\} = cells \ntriangleleft \{add?\}) \end{array}}$$

<u>*InsertCells*</u> adiciona um conjunto de novas célula ao neurônio. Os novos endereços não devem pertencer ao conjunto de endereços de células do neurônio.

$$\boxed{\begin{array}{l} \textit{InsertCells} \\ \hline \Delta(cells) \\ adds? : \mathbb{P}\ ADDRESS \\ cellvalue? : CELL \\ \hline (\forall a : adds? \bullet a \notin cells\ dom) \wedge (cells' = cells \cup \{a : adds? \bullet a \rightarrow cellvalue?\}) \end{array}}$$

<u>*RemoveCells*</u> remove um conjunto de células cujos endereços são dados por *adds?*.

EndClass *WeightLessNeuron*.

3.2 Redes Neurais Artificiais

Nesta seção, é especificado um conjunto de classes para a modelagem de redes neurais artificiais. São utilizadas, como classes clientes, as especificações de classes de neurônio dadas na seção 3.1. A figura 4 mostra a hierarquia das classes especificadas e o relacionamento com as classes clientes. *NeuralNetwork*, *SlabbedNeuralNetwork*, *LayeredNeuralNetwork*, *FeedForward* e *Pyramidal* são as classes abstratas. A intenção é que estas classes sirvam como base para a definição de outras classes abstratas e, conseqüentemente, de classes que modelem paradigmas de redes neurais como os especificados nas classes *MLPBackPropagation* [10], *Hopfield* [11] e *FeedForwardGSN* [2]. A classe *Topology* define uma topologia de nós e arcos e é utilizada como superclasse da classe mais abstrata de rede neural, *NeuralNetwork*. A classe *SlabbedNeuralNetwork* representa os modelos nos quais os neurônios são agrupados em *slabs*. Na classe *LayeredNeuralNetwork*. os *slabs* são ordenados e passam a ser chamados *layers*. A classe *FeedForward* representa os modelos onde em todas as conexões, o neurônio de origem pertence a uma camada de ordem menor que a do neurônio de destino. Já a classe *Pyramidal* se caracteriza por seus neurônios serem organizados em pirâmides. estruturas de multi-camadas onde cada neurônio só pode ser origem de uma conexão da rede.

3.2.1 Classe *Topology*

Nesta classe. é definida uma topologia genérica tal que possa ser usada em outras especificações além da especificação de rede neural. Em uma topologia, são identificados dois objetos básicos: nós e arcos (conexões entre nós). Quatro operações podem ser realizadas sobre uma topologia: incluir nós. incluir arcos, remover nós e remover arcos.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

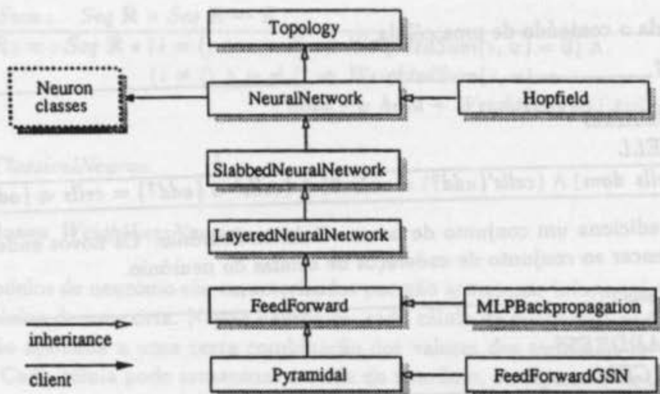

Figura 4: Hierarquia das classes de redes neurais.

Class *Topology*

givensets *NOID*

O tipo dos nós é dado por um *given set*.

state

Uma topologia pode ser dada por um conjunto de nós e um conjunto de arcos que conectam os nós entre si. Arcos só conectam nós conhecidos pela topologia.

$nodes : \mathbb{P}\ NOID$
$arcs : \mathbb{P}\ Arc$
$\forall a : arcs \bullet a.source \in nodes \ \wedge \ a.target \in nodes$

Arc
$source : NOID$
$target : NOID$

initialstates

Uma topologia, no estado inicial, não possui nós, e portanto, não possui arcos.

Init
$\Delta\,(nodes,\ arcs)$
$nodes' = \varnothing$

operations

InsertNodes **define a inserção de nós. Os novos nós não devem pertencer à topologia.**

InsertNodes
$\Delta\,(nodes)$
$nodes? : \mathbb{P}\ NOID$
$(\forall n : nodes? \bullet n \notin nodes) \wedge (nodes' = nodes \cup nodes?)$

OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

InsertArcs define a inserção de arcos. Os nós de origem e de destino devem ser conhecidos.

InsertArcs
$\Delta(arcs)$
$\Xi(nodes)$
$arcs? : \mathbb{P}\ Arc$
$(\forall a : arcs? \bullet a \notin arcs \land a.source \in nodes \land a.target \in nodes) \land (arcs' = arcs \cup arcs?)$

RemoveArcs define a remoção de arcos da topologia. *RemoveNodes* define a remoção de nós da topologia. Os arcos que referenciam os nós a serem removidos também precisam ser removidos nesta operação.

EndClass *Topology*.

3.2.2 Classe *NeuralNetwork*

Uma rede neural artificial [14. 11, 5], em seu aspecto estático, pode ser vista como uma topologia de nós conectados por arcos direcionados, onde os nós são neurônios artificiais. Cada arco funciona como um canal unidirecional de comunicação. Sinais externos servem como entrada para a rede. Sinais também são gerados para o mundo externo. A figura 5 ilustra a arquitetura de uma rede neural, isto é, a descrição da sua morfologia. Cada neurônio possui múltiplas conexões de entrada, que podem ter como origem a saída de outros neurônios, ou entradas do mundo externo.

Os terminais de entrada externa da rede são modelados por um conjunto de neurônios especiais, cuja função é distribuir estes valores para os outros neurônios da rede. Este tipo de neurônio é comumente adotado na descrição de arquiteturas de redes neurais em [14, 11]. Optou-se por utilizar estes neurônios ao invés de uma seqüência de elementos, por exemplo, a fim de manter uniforme o padrão de conexão. Uma conexão entre dois neurônios da rede é estabelecida da mesma forma que entre um neurônio da rede e um objeto do mundo externo.

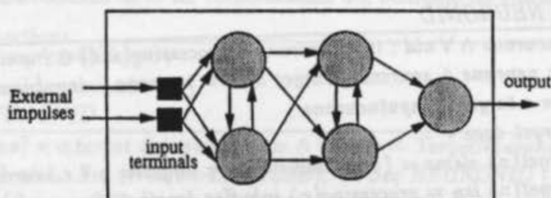

Figura 5: Arquitetura de uma rede neural.

A classe *NeuralNetwork* é especificada como subclasse da classe *Topology* e como cliente das classes de neurônio. Nesta classe, os nós correspondem a identificadores de neurônio. Estes identificadores são mapeados em objetos da classe *Neuron*. Por ser a superclasse mais abstrata, a classe *Neuron* é utilizada para representar os neurônios.

Class *NeuralNetwork*

givensets *NEURONID*
NEURONID é utilizado para representar o conjunto de identificadores de neurônio.

superclasses *Topology(NOID \ NEURONID, Arc \ Connection, nodes \ neurons, arcs \ connections)*
private *InsertNodes, InsertArcs, RemoveNodes, RemoveArcs*

private *TargetNeurons, SourceNeurons, SourcesOfTarget, TargetsOfSource*

state

Ao estado de *NeuralNetwork* são acrescentados os componentes *processing*, usado para associar identificadores de neurônio a objetos da classe *Neuron*, *inputneurons*, para definir o conjunto de neurônios de entrada, e *sourcesOfTarget*, que é um mapeamento de identificadores de neurônio para seqüências de identificadores dos neurônios que são a origem de suas conexões. Os identificadores destas seqüências devem pertencer ao conjunto de identificadores de neurônio da rede: *neurons*. Para todo neurônio i associado a seqüência de um neurônio j em *sourcesOfTarget*, deve existir uma conexão no componente *connections* de i para j. A seqüência de neurônios s, associada a cada neurônio i no componente *sourcesOfTarget*, possui uma correspondência implícita com os seus terminais de entrada: o primeiro neurônio de s está associado ao primeiro terminal de entrada do neurônio i. Assim, o número de entradas do neurônio i deve ser igual ao tamanho da seqüência s. A única exigência no invariante para o componente *processing* é que os identificadores de neurônio que o acessam sejam os próprios identificadores de neurônio da rede: *neurons*. Cada objeto da classe *Neuron* está associado em *processing* a um identificador de neurônio da rede. O conjunto de identificadores de neurônios de entrada deve estar contido no conjunto de neurônios da rede. O domínio de *sourcesOfTarget* não deve incluir os neurônios de entrada, pois estes neurônios não recebem conexões dos outros neurônios da rede. Todos os neurônios do componente *processing* cujos identificadores pertencem a *inputneurons* são objetos da classe *InputNeuron*.

$processing : MAP\ NEURONID\ Neuron$
$sourcesOfTarget : MAP\ NEURONID\ NeuronIds$
$inputneurons : \mathbb{P}\ NEURONID$

$inputneurons \subseteq neurons \wedge \forall\, nid : inputneurons \bullet processing(nid) \in InputNeuron$
$processing\ dom = neurons \wedge sourcesOfTarget\ dom = neurons - inputneurons$
$\forall\, c : connections \bullet c.target \notin inputneurons$
$\forall\, n : sourcesOfTarget\ dom \bullet$
$\quad sourcesOfTarget(n)\ elems = \{c : connections \mid c.target = n \bullet c.source\} \wedge$
$\quad sourcesOfTarget(n)\ len = processing(n)\ inbuffer\ length$

$$NeuronIds = Seq\ NEURONID$$

initialstates

O estado inicial dos componentes *neurons* e *connections* é dado pela operação de inicialização da superclasse. Os componentes *processing* e *sourcesOfTarget* são automaticamente inicializados, uma vez que são definidos no invariante em termos do componente *neurons*.

$$Create \mathrel{\widehat=} \Delta(processing, sourcesOfTarget, inputneurons) \wedge Init$$

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

operations

Inicialmente. são dadas algumas definições auxiliares. *TargetNeurons* devolve o conjunto de identificadores de neurônio que são destino em um conjunto de conexões. *SourceNeurons* devolve o conjunto de identificadores de neurônio que são origem em um conjunto de conexões. *SourcesOfTarget* devolve um conjunto com os identificadores que são origem de conexões de um conjunto c tal que o destino seja o identificador t. *TargetsOfSource* devolve um conjunto com os identificadores que são destino de conexões de um conjunto c tal que a origem seja o identificador t.

InsertNeurons acrescenta novos neurônios à rede. Requer um conjunto de identificadores de neurônio. *neuronIds?*. Estes são inseridos na rede através da operação *InsertNodes* da superclasse, renomeando a entrada *nodes?* para *neuronIds?*. Para cada identificador de *neuronIds?* é criado um neurônio e a associação entre eles é incluída em *processing*. Os novos neurônios são incluídos em *sourcesOfTarget* mapeando uma seqüência vazia, pois ainda não possuem conexões. Uma operação especial (*InsertInputNeurons*) é definida para a inserção de neurônios de entrada.

InsertNeurons

$\Delta\ (processing, sourcesOfTarget)$
$InsertNodes(nodes? \setminus neuronIds?)$

$processing'\ dom = processing\ dom \cup neuronIds?$
$\forall i : neuronIds? \bullet \exists n, n' : Neuron \mid n\ InitNeuron \wedge n = n' \bullet (i \mapsto n) \in processing'$
$processing' \ndres neuronIds? = processing \ndres neuronIds?$
$sourcesOfTarget' = sourcesOfTarget \cup \{nid : neuronIds? \bullet nid \mapsto \langle\rangle\}$

InsertConnections define a inserção de novas conexões na rede. O esquema *InsertArcs* da superclasse é utilizado, renomeando a entrada *arcs?* para *connections?*. Entradas nos neurônios destino devem ser criadas para receber as conexões. Os outros neurônios não devem ser alterados. Novas entradas nas seqüências da imagem do componente *sourcesOfTarget* associadas a neurônios que são destino nas novas conexões também devem ser criadas. Nenhuma conexão deve ter como destino um neurônio de entrada.

InsertConnections

$\Delta\ (processing, sourcesOfTarget)$
$InsertArcs\ (arcs? \setminus connections?)$
$targets : \mathbb{P}\ NEURONID$

$\forall c : connections? \bullet c.target \notin inputneurons \wedge targets = TargetNeurons(connections?)$
$\forall t : targets \bullet \exists\ vls? : Seq\ (Neuron\ IMPULSE);\ s : Seq\ NEURONID \mid$
$\quad s\ len = \#(SourcesOfTarget(t, connections?)) \wedge$
$\quad s\ elems = SourcesOfTarget(t, connections?) \wedge vls?\ len = s\ len \bullet$
$\qquad processing(t)\ IncInBuffer(inbuffervalues? \setminus vls?) \wedge$
$\qquad sourcesOfTarget'(t) = sourcesOfTarget(t) \frown s$
$processing' \ndres targets = processing \ndres targets$
$sourcesOfTarget' \ndres targets = sourceOfTargets \ndres targets$

$$InsertConnections \mathrel{\widehat{=}} InsertConnections \setminus (targets)$$

São definidas também operações para remover neurônios e conexões da rede. Até este ponto. foram abordadas operações associadas ao processo de criação de uma rede neural

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

(definição de sua arquitetura). A seguir, são apresentadas operações que servirão de base para modelar o aspecto dinâmico de uma rede.

SetNeuronInputs recebe novos valores para terminais de entrada de um neurônio.

Os neurônios de uma rede podem ser ativados em modo síncrono ou assícrono. No modo síncrono, um grupo de neurônios é ativado em paralelo ao mesmo tempo, tal que se a entrada de algum neurônio estiver conectada a saída de um outro, o novo valor de saída não será propagado e será utilizado o valor de entrada no tempo anterior. No modo assíncrono, os neurônios são ativados seqüencialmente tal que se um neurônio receber conexão de algum neurônio que já tiver sido ativado, o valor da entrada correspondente deve ser atualizado antes da sua ativação.

ActivateNeurons define a ativação de um conjunto de neurônios em modo síncrono.

ActivateNeurons

$\Xi(neurons)$
$\Delta\ (processing)$
$neuronIds? : \mathbb{P}\ NEURONID$

$neuronIds? \subseteq neurons \wedge \forall n : neuronIds? \bullet processing(n)\ ActivateNeuron$
$processing' \lhd neuronIds? = processing \lhd neuronIds?$

GetNeuronsInputs atualiza o *buffer* de entradas de um conjunto de neurônios usando os valores de saída dos neurônios dos quais recebem conexões.

AsyActivateNeuron define a ativação de um neurônio em modo assíncrono. Os valores dos terminais de entrada do neurônio são atualizados antes da ativação.

$$AsyActivateNeuron \mathrel{\widehat=} [nId? : NEURONID;\ nIds : \mathbb{P}\ NEURONID \mid nIds = \{nId?\}] \wedge$$
$$(GetNeuronsInputs(neuronIds? \setminus nIds) \mathbin{;} ActivateNeurons(neuronIds? \setminus nIds))$$

EndClass *NeuralNetwork*.

4 Conclusões

O principal objetivo deste trabalho é a utilização de especificações formais orientadas a objetos para produzir uma descrição abstrata, precisa, livre de ambigüidades e inconsistências das propriedades de redes neurais artificiais. A especificação é dividida em duas partes, onde duas hierarquias de classes são fornecidas e os conceitos são gradualmente introduzidos. Usando o mecanismo de herança, novos conceitos podem ser incorporados, a partir da criação de subclasses. As classes destas hierarquias podem ser utilizadas para a especificação de diferentes paradigmas de redes neurais ou para a especificação de aplicações. Podem servir também como base para a prova de propriedades sobre redes neurais artificiais, como estabilidade e convergência, reduzindo a necessidade de realizar simulação para verificar tais propriedades.

Visto que novos paradigmas de redes neurais estão constantemente sendo desenvolvidos é importante que um ambiente de simulação facilite a sua incorporação. A adoção de especificações formais e orientação a objetos no desenvolvimento de software para simulação de redes neurais artificiais torna possível a construção de sistemas confiáveis e de

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

fácil evolução. A orientação a objetos promove modularidade e reusabilidade. facilitando a extensibilidade do sistema.

Referências

[1] I. Aleksander and H. B. Norton. An overview of weightless neural nets. In Maureen Caudill and Charles Butler. editors. *Int. Joint Conf. on Neural Networks*. volume II. pages 499–502. San Diego, June 1990. IEEE.

[2] E. C. D. B. C. Filho. D. L. Bisset. and M. C. Fairhurst. Architectures for goal-seeking neurons. *International Journal of Intelligent Systems*, 1991.

[3] N. H. Goddard. K. J. Lynne. and T. Mintz. Rochester connectionist simulator. Technical Report 233. University of Rochester, Computer Science Department, Rochester, New York. 14627. march 1988.

[4] H. M. Gomes and P. D. L. Machado. Especificando Redes Neurais Artificiais em MooZ. Technical report, Universidade Federal de Pernambuco, Departamento de Informática, Recife - PE, 1993.

[5] R. Hecht-Nielsen. *Neurocomputing*. Addison-Wesley Publishing Company, Inc. 1990.

[6] C. A. R. Hoare. Programming: Sorcery or science ? *IEEE Software*, pages 5–16, april 1984.

[7] P. D. L. Machado. E. C. D. B. Carvalho Filho, S. R. L. Meira. and H. M. Gomes. EASY - an [E]nvironment for [A]rtificial neural [SY]stems simulation. In *Proceedings of Fourth Irish Neural Network Conference - INNC'94*, Dublin, Ireland, september 1994.

[8] S. R. L. Meira and A. L. C. Cavalcanti. Modular Object-Oriented Z Specifications. In Prof. C. J. van Rijsbergen, editor, *Workshop on Computing Series*. pages 173 - 192, Oxford - UK, December 1990. Springer-Verlag.

[9] S. R. L. Meira and A. L. C. Cavalcanti. The MooZ Specification Language. Technical report. Universidade Federal de Pernambuco. Departamento de Informática, Recife - PE, 1992. Available via anonymous ftp from rosa.cr-pe.rnp.br, file pub/MooZ/MooZ.ps.Z.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing*, chapter 8, pages 318–362. MIT Press. Cambridge, 1986.

[11] P. K. Simpson. *Artificial Neural Systems: Foundations. Paradigms, Aplications and Implementations.* Pergamon Press, Inc. 1990.

[12] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall. 1989.

[13] C. F. Stevens. The neuron. In *The biology of the brain from neurons to neural networks*. chapter 1. Freeman. September 1979.

[14] P. D. Wasserman. *Neural Computing: Theory and Practice.* ANZA Research. Inc. New York, 1989.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor