

unesp

**UNIVERSIDADE ESTADUAL PAULISTA
FACULDADE DE ENGENHARIA DE ILHA SOLTEIRA
PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA**

***DISSERTAÇÃO DE MESTRADO
EM
ENGENHARIA ELÉTRICA***

1210001005



Nº 6

**SISTEMAS DE CONTROLE AUTOMÁTICO BASEADOS
NO CEREBELLAR MODEL ARTICULATION CONTROLLER
ALUNO: MAURÍCIO JOSÉ BORDON
ORIENTADOR: MARCELO CARVALHO MINHOTO TEIXEIRA**

TE.1005

UNIVERSIDADE ESTADUAL PAULISTA JÚLIO DE MESQUITA
FILHO
- FACULDADE DE ENGENHARIA DE ILHA SOLTEIRA -
FEIS/UNESP

CURSO DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

AUTOMAÇÃO E CONTROLE

**SISTEMAS DE CONTROLE AUTOMÁTICO BASEADOS NO
CEREBELLAR MODEL ARTICULATION CONTROLLER**

"Dissertação apresentada à
Faculdade de Engenharia de Ilha
Solteira - UNESP, como parte dos
requisitos para obtenção do título
de Mestre em Engenharia Elétrica"

Candidato: Maurício José Bordon

Orientador: Prof. Dr. Marcelo C. Minhoto Teixeira

Ilha Solteira - SP
06/10/1995

BCpIS
FEIS-UNESP



"SISTEMAS DE CONTROLE AUTOMÁTICO BASEADOS NO CEREBELLAR
MODEL ARTICULATION CONTROLLER"

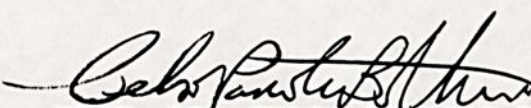
MAURÍCIO JOSÉ BORDON

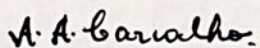
Dissertação submetida à Faculdade de Engenharia de Ilha
Solteira-UNESP como parte dos requisitos necessários para a obtenção
do título de Mestre em Engenharia Elétrica (ME).


PROF. DR. CARLOS ROBERTO MINUSSI - COODENADOR

BANCA EXAMINADORA


PROF. DR MARCELO CARVALHO MINHOTO TEIXEIRA - ORIENTADOR


PROF. DR. CELSO PASCOLI BOTTURA - FEE- UNICAMP


PROF. DR APARECIDO AUGUSTO DE CARVALHO - FEIS - UNESP

Ilha solteira - SP, Outubro de 1995

BCpIS
FEIS-UNESP

AGRADECIMENTOS

Agradecemos aos professores **Aparecido Augusto de Carvalho** e **Carlos Roberto Minussi** pelas importantes correções sugeridas, ao professor **Marcelo Carvalho Minhoto Teixeira** pelo excelente trabalho de orientação e, também, à **FAPESP** pela Bolsa de Mestrado concedida e à **FUNDUNESP** pelo auxílio financeiro a este projeto de pesquisa.

1210001005



Doc. 039/96-URD 53/96

UNESP- "CAMPUS" DE ILHA SOLTEIRA SERVIÇO DE BIBLIOTECARIA E DOCUMENTAÇÃO	
DATA DE RECEBIMENTO <i>01.03.96</i>	DATA DE TOMBAMENTO <i>29.03.96</i>
REGISTRO <i>Ailza</i>	TOMBO <i>Te. 1005</i>
AQUIZIÇÃO <i>Ilustração R\$10.00 Autor</i>	CLASSIFICAÇÃO

BCpIS
FEIS-UNESP

RESUMO

Este trabalho aborda o controle automático de sistemas dinâmicos utilizando o "**Cerebellar Model Articulation Controller**" (**CMAC**). O **CMAC** é uma Rede Neuronal Artificial (**RNA**) desenvolvida por **Albus** e baseada nos cerebelos de alguns mamíferos.

Nesta pesquisa foi desenvolvida uma nova configuração, baseada no **CMAC**, utilizando pesos com valores discretos, com o objetivo de reduzir a memória física necessária. São apresentadas simulações que comprovam a eficiência da configuração proposta.

É descrita, também, uma aplicação dos resultados obtidos no controle automático da posição de um pêndulo forçado, utilizando uma saída tipo relé.

ABSTRACT

This work treats the control of dynamic systems using the "**Cerebellar Model Articulation Controller**" (**CMAC**). **CMAC** is a kind of neural network developed by **Albus**, based on the cerebellar cortex of some mammals.

We present a new configuration based on **CMAC**, using discrete weights, carrying out a smaller memory requirement. Simulations show that the new configuration provides a better performance than using **HASH** code.

This work presents an application of these new results on the control of a forced pendulum using a relay type output.



ÍNDICE

Índice.....	i
Lista de ilustrações.....	iii
Lista de tabelas.....	v
Lista de Abreviações.....	x
Lista de Símbolos.....	xi
 Capítulo 1 - INTRODUÇÃO.....	 1
 Capítulo 2 - REDES NEURONAIS ARTIFICIAIS	
2.1 INTRODUÇÃO.....	4
2.2 ELEMENTO LINEAR ADAPTATIVO.....	5
2.3 ALGOITMOS UTILIZADOS NO TREINAMENTO DO ADALINE	
2.3.1 Regra Delta de Widrow-Hoff.....	9
2.3.2 Algoritmo Backward Error Propagation.....	10
2.4 ESTRUTURAS DE REDES NEURONAIS ARTIFICIAIS.....	13
2.5 TREINAMENTO DE REDES NEURONAIS ARTIFICIAIS	
2.5.1 Madaline Rule I.....	15
2.5.2 Madaline Rule II.....	18
2.5.3 Backward Error Propagation.....	20
2.5.4 Treinamento da Rede de Hopfield.....	25
 Capítulo 3 - CONTROLE DE SISTEMAS COM REDES NEURONAIS ARTIFICIAIS	
3.1 INTRODUÇÃO.....	28
3.2 IDENTIFICAÇÃO DE SISTEMAS.....	29
3.2.1 Modelagem Direta.....	29
3.2.2 Modelagem Inversa.....	31
3.3 ESTRUTURAS DE CONTROLE COM REDES NEURONAIS ARTIFICIAIS	
3.3.1 Controle Supervisionado.....	35
3.3.2 Controle Inverso Direto.....	35
3.3.3 Controle por Modelo de Referência.....	36



3.3.4 Controle por Modelo Interno.....	37
3.3.5 Self Tuning Regulator.....	38
3.3.6 Linearização por Realimentação.....	39
 Capítulo 4 - O CEREBELLAR MODEL ARTICULATION CONTROLLER	
4.1 INTRODUÇÃO.....	41
4.2 AS TRANSFORMAÇÕES PRESENTES NO CMAC	
4.2.1 Transformação $s \rightarrow m$	43
4.2.2 Transformação $m \rightarrow a$	45
4.2.3 Transformação $a \rightarrow \hat{x}$	47
4.3 TREINAMENTO DO CMAC.....	48
4.4 REQUISITOS DE MEMÓRIA.....	57
4.5 CMAC ALTERNATIVO.....	58
 Capítulo 5 - SISTEMAS DE CONTROLE AUTOMÁTICO COM O CMAC	
5.1 INTRODUÇÃO.....	62
5.2 O CMAC COMO CONTROLADOR POR RETROAÇÃO.....	63
5.3 O CMAC TREINADO POR REGRAS.....	64
 Capítulo 6 - O CMAC COM PESOS DISCRETOS.....	68
 Capítulo 7 - APLICAÇÃO DO CMAC COM PESOS DISCRETOS	
7.1 INTRODUÇÃO.....	75
7.2 REPRODUÇÃO DE FUNÇÕES ATRAVÉS DO CMAC DE KRAFT & CAMPAGNA.....	76
7.3 REPRODUÇÃO DE FUNÇÕES ATRAVÉS DO CMAC DE ALBUS.....	93
7.4 EXEMPLO DE CONTROLE POR MODELO INVERSO.....	109
 Capítulo 8 - CONCLUSÕES.....	116
 Capítulo 9 - REFERÊNCIAS BIBLIOGRÁFICAS.....	119



FIGURA 2.1: Elemento Linear Adaptativo.....	5
FIGURA 2.2: Funções de saída típicas em ADALINE.....	6
FIGURA 2.3: ADALINE com duas entradas e função de saída $f(s)=\text{sgn}(s)$	7
FIGURA 2.4: Exemplos de RNA.....	14
FIGURA 2.5: Rede MADALINE I e respectivas funções de saída.....	17
FIGURA 2.6: Arquitetura MADALINE II com duas camadas e cinco neurônios...	19
FIGURA 2.7: RNA MADALINE feedforward com duas camadas e cinco neurônios..	21
FIGURA 2.8: Notação utilizada para o desenvolvimento do algoritmo Backpropagation.....	23
FIGURA 3.1: Estrutura para modelagem direta de sistemas através de RNA...	31
FIGURA 3.2: Estrutura para o treinamento inverso generalizado.....	33
FIGURA 3.3: Estrutura para o treinamento inverso generalizado.....	34
FIGURA 3.4: Arquitetura para o controle por modelo de referência.....	36
FIGURA 3.5: Arquitetura para o controle por modelo interno.....	37
FIGURA 3.6: Estrutura de controle Self Tuning Regulator.....	39
FIGURA 4.1: Modelo simplificado do cerebelo.....	42
FIGURA 4.2: Neste exemplo, as funções de quantização do CMAC fonecem $N=2$ vetores com $C=3$ elementos discretos.....	45
FIGURA 4.3: A capacidade de generalização do CMAC vem da sobreposição que ocorre para dois vetores de entrada próximos.....	47
FIGURA 4.4: Conjunto S e a Região Ativa para $s=(x_0, y_0)$	59
FIGURA 5.1: Controle de um sistema dinâmico de primeira ordem com CMAC...	63
FIGURA 5.2: Sistema físico a ser controlado pelo CMAC ao final do treinamento.....	65
FIGURA 5.3: Os controladores geravam seis parâmetros como função do estado da planta.....	66
FIGURA 6.1: Tipo do conversor A/D empregado na discretização de dados amostrados.....	69
FIGURA 7.1: Função $\text{fsen}()$ amostrada a partir de um conjunto de 50×50 pontos	



igualmente espaçados.....	77
FIGURA 7.2: Função $f_{cos}()$ amostrada a partir de um conjunto de 50×50 pontos igualmente espaçados.....	77
FIGURA 7.3: Função $canto()$ amostrada a partir de um conjunto de 50×50 pontos igualmente espaçados.....	77
FIGURA 7.4: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft & Campagna.....	79
FIGURA 7.5: Aproximação resultante do treinamento representado na figura 7.4.....	79

Referências mais detalhadas sobre as figuras 7.4 a 7.51 são encontradas na lista de tabelas.

FIGURA 7.50: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft & Campagna.....	90
FIGURA 7.51: Aproximação resultante do treinamento representado na figura 7.50.....	90
FIGURA 7.52: Gráfico do erro quadrático médio durante o treinamento do CMAC de Albus.....	95
FIGURA 7.53: Aproximação resultante do treinamento representado na figura 7.52.....	95

Referências mais detalhadas sobre as figuras 7.52 a 7.99 são encontradas na lista de tabelas.

FIGURA 7.98: Gráfico do erro quadrático médio durante o treinamento do CMAC de Albus.....	106
FIGURA 7.99: Aproximação resultante do treinamento representado na figura 7.98.....	106
FIGURA 7.100(a): Diagrama de blocos usado para o treinamento do CMAC a partir de entradas e saídas amostradas diretamente da planta.....	112



FIGURA 7.100(b): Diagrama de blocos para o controle do pêndulo forçado a partir das excitações fornecidas pelo CMAC já treinado. O objetivo é que a planta siga uma sequência de valores de posição e de velocidade previamente especificados.....	113
FIGURA 7.101: Comportamento da posição do pêndulo forçado durante a ação do controlador baseado no CMAC de Albus com pesos discretos...	115
FIGURA 7.102: Comportamento do torque aplicado ao pêndulo forçado pelo CMAC, durante a simulação.....	115
FIGURA 7.103: Diagrama esquemático do pêndulo forçado simulado.....	115



LISTA DE TABELAS

Tabela 2.1: Função reproduzida pelo ADALINE da figura 2.3.....8

Tabela 4.1: Várias possibilidades para a solução da equação matricial (4.2) e os resultados do treinamento pelo algoritmo proposto por Albus.....52

Tabela 7.1: Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $f_{sen}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 1350 bytes; a versão com código HASH necessita de 1360 bytes para armazenar 340 pesos tipo ponto flutuante de 4 bytes cada um.....79

Tabela 7.2: Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $f_{sen}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 1350 bytes; a versão com código HASH necessita de 1360 bytes para armazenar 340 pesos tipo ponto flutuante de 4 bytes cada um.....80

Tabela 7.3. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $f_{sen}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 788 bytes; a versão com código HASH necessita de 800 bytes para armazenar 200 pesos tipo ponto flutuante de 4 bytes cada um.....81

Tabela 7.4. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $f_{sen}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 788 bytes; a versão com código HASH necessita de 800 bytes para



armazenar 200 pesos tipo ponto flutuante de 4 bytes cada um.....82

Tabela 7.5. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $\text{fcos}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 1350 bytes; a versão com código HASH necessita de 1360 bytes para armazenar 340 pesos tipo ponto flutuante de 4 bytes cada um.....83

Tabela 7.6. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $\text{fcos}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 1350 bytes; a versão com código HASH necessita de 1360 bytes para armazenar 340 pesos tipo ponto flutuante de 4 bytes cada um.....84

Tabela 7.7. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $\text{fcos}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 788 bytes; a versão com código HASH necessita de 800 bytes para armazenar 200 pesos tipo ponto flutuante de 4 bytes cada um.....85

Tabela 7.8. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $\text{fcos}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 788 bytes; a versão com código HASH necessita de 800 bytes para armazenar 200 pesos tipo ponto flutuante de 4 bytes cada um.....86

Tabela 7.9. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $\text{canto}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 1350 bytes; a versão com código HASH necessita de 1360 bytes para armazenar 340 pesos tipo ponto flutuante de 4 bytes cada um.....87



Tabela 7.10. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função `canto()` a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 1350 bytes; a versão com código HASH necessita de 1360 bytes para armazenar 340 pesos tipo ponto flutuante de 4 bytes cada um.....88

Tabela 7.11. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função `canto()` a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 788 bytes; a versão com código HASH necessita de 800 bytes para armazenar 200 pesos tipo ponto flutuante de 4 bytes cada um.....89

Tabela 7.12. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função `canto()` a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 788 bytes; a versão com código HASH necessita de 800 bytes para armazenar 200 pesos tipo ponto flutuante de 4 bytes cada um.....90

Tabela 7.13: Resultados totais obtidos a partir das simulações representadas nas figuras 7.4 a 7.51.....92

Tabela 7.14. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função `fsen()` a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 450 bytes; a versão com código HASH necessita de 480 bytes para armazenar 120 pesos tipo ponto flutuante de 4 bytes cada um.....95

Tabela 7.15. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função `fsen()` a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 450 bytes; a versão com código HASH necessita de 480 bytes para armazenar 120 pesos tipo ponto flutuante de 4 bytes cada um.....96



Tabela 7.16. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função `fsen()` a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 264 bytes; a versão com código HASH necessita de 280 bytes para armazenar 70 pesos tipo ponto flutuante de 4 bytes cada um.....97

Tabela 7.17. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função `fsen()` a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 264 bytes; a versão com código HASH necessita de 280 bytes para armazenar 70 pesos tipo ponto flutuante de 4 bytes cada um.....98

Tabela 7.18. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função `fcos()` a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 450 bytes; a versão com código HASH necessita de 480 bytes para armazenar 120 pesos tipo ponto flutuante de 4 bytes cada um.....99

Tabela 7.19. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função `fcos()` a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 450 bytes; a versão com código HASH necessita de 480 bytes para armazenar 120 pesos tipo ponto flutuante de 4 bytes cada um.....100

Tabela 7.20. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função `fcos()` a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 264 bytes; a versão com código HASH necessita de 280 bytes para armazenar 70 pesos tipo ponto flutuante de 4 bytes cada um.....101

Tabela 7.21. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função `fcos()` a partir de um conjunto de 200 pontos de



treinamento. Os pesos discretos são armazenados com 7 bits requerendo 264 bytes; a versão com código HASH necessita de 280 bytes para armazenar 70 pesos tipo ponto flutuante de 4 bytes cada um.....102

Tabela 7.22. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função `fcanto()` a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 450 bytes; a versão com código HASH necessita de 480 bytes para armazenar 120 pesos tipo ponto flutuante de 4 bytes cada um.....103

Tabela 7.23. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função `fcanto()` a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 450 bytes; a versão com código HASH necessita de 480 bytes para armazenar 120 pesos tipo ponto flutuante de 4 bytes cada um.....104

Tabela 7.24. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função `fcanto()` a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 264 bytes; a versão com código HASH necessita de 280 bytes para armazenar 70 pesos tipo ponto flutuante de 4 bytes cada um.....105

Tabela 7.25. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função `fcanto()` a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 264 bytes; a versão com código HASH necessita de 280 bytes para armazenar 70 pesos tipo ponto flutuante de 4 bytes cada um.....106

Tabela 7.26. Resultados totais obtidos a partir das simulações representadas nas figuras 7.52 a 7.99.....108



Lista de Abreviações

CMAC-Cerebellar Model Articulation Controller

RNA-Rede(s) Neuronal(is) Artificial(is)

ADALINE-Adaptive Linear Element

MADALINE-Multi Adaptive Linear Element

MRI-MADALINE Rule I

MR II-MADALINE Rule II

Backpropagation-Backward Error Propagation

RNA_c-RNA atuando como controlador de um sistema

RNA_m-RNA destinada a assimilar o modelo de um sistema

STR-Self Tuning Regulator

AL-algoritmo de treinamento proposto por **Albus**

AV-algoritmo de treinamento por média móvel

OS-algoritmo de treinamento que usa otimização do comprimento do passo na última direção ortogonal

ME-algoritmo de treinamento que atualiza primeiramente a informação com maior erro de representação

GS-algoritmo de treinamento que usa ortogonalização parcial pelo processo de **Grahm-Schmidt** normalizado.



Lista de Símbolos

- $\mathbf{x}$ -vetor de entrada de RNA ou planta
- $\mathbf{s}$ -saída linear de *ADALINE*, entrada do *CMAC*
- $\mathbf{y}$ -vetor de saída de RNA ou sistema
- $\mathbf{W}$ -vetor de pesos de RNA
- $\text{sgn}()$ -função sinal
- $\text{tgh}()$ -função tangente hiperbólica
- N -dimensão do espaço de entrada
- NP, NPT -número de elementos do conjunto de pontos de treinamento
- α -*LMS*-algoritmo α -Least Mean Square
- μ -*LMS*-algoritmo μ -Least Mean Square
- α, μ, β -constante de treinamento, estão usualmente contidas no intervalo $(0,1]$
- ϵ, e -erro linear entre a saída de uma RNA ou planta e o respectivo valor desejado
- d, y^d -valor desejado para a saída de uma RNA ou planta
- E, EQM -erro quadrático médio total entre as saídas de uma RNA e os respectivos valores desejados para vários pontos apresentados
- ξ -precisão requerida entre a saída de uma RNA e o respectivo valor desejado
- $H(\mathbf{x})$ -função a ser reproduzida por uma RNA ao final do treinamento
- $NN(\mathbf{x})$ -função assimilada pela RNA ao final do treinamento
- S -subconjunto compacto do $\mathbb{R}^N$
- δ -erro derivativo associado a um *ADALINE*
- L -número de camadas de uma RNA *MADALINE feedforward*
- N_L -número de *ADALINE* da última camada de uma RNA *MADALINE feedforward*
- u -excitação aplicada à entrada de uma planta
- $\mathbf{y}^p$ -vetor de saída de uma planta
- $\mathbf{y}^r$ -vetor de saída de uma RNA que deve assimilar o modelo de uma sistema
- $\mathbf{r}, \mathbf{y}^r$ -sinal de referência a ser aplicado na entrada de um sistema conhecido
- $\mathbf{y}^r$ -saída de um sistema conhecido em resposta a uma entrada de referência $\mathbf{r}$



e -erro entre a saída de uma RNA que deve assimilar o modelo de uma planta e o respectivo valor desejado

$f()$ -função de saída de um *ADALINE*; função que representa a dinâmica de um sistema não linear

p -dimensão do espaço vetorial imagem de uma transformação a ser reproduzida pelo *CMAC*

$q()$ -função de quantização dos elementos de entrada do *CMAC*

C -quantidade de funções de quantização aplicadas a cada variável de entrada do *CMAC*; constante de generalização do *CMAC*

D -número de intervalos possíveis na saída da função de quantização $q()$ quando esta é aplicada a uma das entradas do *CMAC*

m -vetor discreto que representa a saída de C funções de quantização do *CMAC*, quando são aplicadas a uma de suas entradas; metade do antecessor do número $(2m+1)$ que define a quantidade de valores discretos que os pesos discretos do *CMAC* podem assumir

a -vetor de células de associação

$\hat{a}$ -versor do vetor a

a' -conjunto de elementos unitários do vetor a

Z -quantidade de elementos do vetor de células de associação

A -matriz $NP \times Z$ em que cada linha é um vetor $\hat{a}$ associado a um ponto de treinamento

R -vetor com NP linhas: cada um de seus elementos é a saída desejada para um ponto de treinamento multiplicada por $\sqrt{C}$

Q -posto da matriz A

Q' -posto da matriz aumentada $[A, R]$

$\hat{w}$ -peso discreto do *CMAC*

v -número inteiro que define um valor discreto para um peso $\hat{w}$ do *CMAC* com pesos discretos

b -número de bits usado para representação dos números inteiros que definem um peso discreto do *CMAC*



n -metade do antecessor do número $(2n+1)$ que define a quantidade de valores discretos que as saídas do *CMAC* podem assumir

$\hat{x}$ -valor discreto para uma saída do *CMAC* com pesos discretos

K -relação entre as amplitudes $(w_m - w_{-m})$ e $(r_n - r_{-n})$

S_1 -soma dos números inteiros que definem os pesos responsáveis por uma saída do *CMAC*

θ -posição angular do pêndulo forçado

ω -velocidade angular do pêndulo forçado

α -aceleração angular do pêndulo forçado

r -passo de integração no processo de **Euler**



Capítulo 1

INTRODUÇÃO

A busca de novas alternativas para soluções de problemas no processamento de informações inexatas tais como reconhecimento de padrões, otimização de trajetórias, identificação e controle de sistemas não lineares, sistemas variáveis em sua estrutura e/ou sujeitos a distúrbios, nos incentiva a estudar os caminhos seguidos pelas redes neuronais biológicas para resolução desta classe de problemas que, quando tratados através de teorias convencionais apresentam soluções inviáveis ou tão particulares que descaracterizam o problema inicial.

Qual é o tempo necessário para concluirmos que as constelações que vemos no céu assemelham-se realmente a escorpiões, cães ou caçadores? Qual é o custo para escolhermos o menor caminho que passa por várias localidades? Com que rapidez o cerebelo determina os impulsos nervosos responsáveis pelos movimentos dos globos oculares a centenas de graus por segundo para focalizarem objetos sob as mais variadas condições?

Durante as últimas três décadas, pesquisadores vêm aprimorando técnicas baseadas na filosofia de processamento das



redes neuronais biológicas.

Este trabalho apresenta uma descrição das principais arquiteturas de controle com redes neuronais artificiais e, também uma descrição completa do **Cerebellar Model Articulation Controller (CMAC)**, mostrando suas vantagens, desvantagens e aplicações em identificação de funções, através de exemplos ilustrativos.

Propomos uma nova técnica para superar o problema da grande quantidade de memória requerida pelo **CMAC**, baseada na discretização dos valores que os seus pesos podem assumir.

Simulações realizadas neste trabalho mostram que o **CMAC** com pesos discretos é capaz de reproduzir uma função de duas variáveis com maior precisão do que o **CMAC** provido com código **HASH**, usando uma memória física de tamanho similar.

No capítulo 2 é apresentada uma descrição geral das Redes Neuronais Artificiais (**RNA**). O capítulo 3 descreve as estruturas mais utilizadas no controle automático com **RNA**. São descritos no capítulo 4 os principais resultados disponíveis sobre o **CMAC** e no capítulo 5 as aplicações em controle automático. Nos capítulos 6 e 7 são apresentados os resultados desenvolvidos neste trabalho, sobre o **CMAC** com pesos discretos e suas aplicações, inclusive no controle inverso da posição de



um pêndulo forçado. Nos capítulos 8 e 9 são descritas as conclusões do trabalho e as referências bibliográficas.



Capítulo 2

REDES NEURONAIS ARTIFICIAIS

2.1. INTRODUÇÃO

Uma Rede Neuronal Artificial (**RNA**) consiste de uma arquitetura de processamento de informações baseada nas redes neuronais biológicas e possui habilidades de identificação de funções e classificação de padrões. Similarmente às redes neuronais biológicas, as **RNA** assimilam informações pela técnica de tentativa e erro. Isto significa que podem aprender mediante a apresentação de exemplos. As **RNA** também possuem capacidade de generalização, ou seja, de apresentar soluções satisfatórias nas situações para as quais não houve aprendizado[1][2]; crianças reconhecem números num relógio digital embora tenham aprendido os algarismos arábicos na forma manuscrita. Outra característica das **RNA** é a de não perder a informação já armazenada caso ocorra a destruição total de uma parte da rede. Isto pode ser verificado no funcionamento normal do sistema nervoso humano caso alguns milhões de neurônios sejam anulados.



2.2. ELEMENTO LINEAR ADAPTATIVO

O elemento básico de uma RNA é o neurônio artificial, inspirado no modelo do neurônio biológico idealizado por *McCulloch* e *Pitts* em 1943. A figura 2.1 ilustra o neurônio artificial chamado **ADALINE** (**ADaptive LINEar Element**).

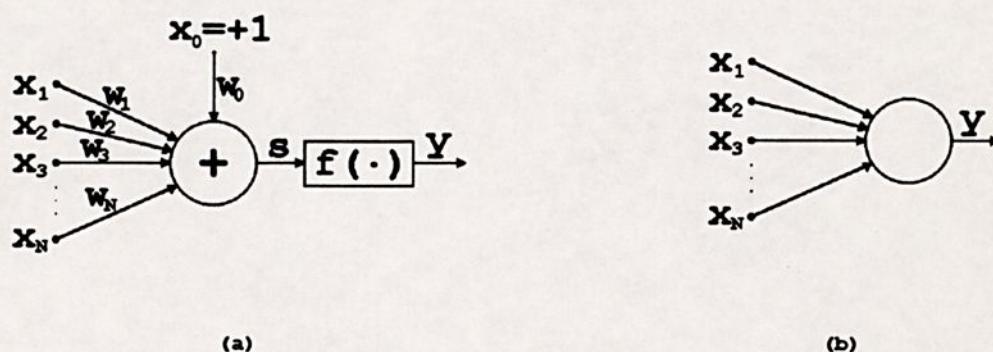


Figura 2.1: Elemento Linear Adaptativo (ADALINE). (a) Diagrama de blocos; (b) Símbolo.

Os elementos do vetor de entrada $X = [x_1 \ x_2 \ \dots \ x_N]^T$ pertencem a conjuntos de valores contínuos ou discretos. A função de saída do neurônio (f) é do tipo não decrescente com domínio real cuja saída y se encontra no intervalo $[0,1]$ quando se deseja que o neurônio responda com sinais de reforço numa maior ou menor intensidade, ou, ainda, y no intervalo $[-1,1]$ quando a saída do neurônio tiver que assumir caráter de reforço ou de inibição numa forma mais ou menos intensa. A figura 2.2 mostra alguns exemplos de funções de saída de **ADALINE**.

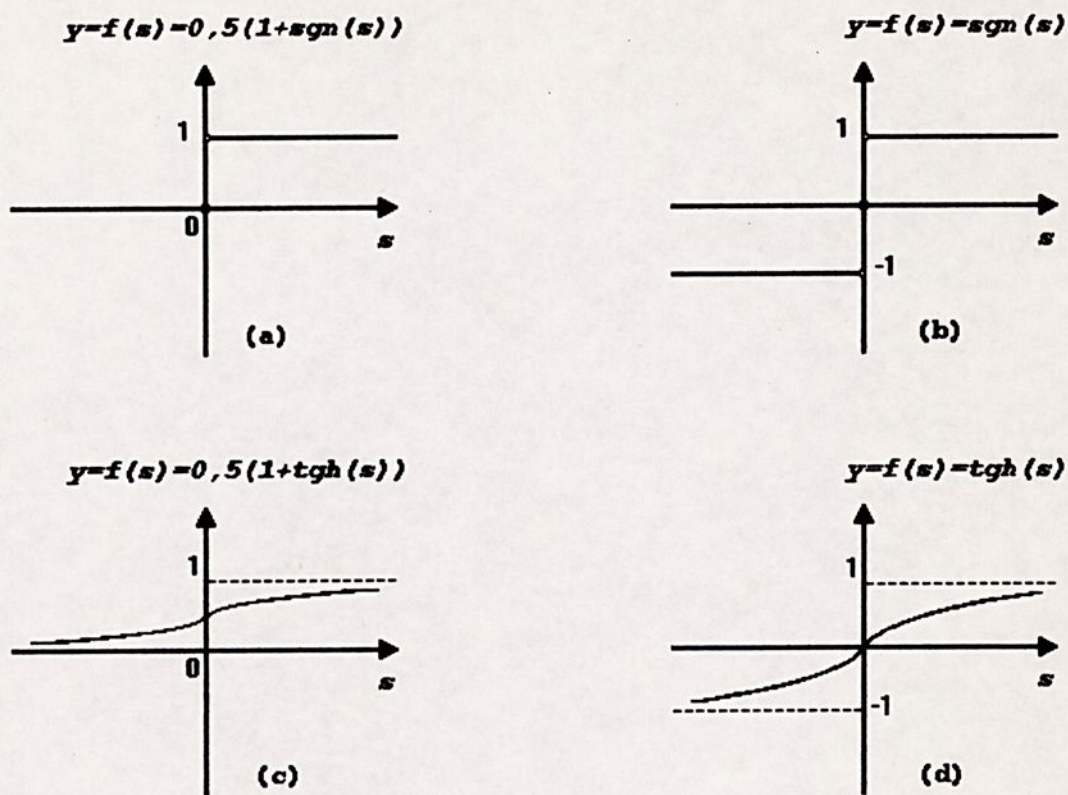


FIGURA 2.2: Funções de saída de ADALINE.

A saída do neurônio é computada através das equações (2.1) e (2.2):

$$s = w_0 + \sum_{i=1}^N x_i w_i, \quad (2.1)$$

$$y = f(s). \quad (2.2)$$

A entrada $x_0=1$ é denominada **bias** e seu peso w_0 determina a saída do neurônio quando $X=0$. Analiticamente, a equação (2.1) determina um hiperplano no R^N onde $W_N = [w_1 \ w_2 \dots w_N]^T$

define seu vetor ortogonal, não necessariamente com módulo unitário. O peso w_0 corresponde ao termo independente da equação geral do hiperplano e proporciona o seu deslocamento em relação à origem, o que é muito importante em classificação de padrões.

A figura 2.3 ilustra um **ADALINE** com duas entradas e três pesos classificando os pontos de treinamento (padrões) de acordo com as saídas desejadas, geradas pela função definida na

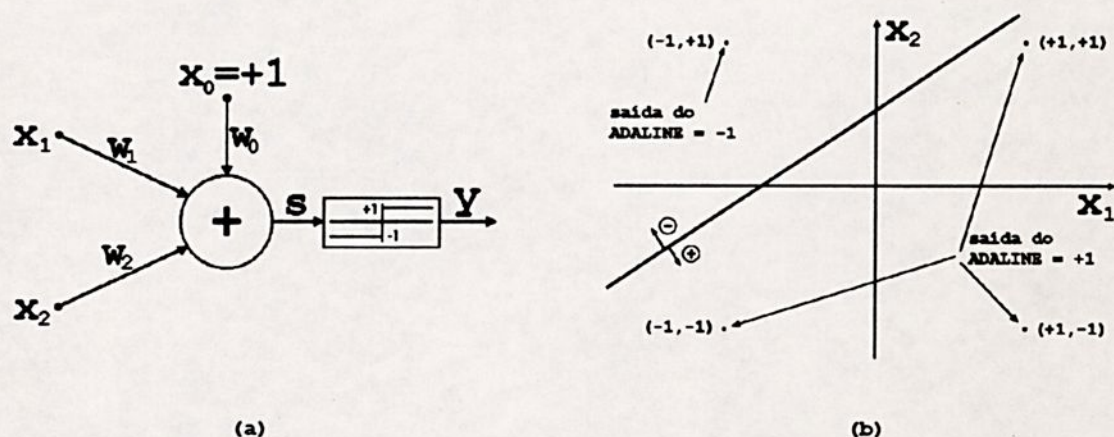


FIGURA 2.3: ADALINE com duas entradas e função de saída $f(s) = \text{sgn}(s)$. (a) diagrama de blocos; (b) hiperplano unidimensional definido por $W = [w_0, w_1, w_2]^T$ separando o espaço (x_1, x_2) em dois semi-espacos onde a saída do ADALINE será +1 ou -1.

tabela 2.1. A equação geral e a equação reduzida do hiperplano separador são dadas pelas equações 2.3.

Note que as saídas da função descrita na tabela 2.1 são separáveis por uma variedade linear do $\mathbb{R}^2$. Dizemos que esta função é linearmente separável. Segundo **Widrow & Lehr[1]**

TABELA 2.1: Função reproduzida pelo ADALINE da figura 2.3.

ponto de entrada (x_1, x_2)	saída desejada (d)
(+1, +1)	+1
(+1, -1)	+1
(-1, -1)	+1
(-1, +1)	-1

$$w_2 x_2 + w_1 x_1 + w_0 = 0$$

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2} \quad (2.3)$$

qualquer função linearmente separável com N variáveis pode ser reproduzida por um **ADALINE** com N entradas.

2.3. ALGORITMOS UTILIZADOS NO TREINAMENTO DO **ADALINE**

O treinamento ou aprendizado de um **ADALINE** é um procedimento que armazena informação nos seus pesos, de modo que sua saída corresponda a valores desejados para vetores de entrada $X^{(i)}$ tomados num conjunto de interesse com NP elementos. O aprendizado termina quando o erro entre a saída da **RNA** e seu valor desejado atinge um valor tolerável. O treinamento de **ADALINE** baseia-se no princípio da mínima perturbação: a alteração que se faz nos pesos para corrigir o erro devido a um ponto de treinamento não deve descaracterizar a informação já assimilada



pela RNA.

Descreveremos, a seguir, algoritmos usais para o treinamento de **ADALINE** descrito em [1]. Um algoritmo de treinamento de RNA com mais de um **ADALINE** será abordado no item 2.4.

2.3.1. Regra Delta de Widrow-Hoff

Também conhecido por α -LMS, este algoritmo [1] baseia-se na alteração dos pesos w_i através das equações (2.4) e (2.5):

$$w_{k+1} = w_k + \alpha \frac{e_k X_k}{|X_k|^2}, \quad (2.4)$$

$$e_k \triangleq d_k - w_k^T X_k, \quad (2.5)$$

sendo w_{k+1} e w_k os vetores pesos nas iterações $(k+1)$ e (k) respectivamente, X_k é o vetor de entrada do **ADALINE** e α é a taxa de aprendizado que pode assumir valores no intervalo $0 < \alpha < 2$ como será mostrado a seguir. O erro linear e_k é definido como a diferença entre a resposta desejada d_k e a saída linear $s_k = w_k^T X_k$ anterior à adaptação.

Uma alteração nos pesos leva a uma correspondente alteração no erro. Combinando as equações (2.4) e (2.5), vemos que a modificação no erro é dada pela equação (2.6).



$$\Delta e_k = e_{k+1} - e_k$$

$$\Delta e_k = \Delta (d_k - \mathbf{w}_k^T \mathbf{x}_k) = (d_k - \mathbf{w}_{k+1}^T \mathbf{x}_k) - (d_k - \mathbf{w}_k^T \mathbf{x}_k)$$

$$\Delta e_k = -\mathbf{x}_k^T \Delta \mathbf{w}_k, \quad \Delta \mathbf{w}_k = \mathbf{w}_{k+1} - \mathbf{w}_k$$

$$\Delta e_k = -\alpha \frac{e_k \mathbf{x}_k^T \mathbf{x}_k}{|\mathbf{x}_k|^2}$$

$$\Delta e_k = -\alpha e_k \quad (2.6)$$

O parâmetro α controla a estabilidade e a velocidade de convergência do processo de treinamento. Para padrões de treinamento invariantes no tempo ($\mathbf{x}_k$ e d_k constantes) a estabilidade estará assegurada se escolhermos $0 < \alpha < 2$, pois neste caso, $\Delta e_k = e_{k+1} - e_k$. Valores de α maiores do que 1 diminuem a velocidade de convergência, portanto, recomenda-se $0 < \alpha < 1$.

2.3.2. Algoritmo Backward Error Propagation (Backpropagation)

Backpropagation é um algoritmo do tipo gradiente descendente e não linear[1] que exige funções de saída diferenciáveis tais como:

$$y = f(s) = \tanh(s) \quad \text{ou}$$

$$y = f(s) = 0,5[1 + \tanh(s)],$$

indicadas na figura 2.2. O objetivo é minimizar o erro quadrático



médio entre a saída do **ADALINE** e o seu valor desejado, para cada ponto de treinamento. Consideremos E_k o erro quadrático médio na k -ésima iteração:

$$E_k = \frac{1}{2} \sum_{i=1}^{NP} [e_k^{(i)}]^2 \quad (2.7)$$

$$e_k^{(i)} = d^{(i)} - y_k^{(i)}$$

obtido pelas diferenças entre as saídas do **ADALINE** e seus respectivos valores desejados avaliadas para cada um dos **NP** pontos de treinamento. Na equação (2.7), E_k é um funcional convexo e apresenta um mínimo global no ponto determinado pelos pesos que levam a

$$\frac{\partial E_k}{\partial W} = 0,$$

sendo W o vetor dos pesos da rede neuronal.

O algoritmo **Backpropagation** consiste em modificar os pesos do **ADALINE** segundo a equação (2.8):

$$W_{k+1} = W_k + \mu (-\nabla E_k). \quad (2.8)$$

sendo que W_k e W_{k+1} são os vetores pesos na iteração k e $k+1$, respectivamente, μ é a constante de aprendizado e, usualmente,



$0 < \mu < 1$, ∇E_k é o vetor gradiente do funcional convexo E_k dado na equação (2.7), relativamente aos pesos $[w_0 \ w_1 \ w_2 \ \dots \ w_N]$.

Aplicando o algoritmo (2.8) aos pesos do **ADALINE**, teremos, considerando a figura (2.1):

$$\begin{aligned} W_{k+1} &= W_k - \mu \left(\frac{\partial E_k}{\partial y_k^{(i)}} \frac{\partial y_k^{(i)}}{\partial s_k^{(i)}} \frac{\partial s_k^{(i)}}{\partial W_k} \right), \\ W_{k+1} &= W_k - \mu \left(\sum_{i=1}^{NP} (-e_k^{(i)} \dot{f}(s_k^{(i)}) X_k^{(i)}) \right), \quad \dot{f}(s) = \frac{\partial f(s)}{\partial s}, \\ W_{k+1} &= W_k + \mu \sum_{i=1}^{NP} (e_k^{(i)} \dot{f}(s_k^{(i)}) X_k^{(i)}). \end{aligned} \quad (2.9)$$

A equação (2.9) é a regra de atualização a cada iteração, após a apresentação de todos os pontos de treinamento(*i*) dentro da mesma iteração(*k*). Outra forma de minimizar a função E_k [1] é minimizando $e_k^{(i)}$ pela aplicação da equação (2.10) a cada nova apresentação de um ponto de treinamento(*i*) para cada iteração(*k*). Se executarmos uma correção para cada ponto de treinamento, teremos **NP** aplicações da equação (2.10). Em algumas situações este procedimento nos leva a uma formulação mais simples para o algoritmo de treinamento em questão[10].

$$W_{k+1} = W_k + \mu e_k^{(i)} \dot{f}(s_k^{(i)}) X_k^{(i)} \quad (2.10)$$

O uso de **ADALINE** com função de saída $f(s)=s$ mostra um caso particular do **Backward Error Propagation**, similar ao



algoritmo α -LMS, chamado μ -LMS [1], representado pela equação (2.11):

$$W_{k+1} = W_k + \mu e_k^{(1)} X_k^{(1)}. \quad (2.11)$$

Pode-se notar que a diferença entre as regras α -LMS e μ -LMS é a normalização devido à divisão por $|X_k|^2$ na equação (2.4). Outro ponto a ser observado é que o α -LMS é uma técnica tipo correção de erro enquanto que o μ -LMS é um algoritmo baseado em gradiente descendente.

2.4. ESTRUTURAS DE REDES NEURONAIS ARTIFICIAIS

Em geral, um único **ADALINE** não resolve os problemas mais complexos[1][2]. Uma solução consiste em organizar vários **ADALINE** de modo a formar uma camada ou ainda, várias camadas, construindo estruturas chamadas **MADALINE** (**MultiADALINE**). A figura 2.4 mostra alguns tipos de **RNA** que se pode construir com **ADALINE**[2].

Uma **RNA** tipo **feedforward** é aquela onde a informação segue um único sentido desde a entrada até a saída, como mostram as figuras 2.4(a) e 2.4(b). Uma rede recorrente é construída por conexões entre **ADALINE** numa mesma camada como na Rede de **Hopfield** (figura 2.4(c)). Outra forma alternativa



é conectar a saída de uma **RNA feedforward** às entradas de um ou mais neurônios de alguma(s) camada(s) desta **RNA**.

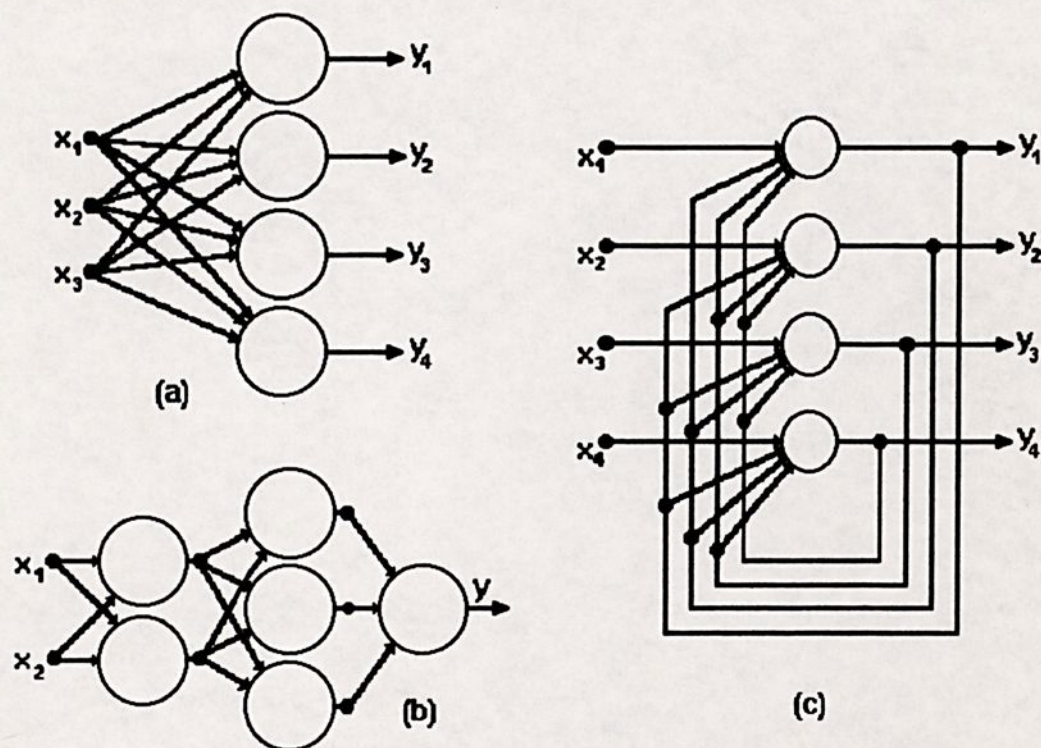


FIGURA 2.4: Exemplos de RNA: (a) **MADALINE** com uma camada e quatro neurônios; (b) **MADALINE** com três camadas e seis neurônios; (c) **Rede de Hopfield**.

Embora não se conheça regras práticas para o projeto de uma **RNA** que resolva um determinado problema, em [11] encontramos um resultado sobre a capacidade das **RNA**: qualquer função contínua sobre um conjunto compacto pode ser aproximada, com uma precisão arbitrariamente especificada, por uma **RNA feedforward** com três camadas. Isto implica que: dado qualquer $\xi > 0$, uma rede neuronal com três camadas e com número suficientemente grande de neurônios pode ser treinada de modo que

$$\|H(x) - NN(x)\| < \xi \text{ para todo } x \in S,$$

sendo H a função que desejamos reproduzir, $NN(x)$ é a saída da rede neuronal e S é um domínio compacto de um espaço vetorial de dimensão finita.

2.5. TREINAMENTO DE REDES NEURONAIS ARTIFICIAIS

O treinamento das **RNA feedforward** pode ser feito pelas técnicas **Madaline Rule I (MRI)**, **Madaline Rule II (MRII)** (correção de erro) ou pela técnica **Backward Error Propagation** (gradiente descendente) [1]. Redes recorrentes tais como a rede de **Hopfield** possuem um algoritmo de treinamento próprio, também descrito neste item.

2.5.1. Madaline Rule I (MRI)

O algoritmo **Madaline Rule I** destina-se ao treinamento de **RNA** com duas camadas, sendo a primeira delas constituída por três ou mais **ADALINE** com função de saída $f(s^{(i)}[1]) = \text{sgn}(s^{(i)}[1])$ e a segunda camada formada apenas por um **ADALINE** com pesos fixos que implementa uma função lógica tipo **OU**,



E ou **MAJ**. Estas funções podem ser definidas da seguinte forma, considerando que as entradas x_i , $i=1,2,\dots,N$, podem assumir somente os valores $+1$ e -1 :

$$x_1 \text{ E } x_2 \text{ E } \dots \text{ E } x_N = \begin{matrix} 1, \text{ se } x_1 = x_2 = \dots = x_N = 1 \text{ e} \\ -1, \text{ caso contrário ;} \end{matrix}$$

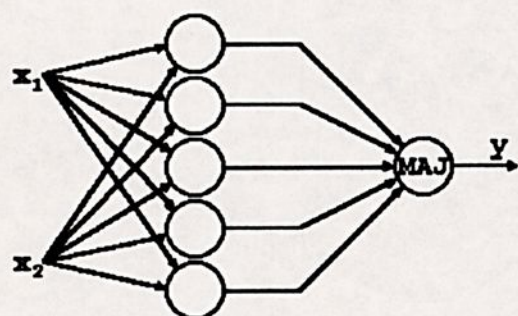
$$x_1 \text{ OU } x_2 \text{ OU } \dots \text{ OU } x_N = \begin{matrix} 1, \text{ se } x_1 \text{ ou } x_2 \text{ ou } \dots \text{ ou } x_N = 1 \text{ e} \\ -1, \text{ caso contrário ;} \end{matrix}$$

$$\text{MAJ}(x_1, x_2, \dots, x_N) = \begin{matrix} 1, \text{ se } x_1 + x_2 + \dots + x_N > 0 \text{ e} \\ -1, \text{ caso contrário .} \end{matrix}$$

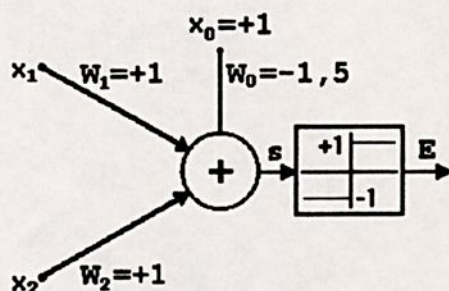
A figura 2.5 ilustra este tipo de **RNA**. A figura 2.5(a) apresenta uma arquitetura conhecida por **MADALINE I**, idealizada na década de 60 [1].

O treinamento se resume em avaliar a saída atual da rede e, no caso desta não corresponder ao valor desejado, verificar qual(is) dos **ADALINE** da primeira camada contribuem para este resultado incorreto. Por exemplo, se a saída desejada para a **RNA**, supostamente com uma função **MAJ** na segunda camada, é $+1$ e os três primeiros neurônios da primeira camada apresentam saída -1 e, ainda, os dois últimos neurônios apresentam saída $+1$ logo, os três primeiros **ADALINE** contribuem para o resultado incorreto. Dentre os neurônios selecionados na primeira camada, deve-se

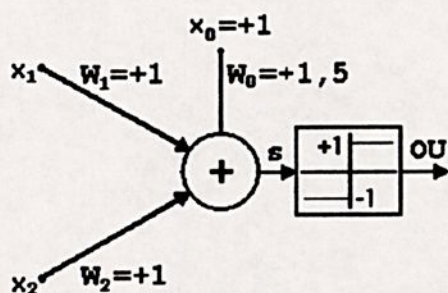
corrigir os pesos daquele que apresentar saída linear $s_k^{[1]} = W_k^{T[1]} X_k^{(1)[1]}$ mais próxima de zero, com isso estaremos atuando sobre o neurônio que está na iminência de obter uma saída lógica $+1$ e, conseqüentemente, levar a saída da função **MAJ** para o valor desejado $+1$.



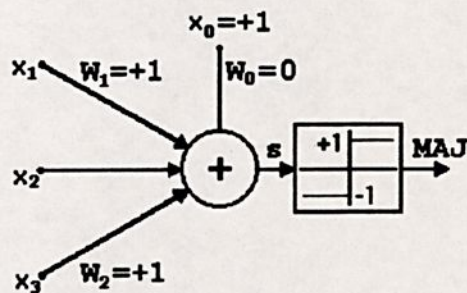
(a)



(b)



(c)



(d)

FIGURA 2.5: Rede MADALINE I e respectivas funções de saída. (a) MADALINE com duas camadas sendo a que última camada consiste de um ADALINE que implementa uma função lógica fixa que pode ser uma das operações: (b) produto E, (c) soma OU, ainda (d) MAJ.

A correção nos pesos da primeira camada é feita pelo algoritmo α -LMS, descrito em 2.3.1, a saber:

$$W_{k+1} = W_k + \alpha \frac{e_k X_k}{|X_k|^2},$$

$$e_k \triangleq d_k - W_k^T X_k.$$

A saída desejada d_k é determinada por um mecanismo inerente ao **MRI** que a partir do valor da saída desejada para a **RNA**, das saídas lineares $s_k^{(j)}$ e do tipo de função lógica da segunda camada, atribui as saídas corretas para os **ADALINE** da primeira camada[1].

2.5.2. Madaline Rule II (MRII)

O algoritmo **Madaline Rule II (MRII)** destina-se ao treinamento de **RNA** com a arquitetura mostrada na figura 2.6. Da mesma forma que em **MRI**, avalia-se o erro da apresentação na saída da **RNA**. Calcula-se esse erro através da equação 2.12:

$$E_k = \sum [e_k^{(l)}]^2, \quad (2.12)$$

$$e_k^{(l)} = d_k^{(l)} - y_k^{(l)},$$

sendo que l indica um **ADALINE** na última camada e k especifica o ponto de treinamento tomado na k -ésima iteração do processo de treinamento, ou seja, uma iteração consiste em corrigir o erro referente à apresentação de um único ponto de treinamento.



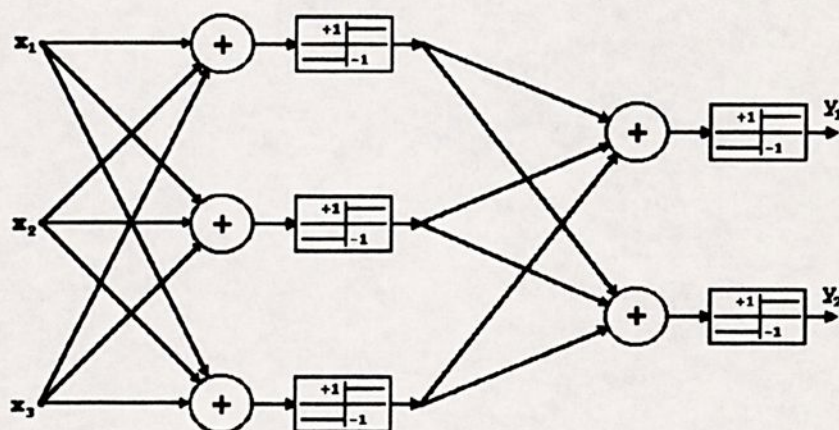


FIGURA 2.6: Arquitetura MADALINE II com duas camadas e cinco neurônios.

Caso a saída da RNA não corresponda ao valor desejado, escolhe-se o **ADALINE** da primeira camada que apresente saída linear $s_{k(j)}^{[1]} = W_k^{[1]} X_{k(j)}^{[1]}$ mais próxima de zero. Analisa-se a sensibilidade de E_k em relação à saída binária $y_{k(j)}^{[1]}$ do neurônio escolhido. Se a mudança no estado lógico de $y_{k(j)}^{[1]}$ reduz E_k , os pesos $W_{k(j)}^{[1]}$ são corrigidos pelo algoritmo α -LMS de forma a inverter a saída $y_{k(j)}^{[1]}$. Se a inversão em $y_{k(j)}^{[1]}$ não reduz E_k nenhuma correção é aplicada aos pesos $W_{k(j)}^{[1]}$. Repete-se o procedimento para cada **ADALINE** da primeira camada com saída linear suficientemente próxima de zero, respeitando uma distância máxima escolhida a fim

de que as correções nos pesos sejam as menores possíveis ao tentarmos mudar o sinal da saída $y_{k(j)}^{[1]}$. Isto é feito para garantir o princípio da mínima perturbação, exposto em 2.3. É possível, se desejado, analisar a sensibilidade de E_k em relação às saídas de dois, três ou mais **ADALINE** da primeira camada simultaneamente. Após terminada esta busca exaustiva na primeira camada, procede-se da mesma forma na segunda camada.

Quando todos os **ADALINE** tiverem sido abordados, toma-se o próximo ponto de treinamento, usualmente ordenados aleatoriamente, e repete-se o procedimento. O objetivo é minimizar E_k durante as várias apresentações dos pontos de treinamento.

2.5.3. *Backward Error Propagation (Backpropagation)*

O algoritmo **Backpropagation** é uma técnica tipo gradiente descendente para minimização do erro quadrático médio na saída de uma **RNA feedforward** com funções de saída de **ADALINE** diferenciáveis. Um exemplo deste tipo de rede é mostrado na figura 2.7.

Da mesma forma que para um único **ADALINE**, deseja-se minimizar o erro E_k na saída da **RNA** porém, deve-se redefinir o erro quadrático médio para abordar várias saídas. Assim, temos a equação 2.13, sendo que k indica a iteração corrente, NP é a



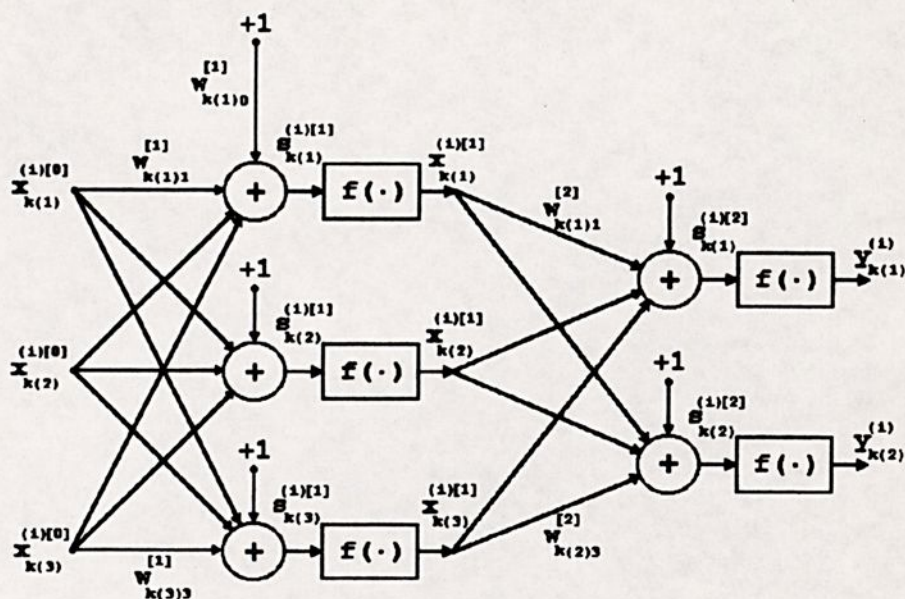


FIGURA 2.7: RNA MADALINE feedforward com duas camadas e cinco neurônios. Cada neurônio tem função de saída $f(s)$ diferenciável, por exemplo, $f(s)$ pode ser uma das funções das figuras 2.2(c) ou 2.2(d).

quantidade de pontos de treinamento, N_L é a quantidade de neurônios na última camada (L).

$$E_k = \frac{1}{2} \sum_{i=1}^{NP} \sum_{j=1}^{N_L} [e_{k(j)}^{(i)[L]}]^2, \quad (2.13)$$

$$e_{k(j)}^{(i)[L]} = d_{(j)}^{(i)} - y_{k(j)}^{(i)}.$$

Para treinar a RNA MADALINE feedforward, a equação

$$\mathbf{W}_{k+1} = \mathbf{W}_k + \mu (-\nabla E_k)$$

deve ser aplicada a cada peso da **RNA**. A dificuldade é descobrir os valores desejados para as saídas dos **ADALINE** das camadas anteriores à última camada e, com esse objetivo, usaremos a notação indicada na figura (2.8).

A equação de atualização dos pesos da última camada é semelhante àquela desenvolvida para um único **ADALINE** pois, a saída desejada $d_{(j)}^{(1)}$ é, supostamente, conhecida. Portanto, os pesos da última camada são modificados pela equação (2.14).

$$w_{k+1(j)q}^{[L]} = w_{k(j)q}^{[L]} - \mu \left(\frac{\partial E_k}{\partial y_{k(j)}^{(1)}} \frac{\partial y_{k(j)}^{(1)}}{\partial s_{k(j)}^{(1)[L]}} \frac{\partial s_{k(j)}^{(1)[L]}}{\partial w_{k(j)q}^{[L]}} \right),$$

$$w_{k+1(j)q}^{[L]} = w_{k(j)q}^{[L]} - \mu \left(- \sum_{i=1}^{NP} \sum_{j=1}^{N_L} e_{k(j)}^{(1)[L]} \dot{f}(s_{k(j)}^{(1)[L]}) x_{k(q)}^{(1)[L-1]} \right), \quad \dot{f} = \frac{\partial f(s)}{\partial s},$$

$$w_{k+1(j)q}^{[L]} = w_{k(j)q}^{[L]} + \mu \sum_{i=1}^{NP} \sum_{j=1}^{N_L} (e_{k(j)}^{(1)[L]} \dot{f}(s_{k(j)}^{(1)[L]}) x_{k(q)}^{(1)[L-1]}).$$

Definindo o erro derivativo do neurônio j da camada de saída L durante a apresentação do ponto de treinamento i na k -ésima iteração, $\delta_{k(j)}^{(1)[L]}$:

$$\delta_{k(j)}^{(1)[L]} = e_{k(j)}^{(1)[L]} \dot{f}(s_{k(j)}^{(1)[L]}),$$

$$w_{k+1(j)q}^{[L]} = w_{k(j)q}^{[L]} + \mu \sum_{i=1}^{NP} \sum_{j=1}^{N_L} \delta_{k(j)}^{(1)[L]} x_{k(q)}^{(1)[L-1]}, \quad (2.14)$$

sendo que L é um número natural que especifica a última camada da RNA, e N_L é a quantidade de neurônios da última camada. A equação (2.14) deve ser aplicada a cada peso $w_{k(j)q}^{[L]}$, que conecta o j -ésimo neurônio da L -ésima camada ao q -ésimo neurônio da $(L-1)$ -ésima (penúltima) camada.

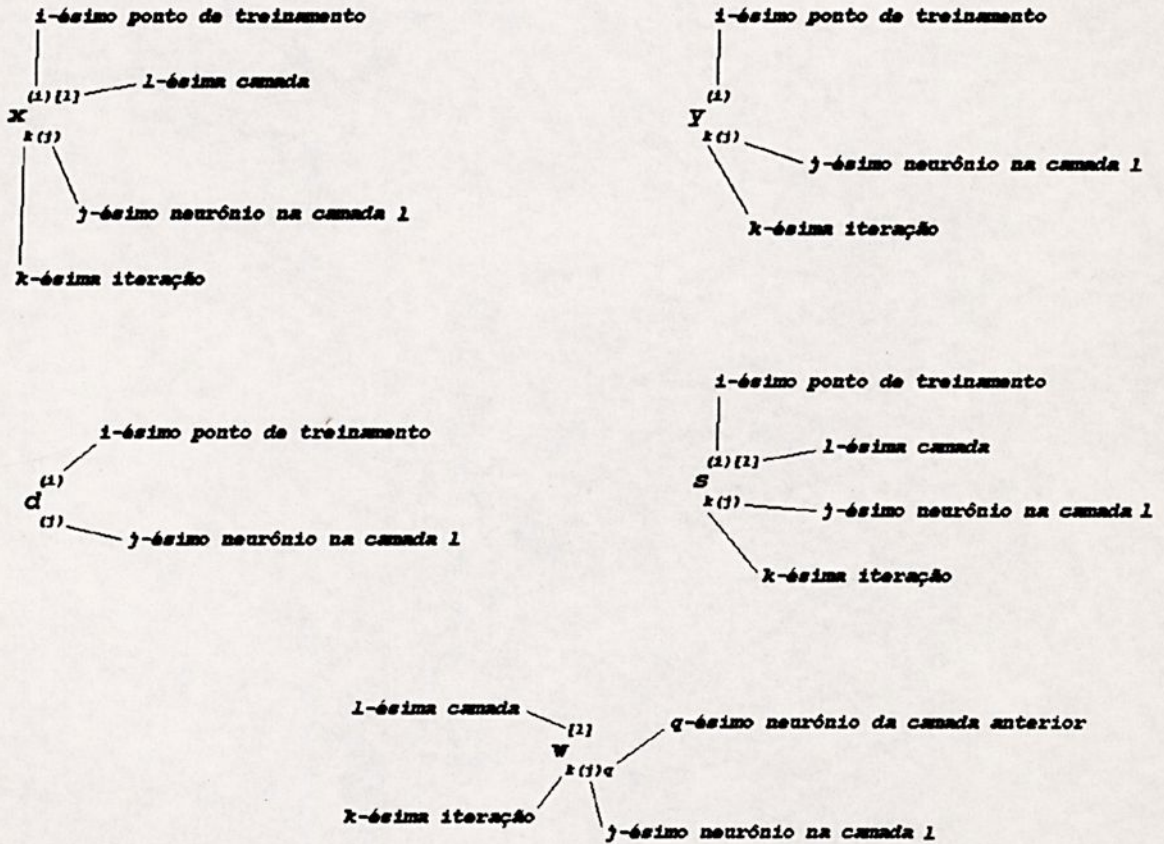


FIGURA 2.8: Notação utilizada para o desenvolvimento do algoritmo Backpropagation. x é a saída de um neurônio de uma determinada camada que não seja a última; considera-se que a saída da camada zero é o vetor de entrada da RNA. y é a saída de um dos neurônios da última camada e, portanto, uma das saídas da RNA. d é o valor desejado (conhecido) para a saída y . s é a saída linear de um neurônio da RNA. w é o peso de um neurônio conectado a uma das saídas da camada anterior.

Para modificar os pesos dos neurônios da penúltima camada, precisamos conhecer o gradiente de E_k em relação a esses pesos. A equação (2.15) fornece esta informação:

$$\frac{\partial E_k}{\partial w_{k(q)g}^{[L-1]}} = \frac{\partial E_k}{\partial y_{k(j)}^{(1)}} \frac{\partial y_{k(j)}^{(1)}}{\partial s_{k(j)}^{(1)[L]}} \frac{\partial s_{k(j)}^{(1)[L]}}{\partial x_{k(q)}^{(1)[L-1]}} \frac{\partial x_{k(q)}^{(1)[L-1]}}{\partial s_{k(q)}^{(1)[L-1]}} \frac{\partial s_{k(q)}^{(1)[L-1]}}{\partial w_{k(q)g}^{[L-1]}}$$

$$\frac{\partial E_k}{\partial w_{k(q)g}^{[L-1]}} = - \sum_{i=1}^{NP} \sum_{j=1}^{N_L} (e_{k(j)}^{(1)[L]} \dot{f}(s_{k(j)}^{(1)[L]}) w_{k(j)q}^{[L]}) \dot{f}(s_{k(q)}^{(1)[L-1]}) x_{k(g)}^{(1)[L-2]}$$

$$\delta_{k(j)}^{(1)[L]} = e_{k(j)}^{(1)[L]} \dot{f}(s_{k(j)}^{(1)[L]})$$

$$\frac{\partial E_k}{\partial w_{k(q)g}^{[L-1]}} = - \sum_{i=1}^{NP} \sum_{j=1}^{N_L} \delta_{k(j)}^{(1)[L]} w_{k(j)q}^{[L]} \dot{f}(s_{k(q)}^{(1)[L-1]}) x_{k(g)}^{(1)[L-2]}$$

Definiremos o erro $e_{k(q)}^{(1)[L-1]}$ associado ao q -ésimo neurônio da penúltima camada por:

$$e_{k(q)}^{(1)[L-1]} = \sum_{j=1}^{N_L} \delta_{k(j)}^{(1)[L]} w_{k(j)q}^{[L]}$$

Assim,

$$\frac{\partial E_k}{\partial w_{k(q)g}^{[L-1]}} = - \sum_{i=1}^{NP} e_{k(q)}^{(1)[L-1]} \dot{f}(s_{k(q)}^{(1)[L-1]}) x_{k(g)}^{(1)[L-2]}$$

$$\delta_{k(q)}^{(1)[L-1]} = e_{k(q)}^{(1)[L-1]} \dot{f}(s_{k(q)}^{(1)[L-1]})$$

$$\frac{\partial E_k}{\partial w_{k(q)q}^{[L-1]}} = - \sum_{i=1}^{NP} \delta_{k(q)}^{(i)[L-1]} x_{k(q)}^{(i)[L-2]} \quad (2.15)$$

Neste ponto, podemos escrever o algoritmo **Backpropagation** a ser aplicado a qualquer peso de uma **RNA MADALINE feedforward** dada na figura 2.7:

$$w_{k+1(j)q}^{[1]} = w_{k(j)q}^{[1]} + \mu \sum_{i=1}^{NP} \delta_{k(j)}^{(i)[1]} x_{k(q)}^{(i)[1-1]},$$

$$\delta_{k(j)}^{(i)[L]} = e_{k(j)}^{(i)[L]} \dot{f}(s_{k(j)}^{(i)[L]}),$$

$$e_{k(q)}^{(i)[1-1]} = \sum_{j=1}^{N_1} \delta_{k(j)}^{(i)[1]} w_{k(j)q}^{[1]},$$

$$\delta_{k(q)}^{(i)[1]} = e_{k(q)}^{(i)[1]} \dot{f}(s_{k(q)}^{(i)[1]}),$$

$l=L, L-1, L-2, \dots, 1$; L é o número de camadas da **RNA**;

$j=1, 2, 3, \dots, N_1$; N_1 é o número de neurônios da camada l ;

$q=0, 1, 2, \dots, N_{l-1}$; N_{l-1} é o número de neurônios da camada $l-1$;

$i=1, 2, 3, \dots, NP$, NP é a quantidade de pontos de treinamento.

2.5.4. Treinamento da Rede de Hopfield

Tomemos a rede de **Hopfield** da figura (2.4(c)), com $N = 4$ **ADALINE**. Definimos w_{ij} como o peso do i -ésimo neurônio conectado à saída do j -ésimo neurônio. Suponhamos que as funções de saída de **ADALINE** sejam do tipo $f(s) = \text{sgn}(s)$, da figura (2.2(b))



e que as entradas x_i possam assumir os valores $+1$ ou -1 . O treinamento consiste em aplicar a equação (2.16) aos pesos da RNA [2].

$$w_{ij} = \begin{cases} \sum_{n=1}^{NP} x_i^{(n)} x_j^{(n)}, & i \neq j \\ 0, & i = j, i \geq 0, j < NP \end{cases} \quad (2.16)$$

NP é a quantidade de pontos de treinamento. O treinamento, portanto, não é um processo iterativo mas sim, uma simples atribuição de valores iniciais. Para o reconhecimento de padrões por uma rede de **Hopfield**, deve-se apresentar o vetor $X = (x_1 \ x_2 \ \dots \ x_N)^T$ à saída da RNA, fazendo $y_{k=0(i)} = x_{(i)}$, $x_{(i)}$ é um elemento do vetor de entrada a ser classificado pela RNA, em seguida, executa-se o cálculo iterativo das saídas finais, dado pela equação (2.17).

$$y_{k+1(j)} = \text{sgn} \left(\sum_{i=0}^N w_{ij} y_{k(i)} \right), \quad 0 \leq j < N \quad (2.17)$$

O final do reconhecimento é caracterizado por uma saída constante nas últimas iterações. **Hopfield** provou que a RNA converge quando $w_{ij} = w_{ji}$, se os pesos são ajustados sem que se obedeça alguma ordem, isto é, de forma assíncrona[2]. O objetivo do reconhecimento é que as saídas da RNA se igualem aos elementos do padrão armazenado pela equação (2.16) que mais se assemelha do padrão a ser classificado. Caso isso não ocorra, as saídas da rede atingirão um outro ponto não similar ou sem significado



perante as informações armazenadas pela equação (2.16).

Outra configuração de **RNA**, adequada ao controle automático de sistemas é o **Cerebellar Model Articulation Controller (CMAC)** [5][6][10]. Como o **CMAC** é o tema principal desta Dissertação; ele é descrito detalhadamente, inclusive com exemplos de aplicação, nos capítulos **4, 5, 6, e 7**. O próximo capítulo apresenta as configurações mais utilizadas para o controle automático utilizando **RNA**.



Capítulo 3

CONTROLE DE SISTEMAS COM REDES NEURONAIS ARTIFICIAIS

3.1. INTRODUÇÃO

A inexistência de uma teoria geral e sistemática para abordagem de problemas de controle não lineares é decorrente da grande diversidade de casos. As **RNA** são ferramentas muito eficientes na reprodução de transformações não lineares e talvez, seja esta sua contribuição mais significativa para a Teoria de Controle[3][11].

A abordagem para o controle de sistemas variantes no tempo, em geral, para manter um certo comportamento desejado, requer o uso de arquiteturas que se adaptem às variações da planta. As **RNA** também satisfazem esta necessidade, já que se ajustam a fim de assimilar o conhecimento adquirido em novas situações. Dizemos então, que as **RNA** podem ser treinadas mediante a apresentação de exemplos.

Entre outras características das **RNA** podemos citar a capacidade de generalização, isto é, solucionar satisfatoriamente um problema desconhecido mas que apresente



certa semelhança a um ou mais exemplo(s) assimilado(s) na etapa de treinamento. Outra característica das **RNA** é que os dados são distribuídos por vários canais de informação antes de serem processados, o que proporciona um intenso paralelismo no processamento das soluções. As **RNA** ainda conseguem guardar a informação assimilada com qualidade satisfatória no caso de ocorrerem defeitos parciais na sua estrutura. A habilidade que as **RNA** possuem no tratamento de problemas com muitas variáveis e a robustez perante defeitos parciais são decorrentes dessa capacidade de processamento paralelo.

3.2. IDENTIFICAÇÃO DE SISTEMAS

Embora a escolha de uma **RNA** adequada, quanto ao tipo e estrutura para solução de um dado problema, ainda seja feita de forma empírica[2], estudos mostram que elas são ferramentas promissoras em identificação, modelagem e controle de sistemas, especialmente para plantas não lineares[11].

3.2.1. Modelagem Direta

O procedimento para o treinamento de uma **RNA** para que ela reproduza as saídas que um sistema fornece para certas entradas é conhecido como modelagem direta (*forward modelling*) [11][4]. A figura (3.1) mostra um diagrama de blocos



usado para esse tipo de modelagem, que também é conhecido por treinamento supervisionado.

Os pontos de treinamento são obtidos diretamente da planta e o erro e entre sua saída e a saída da RNA_n é usado no treinamento. Este procedimento pode estar sujeito à presença de perturbações nos parâmetros da planta (d), que afetem diretamente a resposta da planta, de ruídos (d'), por exemplo, decorrentes do processo de amostragem de y^p . Se a planta for um sistema dinâmico, pode-se introduzir suas entradas e saídas passadas como entradas da RNA , ou, ainda, podemos usar uma RNA recorrente. É importante observar que esta configuração permite identificar plantas não lineares. Por exemplo, suponha que a planta discretizada seja dada pela equação (3.1), sendo T o período de amostragem:

$$y^p(KT) = sen(y^p(KT-T)) + u(KT-T) = f(y^p(KT-T), u(KT-T)) \quad (3.1)$$

Assim, usaríamos como entradas da RNA_n , na figura (3.1) os sinais $u(KT-T)$, $y^p(KT-T)$ e teríamos o erro $e(KT) = y^p(KT) + d'(KT) - y^n(KT)$.



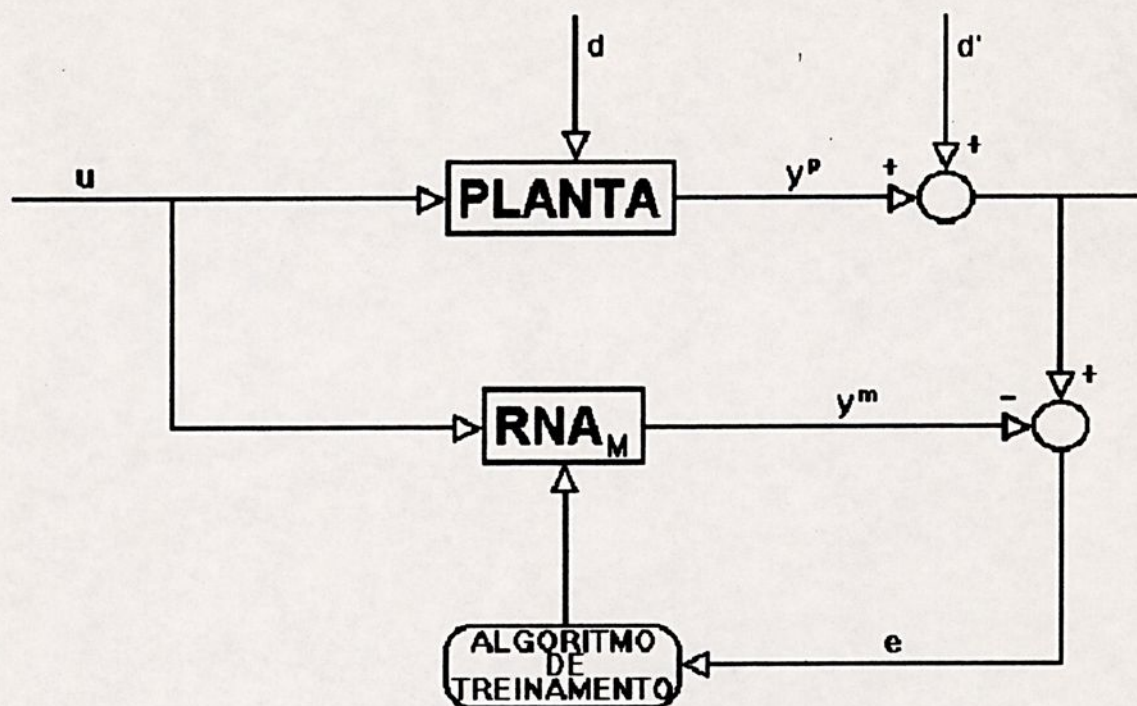


FIGURA 3.1: Estrutura para modelagem direta de sistemas através de RNA.

3.2.2. Modelagem Inversa

A obtenção da dinâmica inversa de um sistema tem um objetivo muito importante em estruturas de controle com RNA, e pode ser obtida através de dois métodos distintos: treinamento inverso generalizado e treinamento inverso especializado[3].

A figura (3.2) mostra o diagrama de blocos para o treinamento inverso generalizado, também conhecido por modelagem inversa direta.

Nessa estrutura, aplica-se uma entrada s à planta e a RNA recebe a saída da planta correspondente. O treinamento tem a finalidade de aproximar a saída da RNA ao sinal s , e quando isso ocorrer, o modelo inverso da planta estará assimilado pela RNA. Este procedimento só é viável se a planta for, matematicamente, uma aplicação biunívoca[3], isto é, dados $s^{(1)}$ e $s^{(2)}$ tais que $s^{(1)} \neq s^{(2)}$, então $y^p(1) \neq y^p(2)$ onde $y^p(i)$ é a resposta que a planta fornece para a entrada $s^{(i)}$, $i = 1$ e 2 . Por exemplo, para a planta dada em (3.1), note que $u(KT)$ é uma função da saída atual $y^p(KT)$ e da saída posterior $y^p(KT+T)$. Assim, durante o aprendizado da RNA na figura (3.2), as suas entradas serão $y^p(KT)$, $y^p(KT+T)$ e a sua saída, $u(KT)$. Após o treinamento, aplicando-se a saída atual $y^p(KT)$ e a desejada $y^p_d(KT+T)$ à RNA, ela fornecerá $u(KT)$ que aplicada na planta fará com que $y^p(KT+T) = y^p_d(KT+T)$.

Como o objetivo é que a saída da planta se comporte de forma desejada, durante o treinamento, deve-se aplicar as entradas s que estejam na região operacional do processo. Mas, a princípio, estes valores de s dificilmente são obtidos, o que nos obriga a tomar valores distribuídos numa extensa faixa. Logo, o controle não é dirigido diretamente ao seu objetivo[3][4].



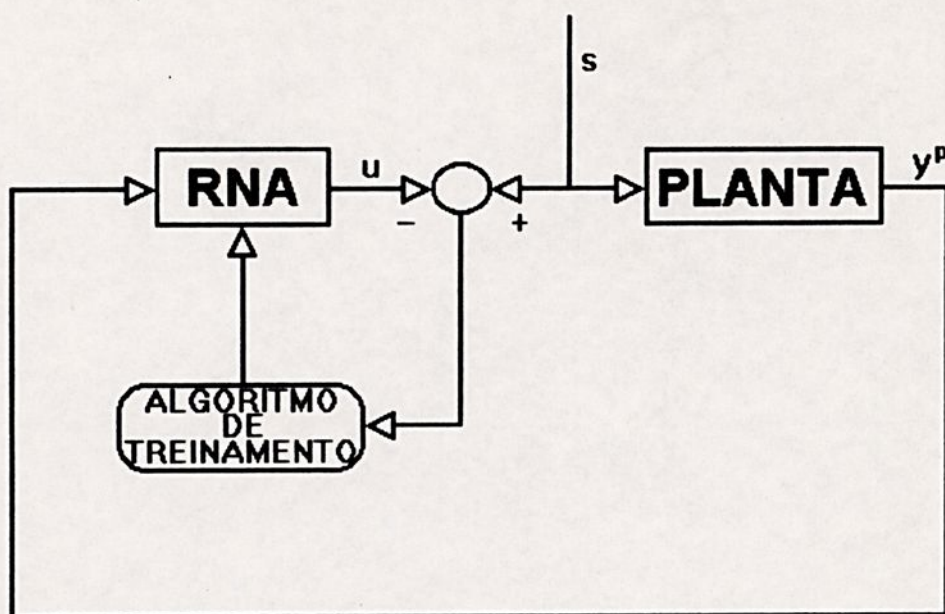


FIGURA 3.2: Estrutura para o treinamento inverso generalizado.

Uma forma de contornar este problema é usar treinamento inverso especializado. A figura (3.3) ilustra o método.

Aqui, o valor desejado para a saída da planta é aplicado à RNA_c e sua saída é aplicada à planta e a uma outra RNA_m já treinada para reproduzir o modelo direto da planta. O erro usado pelo algoritmo de treinamento é calculado a partir do valor desejado para a saída da planta y^d e do valor atual dessa saída y^p . A saída da RNA_m pode ser usada no lugar da saída atual da planta nas situações em que ruídos acompanham esse valor, ou quando não é viável usar o sistema físico. A planta, geralmente,

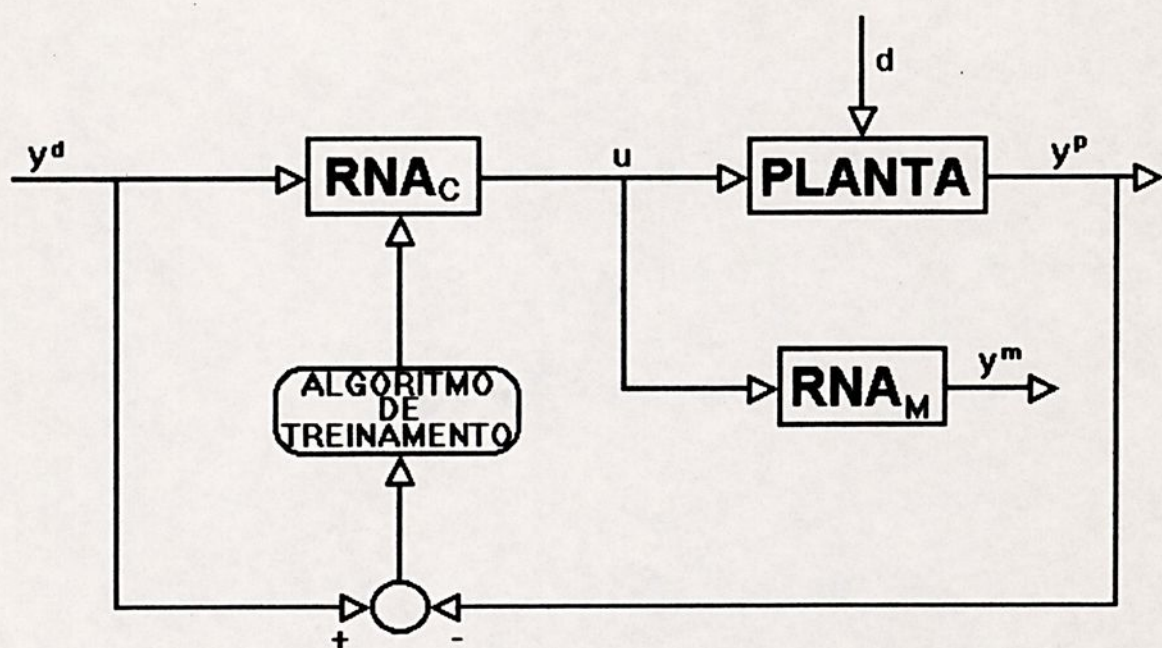


FIGURA 3.3: Estrutura para o treinamento inverso especializado.

está sujeita a perturbações (d) neste tipo de modelagem.

O algoritmo de treinamento deve transformar o erro na variável y^p (saída da planta) para um erro na variável u (saída da RNA_C) pela propagação retroativa através da RNA_M [11], ou por alguma outra aproximação do jacobiano da planta[12].

Comparado ao treinamento inverso generalizado, o caso especializado apresenta duas vantagens: o treinamento é feito na faixa operacional do sistema e, quando este não é uma transformação biunívoca, uma dinâmica inversa particular do

sistema é encontrada[3], pois, supostamente, a planta é localmente estabilizável.

3.3. ESTRUTURAS DE CONTROLE COM RNA

3.3.1. Controle Supervisionado

Nos casos em que a lei de controle é conhecida, seja ela fornecida por um operador humano ou por um sistema especialista, nós devemos apenas treinar a **RNA** com os pares entrada-saída na região operacional da planta.

As entradas da **RNA** correspondem às observações que o operador humano faz durante o processo de controle, as saídas correspondem às ações de controle que são aplicadas ao sistema e o treinamento da **RNA** é feito usando modelagem direta.

3.3.2. Controle Inverso Direto

Essa estrutura de controle consiste em usar a **RNA** como controlador em malha aberta. A modelagem inversa é usada no treinamento da **RNA**, de modo que se sua entrada for y^d , a planta receberá a ação de controle necessária para que sua saída seja y^d . Veja as figuras (3.2) e (3.3).



Como não há realimentação, este tipo de controle não é robusto, porém, com o uso de treinamento inverso especializado *on line* podemos superar este problema.

3.3.3. Controle por Modelo de Referência

Nessa estrutura, a saída da planta deve seguir de forma assintótica o sinal dado por um sistema estável conhecido, que traduz o comportamento desejado do sistema de controle, chamado modelo de referência. A figura (3.4) mostra o diagrama de blocos correspondente.

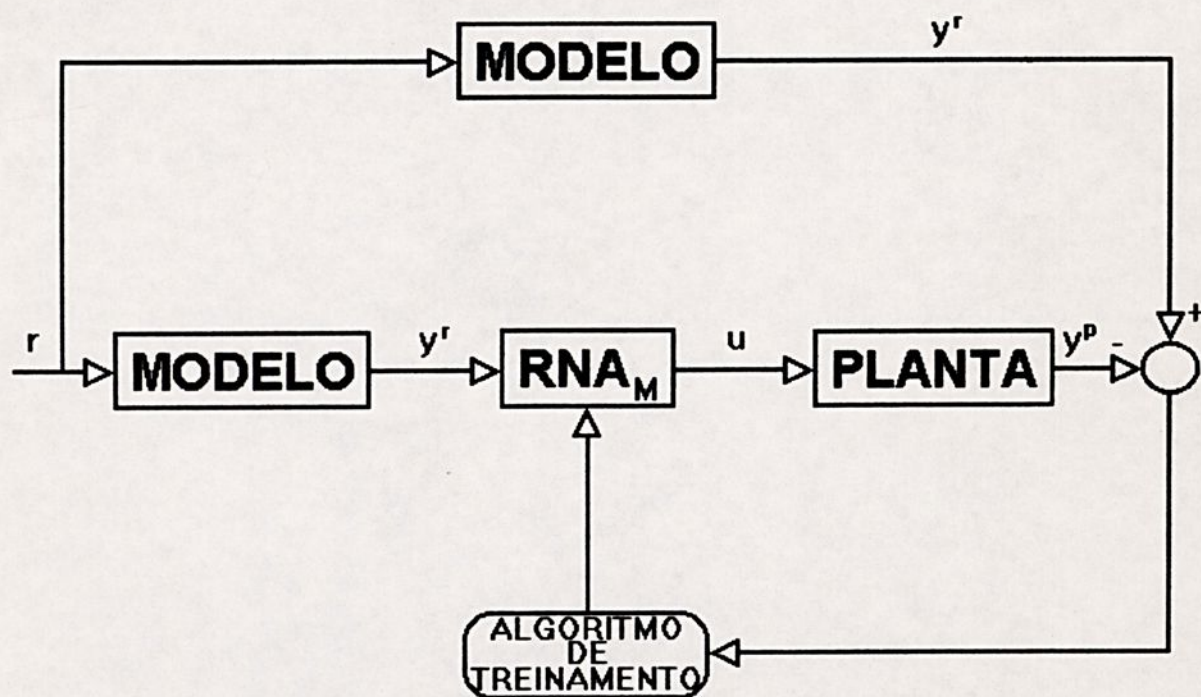


FIGURA 3.4: Arquitetura para o Controle por Modelo de Referência.

O algoritmo de treinamento deve adaptar a RNA_M tal como no treinamento inverso especializado. Quando treinada, a RNA_M deve fornecer a ação de controle necessária para que a saída da planta seja a mesma que a do modelo de referência.

3.3.4. Controle por Modelo Interno

A figura (3.5) mostra a arquitetura de controle por modelo interno.

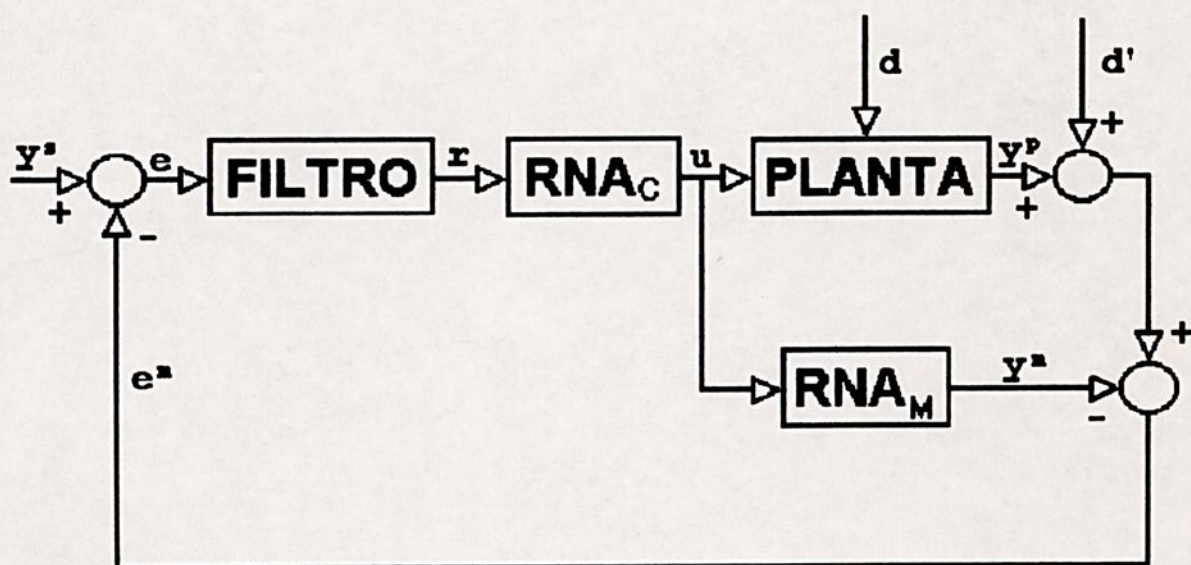


FIGURA 3.5: Arquitetura de Controle por Modelo Interno.

A RNA_M e a RNA_C devem estar adaptadas para representar os modelos direto e inverso da planta,

respectivamente. O erro (e^*) entre as saídas da RNA_m e da planta é processado por um filtro, geralmente de natureza linear, projetado para garantir que o sistema de malha fechada ofereça respostas satisfatórias em regimes transitório e permanente. A saída da planta está sujeita a perturbações vindas do meio em que o sistema se encontra (d) e a distúrbios do processo de amostragem (d') e outros. O objetivo é trabalhar com erro e^* nulo, situação em que a saída da planta seguirá a saída do filtro (r), que se comporta como um modelo conhecido.

3.3.5. *Self Tuning Regulator (STR)*

O diagrama de blocos do algoritmo *Self Tuning Regulator*[8] encontra-se na figura (3.6). A RNA_c e a RNA_m devem reproduzir o modelo inverso da planta no ponto (u_d, y_d) e são adaptadas a partir do mesmo erro e porém, RNA_c tem função de controlador enquanto que a RNA_m deve identificar a planta. O *STR* é um algoritmo executado, *on line* assim, perturbações d nos parâmetros da planta, supostamente desconhecida, não alteram a qualidade de sua resposta pois, a RNA_m detecta as variações da relação entrada-saída da planta e as transfere para a RNA_c . O objetivo é manter o erro e nulo durante o processo de controle.

Neste tipo de arquitetura, é conveniente executar previamente o treinamento generalizado da RNA_c *off line* a fim de



assegurar uma adaptação mais rápida na etapa de controle *on line*.

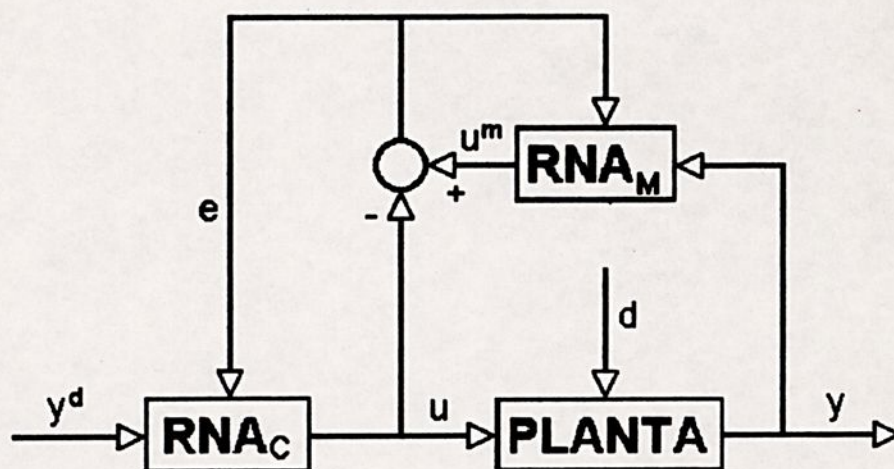


FIGURA 3.6: Estrutura de controle *Self Tuning Regulator*.

3.3.6. Linearização por Realimentação

Tomemos o sistema discreto dado pela equação

(3.2) [11]

$$x(KT+T) = f[x(KT), u(KT)], \quad (3.2)$$

sendo $f(\cdot)$ uma função não linear. Realizando uma mudança de variáveis no espaço de estados, em que:

→ $z = \Phi(x)$, $\Phi(x)$ diferenciável e inversível, z é a nova variável de estado;

→ $u(KT) = \psi[x(KT), v(KT)]$, v é a nova excitação do sistema, então:

$$z(KT+T) = \Phi[x(KT+T)]$$

$$z(KT+T) = \Phi[f(\Phi^{-1}(z(KT)), \psi[\Phi^{-1}(z(KT)), v(KT)])] \quad (3.3)$$

e a transformação dada em (3.3) pode resultar em um sistema linear se o sistema (3.2) for linearizável por realimentação de estado. Caso as funções $\Phi(\cdot)$ e $\psi(\cdot)$ existam, é possível reproduzi-las através de RNA tipo **MADALINE feedforward**. Em [11] são apresentadas condições para a existência das transformações $\Phi(\cdot)$ e $\psi(\cdot)$ de modo que dada a planta (3.2), então (3.3) resulte em um sistema linear.



Capítulo 4

O CEREBELLAR MODEL ARTICULATION CONTROLLER

4.1. INTRODUÇÃO

Albus[5][6] desenvolveu uma **RNA** baseada em estudos neurofisiológicos dos cerebelos de mamíferos com sistema motor aprimorado, tais como o homem, o gato e o esquilo. Essa arquitetura, o **Cerebellar Model Articulation Controller (CMAC)**, é capaz de reproduzir funções não lineares, de múltiplas entradas e múltiplas saídas, definidas num conjunto compacto, isto é, num conjunto fechado e limitado. Um modelo do cerebelo pode ser visto na figura (4.1).

Os comandos vindos de centros de informação de níveis mais altos do cérebro e as informações sobre membros e juntas do sistema motor entram no cerebelo pelas **fibras musgosas**. As **células granulares** transformam os vetores de entrada em subconjuntos de **fibras paralelas** ativas. Os pesos ajustáveis das sinapses correspondentes a essas fibras ativas produzirão a saída. As células de **Purkinje** somarão os estímulos positivos diretamente e os negativos através das **células em cesto** e **células estreladas**. O ajuste das sinapses durante o treinamento é controlado pela entrada proveniente das **fibras trepadeiras**.



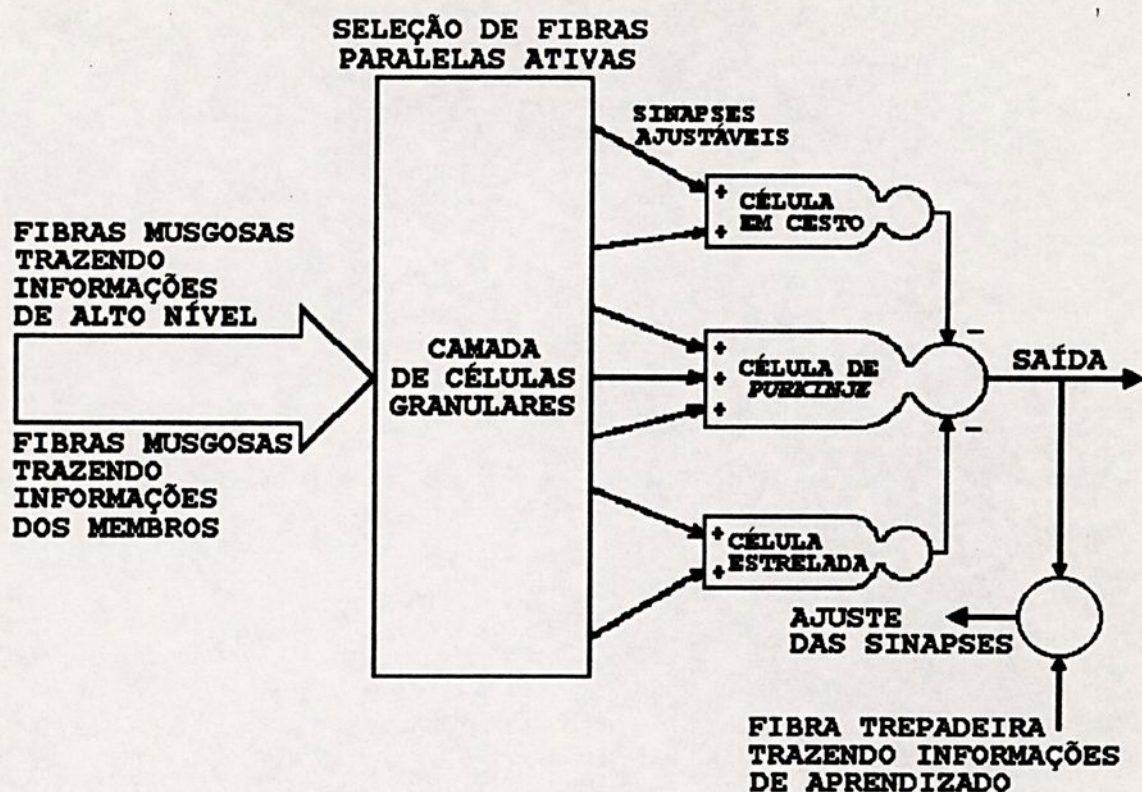


FIGURA 4.1: Modelo simplificado do cerebelo.

4.2. AS TRANSFORMAÇÕES PRESENTES NO CMAC

Cada vetor de entrada do **CMAC** é da forma $s = (s_1 \ s_2 \ \dots \ s_N)^T$ e pertence ao conjunto compacto $S \subset \mathbb{R}^N$. A saída é um escalar, porém p estruturas operando sobre o mesmo conjunto de entrada produzirão o vetor $\hat{f} \in \mathbb{R}^p$.

Portanto o **CMAC** implementará uma função

$$\hat{f} = h(s),$$

sendo, $s \in S \subset \mathbb{R}^N$ e $\hat{f} \in \mathbb{R}^P$. A função h pode ser decomposta em três transformações:

$$s \rightarrow m \rightarrow a \rightarrow \hat{f}$$

sendo: s = vetor de entrada,

m = variável vetorial intermediária,

a = vetor de células de associação e

$\hat{f}$ = vetor de saída.

4.2.1. Transformação $s \rightarrow m$

Cada vetor de entrada $s = (s_1 \ s_2 \ \dots \ s_N)^T$ é composto por N variáveis que podem ser contínuas ou discretas. A faixa de variação de cada s_i , $i \in \{1, 2, \dots, N\}$, é quantizada por C funções de quantização, a saber, $q_1, q_2, \dots, q_C$, em D intervalos, sendo que há sobreposição de uma C -ésima parte do intervalo de quantização entre duas funções adjacentes. A cada variável s_i está associado um vetor m_i com C elementos discretos que são os campos de recepção dados por $q_1, q_2, \dots, q_C$. A figura (4.2) mostra um exemplo para $N = 2$, $D = 4$, sendo que s_i é representado com resolução unitária no intervalo de 0 a 9. A faixa de cada variável é discretizada por $C = 3$ funções de quantização.



Para todo valor inteiro de s_i existe um único vetor m_i , tal que: $m_i = (q_1(s_i) \ q_2(s_i) \ \dots q_c(s_i))^T$. Na figura (4.2), por exemplo, para $s = (s_1 \ s_2)^T = (1 \ 7)^T$, teremos:

$$q_1(s_1) = q_1(1) = a_{(1)}$$

$$q_2(s_1) = q_2(1) = f_{(1)} \quad \rightarrow \quad m_1 = \{a_{(1)}, f_{(1)}, i_{(1)}\},$$

$$q_3(s_1) = q_3(1) = i_{(1)}$$

$$q_1(s_2) = q_1(7) = c_{(2)}$$

$$q_2(s_2) = q_2(7) = h_{(2)} \quad \rightarrow \quad m_2 = \{c_{(2)}, h_{(2)}, k_{(2)}\}.$$

$$q_3(s_2) = q_3(7) = k_{(2)}$$

Um resultado da transformação $s \rightarrow m$ é que a informação de um único vetor s é transmitida a NC canais de dados discretos pelas funções $q_1, q_2, \dots q_c$ aplicadas em $s_1, s_2, \dots s_N$. Pequenas variações em s podem produzir alterações em $m_1, m_2, \dots m_c$, mas não em todos os seus elementos. Com isso, a dois vetores próximos em S corresponderão canais coincidentes, mas não todos. Isto confere ao **CMAC** a capacidade de generalização que é muito importante no armazenamento e recuperação de informações quando duas situações apresentam semelhanças mas não são coincidentes.



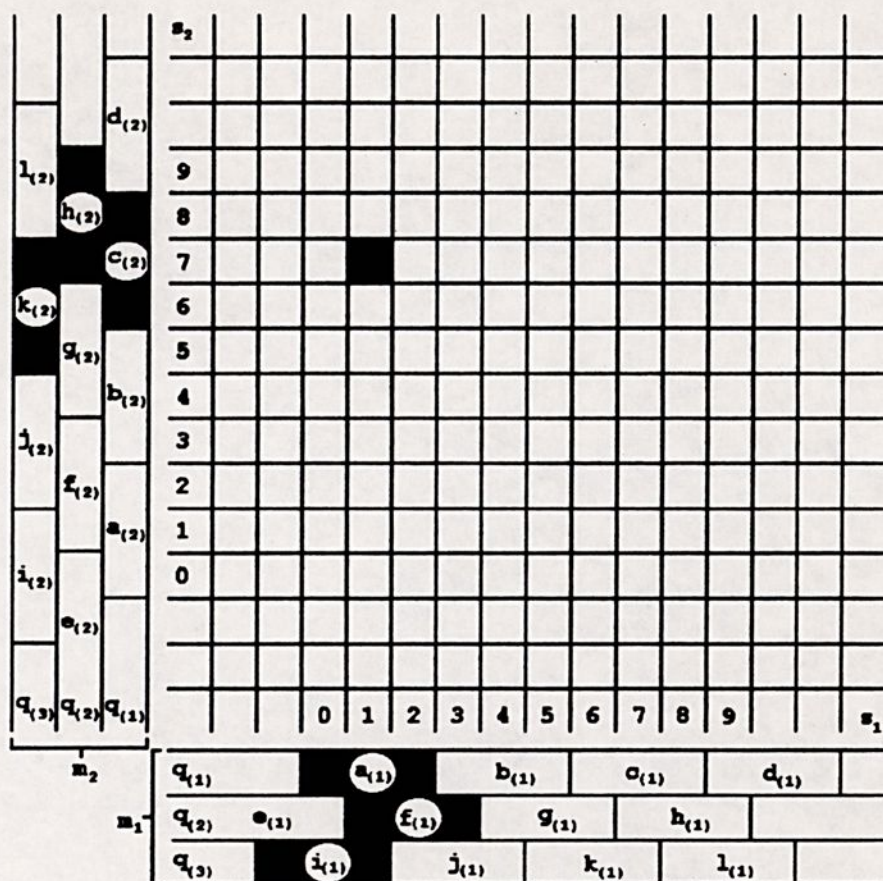


FIGURA 4.2: Neste exemplo, as funções de quantização do *QMAC* fornecem $N=2$ vetores com $C=3$ elementos discretos.

4.2.2. Transformação $m \rightarrow a$

No modelo proposto na figura (4.1), a seleção de **fibras paralelas** feita pelas células granulares é simulada no **CMAC** pela concatenação dos vetores discretos $\mathbf{m}_1$ a fim de se obter o vetor de células de associação. Tomemos o exemplo ilustrado na figura (4.2). Cada um dos $C = 3$ elementos do vetor discreto $\mathbf{m}_1$ pode assumir $D = 4$ valores diferentes como resultado de q_1 , q_2 e q_3 . O mesmo ocorre com o vetor $\mathbf{m}_2$. A formação de um par ordenado

devido a q_1 aplicada a s_1 e s_2 , onde o primeiro elemento pertence a $\{a_1, b_1, c_1, d_1\}$ e o segundo elemento pertence a $\{a_2, b_2, c_2, d_2\}$ oferece 16 possibilidades de arranjo. Então, a concatenação das primeiras posições de m_1 e de m_2 pode gerar 16 resultados diferentes, isto é, deve haver 16 células de associação e por conseguinte 16 pesos associados, sendo que somente uma é selecionada. Analogamente, teremos 16 pesos associados a q_2 e q_3 aplicados a s_1 e s_2 , o que resulta em um número total de pesos $Z=48$.

De forma geral, a concatenação de N vetores discretos m_i produz um vetor $a = (a_1 \ a_2 \dots a_z)^T$ de células de associação com C elementos selecionados aos quais se atribui valor 1; os demais elementos são zerados. Os elementos selecionados em a formam o conjunto a^* que chamaremos de a^* .

Dois vetores de entrada $s_{(1)}$ e $s_{(2)}$ muito próximos, a ponto de pertencerem aos mesmos intervalos de quantização, terão iguais conjuntos a^* , isto é, $a_{(1)}^* = a_{(2)}^*$ e assim, $h(s_{(1)}) = h(s_{(2)})$. Se $s_{(1)}$ e $s_{(2)}$ forem próximos, mas sem pertencerem aos mesmos intervalos de quantização, então $a_{(1)}^* \cap a_{(2)}^*$ terá menos de C elementos, e $h(s_{(1)})$ irá diferir de $h(s_{(2)})$ devido às células de associação ativas que não são comuns. Quando $s_{(1)}$ e $s_{(2)}$ forem distantes, $a_{(1)}^*$ e $a_{(2)}^*$ serão disjuntos, e $h(s_{(1)})$ será independente de $h(s_{(2)})$. A figura (4.3) mostra a sobreposição entre $a_{(1)}^*$ e $a_{(2)}^*$ para $s_{(1)}$ e $s_{(2)}$ na mesma vizinhança em S .



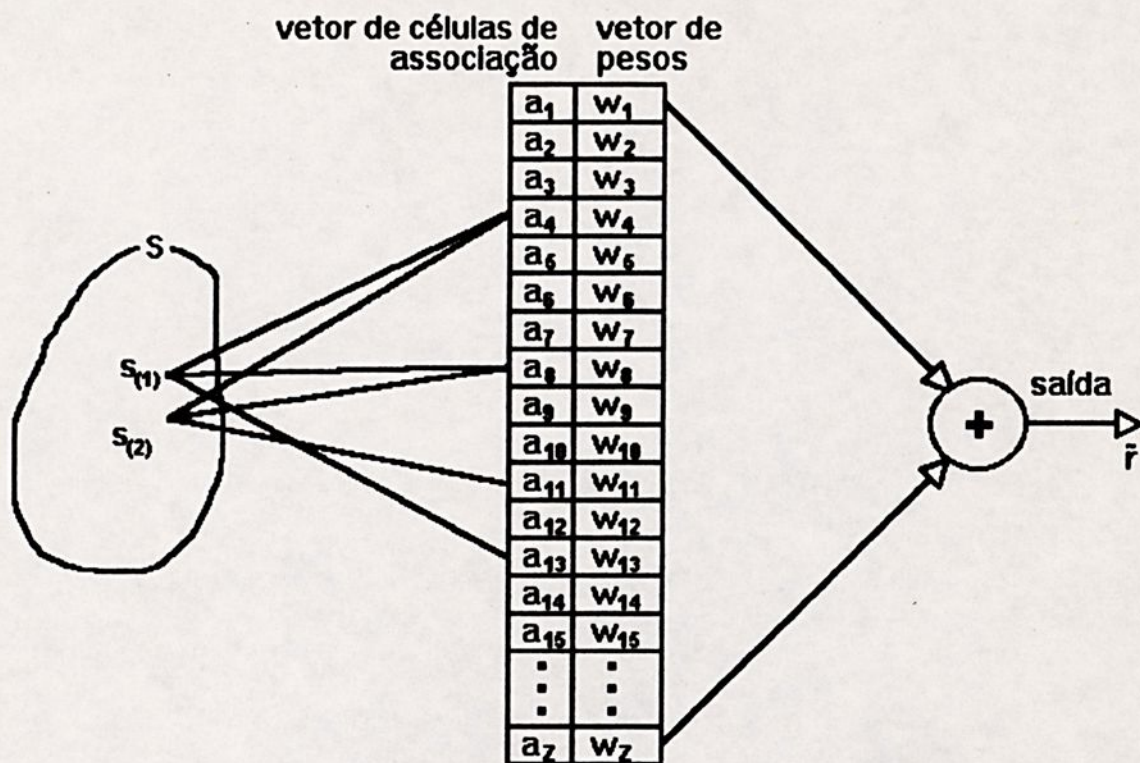


FIGURA 4.3: A capacidade de generalização do CMAC vem da sobreposição que ocorre para dois vetores de entrada próximos em S .

4.2.3. Transformação $a \rightarrow \hat{x}$

As sinapses entre as **células granulares** e a **célula de Purkinje** contribuem de forma a estimular a ação característica desta, enquanto que as sinapses entre as **células granulares** e as **células em cesto** e **estreladas** inibem a ação característica da célula de **Purkinje**. O **CMAC** gera a saída somando os elementos de um vetor $w = (w_1 \ w_2 \ \dots \ w_z)^T$, $w \in \mathbb{R}^z$, escolhidos pelas células não nulas do vetor a (veja figura 4.3). Então, o j -ésimo elemento do vetor de saída, $j \in \{1, 2, \dots, p\}$ é calculado através da equação (4.1):

$$\hat{x}_j = \sum_{i=1}^s a_i w_{ji} = a \cdot w_{(j)}. \quad (4.1)$$

Nesse ponto, vemos que a mesma transformação $s \rightarrow a$ pode ser usada para computar os p componentes do vetor $\hat{x} \in R^p$, cada um deles com uma aplicação $a \rightarrow \hat{x}_j$. Logo, há tantos vetores $w_{(j)}$ quantas forem as saídas do **CMAC**.

4.3. TREINAMENTO DO CMAC

Numa tentativa de imitar o mecanismo de adaptação das sinapses no cerebelo, usa-se o algoritmo dos mínimos quadrados para atualizar os pesos correspondentes a uma saída errônea.

Suponhamos que o **CMAC** deva reproduzir a função $r = f(s)$, $s \in S$ compacto e $S \in R^N$ e $r \in R$. Então, dado um vetor $s_{(i)}$, o **CMAC** deve fornecer $\hat{x}_{(i)} = h(s_{(i)})$ tão próximo de $r_{(i)} = f(s_{(i)})$ quanto possível. Logo, para cada par $r_{(i)}$ e $\hat{x}_{(i)}$, se $|r_{(i)} - \hat{x}_{(i)}| \leq \xi$, onde ξ é a precisão requerida, não é necessário modificar nenhum peso pois a informação já está armazenada. Porém, se $|r_{(i)} - \hat{x}_{(i)}| > \xi$, deve-se somar a cada peso que contribuiu para $\hat{x}_{(i)}$ a quantidade

$$\frac{1}{c} (r_{(i)} - \hat{x}_{(i)}).$$

Vetorialmente, esta lei de treinamento pode ser escrita como,

$$\mathbf{w}^{[k]} = \mathbf{w}^{[k-1]} + \frac{1}{C} (\mathbf{x}_{(1)} - \mathbf{z}_{(1)} \cdot \mathbf{w}^{[k-1]}) \mathbf{z}_{(1)}.$$

Devido à sua capacidade de generalização, torna-se desnecessário treinar o **CMAC** em dois pontos muito próximos em $\mathbf{S}$, o que nos permite tomar um menor conjunto de pontos de treinamento para assimilar o comportamento da função $\mathbf{r} = \mathbf{f}(\mathbf{s})$. Obviamente, o processo de adaptação deve ser repetido várias vezes para ajustar os pesos eventualmente compartilhados pelos pontos de treinamento. A habilidade do **CMAC** produzir $\hat{\mathbf{z}}_{(1)} \neq \hat{\mathbf{z}}_{(2)}$ para $\mathbf{s}_{(1)} \neq \mathbf{s}_{(2)}$ depende de quantos pesos são compartilhados pelas duas saídas e de quanto os dois conjuntos de pesos não comuns diferem entre si. Até 1989 não havia sido publicada uma comprovação formal da convergência deste processo de treinamento. **Parks & Militzer**[13] mostraram que o algoritmo pode convergir para um único ponto, para um ciclo limite de pontos ou para uma mínima zona de captura. A demonstração é baseada numa interpretação geométrica do algoritmo proposto por **Albus** e o resultado final depende de condições que discutiremos a seguir.

Neste algoritmo de treinamento, cada ponto $\mathbf{w}^{[k]} \in \mathbb{R}^Z$ gerado pertence ao hiperplano de dimensão $(Z-1)$



$$\hat{\mathbf{a}}_{(1)}^T \mathbf{w} = \sqrt{C} \mathbf{x}_{(1)},$$

$$\hat{\mathbf{a}}_{(1)} = \frac{1}{\sqrt{C}} \mathbf{a}_{(1)}, \quad |\hat{\mathbf{a}}_{(1)}| = 1,$$

e, ainda, este hiperplano é ortogonal ao vetor $(\mathbf{w}^{[k]} - \mathbf{w}^{[k-1]}) \in \mathbb{R}^2$.

Parks & Militzer adotaram a seguinte equação para computar a saída do CMAC:

$$\hat{x}_{(1)} = \frac{1}{C} \mathbf{a}_{(1)} \cdot \mathbf{w} = \frac{1}{\sqrt{C}} \hat{\mathbf{a}}_{(1)} \cdot \mathbf{w}.$$

E a regra de treinamento usada é:

$$\mathbf{w}^{[k]} = \mathbf{w}^{[k-1]} + (\sqrt{C} \mathbf{x}_{(1)} - \hat{\mathbf{a}}_{(1)} \cdot \mathbf{w}^{[k-1]}) \hat{\mathbf{a}}_{(1)}.$$

Matematicamente, este algoritmo é equivalente ao adotado por **Albus**, pois, durante o treinamento, nesta última versão, os pesos devem assumir um valor C vezes menor do que na primeira.

O objetivo do processo de treinamento é que, para o conjunto de pontos de treinamento $\{\mathbf{s}_{(1)}, \mathbf{s}_{(2)}, \dots, \mathbf{s}_{(NP)}\}$, tenhamos $\hat{x}_{(1)} = x_{(1)}$, o que é equivalente a

$$\hat{\mathbf{a}}_{(1)}^T \mathbf{w} = \sqrt{C} \mathbf{x}_{(1)} \quad (4.2)$$



A equação (4.2) pode ser considerada uma das linhas da seguinte equação matricial:

$$AW = R \quad (4.3)$$

A é uma matriz $NP \times Z$, suas linhas possuem C elementos iguais a $\frac{1}{\sqrt{C}}$ e os demais são nulos. R é um vetor com NP linhas, seu i -ésimo elemento é dado por $\sqrt{C}r_{(i)}$.

Pode-se escolher $w^{[0]} = 0$ como ponto inicial do processo iterativo, mas isso não é uma regra; outra escolha é atribuir valores aleatórios aos elementos de $w^{[0]}$. Antes de iniciar o processo de aprendizado devemos decidir se os pontos de treinamento serão tomados de forma cíclica, isto é, um a um segundo uma ordem particular e repetidas vezes, ou aleatoriamente dentro do conjunto $\{s_{(1)}, s_{(2)}, \dots, s_{(NP)}\}$. A tabela (4.1) reúne algumas conclusões baseadas nos critérios que acabamos de discutir.

Parks & Militzer também apresentaram uma comparação entre cinco algoritmos de treinamento, entre eles o criado por **Albus**.

1) algoritmo proposto por Albus ("AL")

Este algoritmo foi detalhadamente discutido na seção anterior.



Tabela 4.1: Várias possibilidades para a solução da equação matricial (4.2) e os resultados do treinamento pelo algoritmo proposto por Albus.

caso	relação entre Z e NP	posto (Q) da matriz A	posto (Q') da matriz aumentada $[A, R]$	interseção dos hiperplanos definidos pela equação (4.1)	tipo de convergência do algoritmo
1	$NP < Z$	$Q = NP$	$Q' = Q$	interseção é um hiperplano $(Z - NP)$ -dimensional	converge para um ponto que é único se w'' é gerado pelos vetores $a_{(j)}$, caso contrário o ponto de convergência depende de w''
2	$NP = Z$	$Q = NP = Z$	$Q' = Q$	interseção é um ponto no espaço Z -dimensional	converge para um ponto que é único para qualquer w''
3	$NP > Z$	$Q = Z$	$Q' = Q$	equações redundantes e a interseção é um ponto como no caso 2	converge para um ponto que é único para qualquer w'' , como no caso 2
4	$NP > Z$	$Q = Z$	$Q' = Q + 1$	e q u a q ã o s inconsistentes e não existe um único hiperplano que seja a interseção	quando o treinamento é cíclico, converge para um ciclo limite de pontos que depende da sequência escolhida para o treinamento mas é independente de w'' ; quando o treinamento é aleatório, converge eventualmente para uma mínima zona de captura
5	-----	$Q = \min(NP, Z)$	$Q' = Q$	equações redundantes e a interseção é um hiperplano de dimensão $Z - Q$	como no caso 1
6	-----	$Q = \min(NP, Z)$	$Q' = Q + 1$	e q u a q ã o s inconsistentes e não existe um único hiperplano seja a interseção	quando o treinamento é cíclico, converge para um ciclo limite de pontos que depende da sequência escolhida para o treinamento mas é independente de w'' , e sua localização independe de w'' se os vetores $a_{(j)}$ gerarem w'' ; quando o treinamento é aleatório, converge, eventualmente, para uma mínima zona de captura

2) treinamento por média móvel ("AV")

Define-se um vetor contador $C = (C_1 \ C_2 \dots C_Z)^T$ que, no início de cada passo de treinamento, é atualizado por:

$$C^{[k]} = C^{[k-1]} + \sqrt{C} \Delta_{(j)}^{[k-1]},$$

$$C^{[0]} = 0,$$

sendo que $\sqrt{C} \Delta_{(j)}^{[k]}$ possui elementos unitários nas posições



definidas pelas células ativas e zeros nas demais posições.

Calcula-se um vetor

$$\mathbf{b}^{[k]} = (b_1^{[k]} \ b_2^{[k]} \ \dots \ b_s^{[k]})^T,$$

$$b_i^{[k]} = \frac{\hat{a}_i^{[k]}(j)}{c_i^{[k]}},$$

$\hat{a}_i^{[k]}(j)$ é o i-ésimo elemento de $\hat{\mathbf{a}}^{[k]}(j)$, $c_i^{[k]}$ é o i-ésimo elemento de $\mathbf{c}^{[k]}$.

Finalmente, modifica-se o vetor de pesos por

$$\mathbf{w}^{[k]} = \mathbf{w}^{[k-1]} + \sqrt{C} (x_{(j)} - \hat{x}_{(j)}^{[k-1]}) \mathbf{b}^{[k]}.$$

Se o conjunto de saídas desejadas $\mathbf{r}_{(j)}$ for afetado por ruído com média zero, como no caso de dados amostrados em situações práticas, este algoritmo filtrará o ruído[13].

3) otimização parcial do comprimento do passo na última direção ortogonal ("OS")

Este algoritmo corrige o vetor $\mathbf{w}^{[k-1]}$ na direção dada por $\hat{\mathbf{a}}^{[k]}(j)$ da mesma forma que o algoritmo (AL). Porém, o ponto $\mathbf{w}^{[k]}$ não atinge o plano



$$\hat{\mathbf{a}}_{(j)}^T \mathbf{w} = \sqrt{C} \mathbf{x}_{(j)}$$

O objetivo é minimizar a soma

$$I_\lambda = \sum_{i=k-\lambda+1}^k (\hat{\mathbf{a}}_{(j_i)}^T \mathbf{w}^{[k]} - \sqrt{C} \mathbf{x}_{(j_i)})^2$$

$\hat{\mathbf{a}}_{(j_i)}$ é o vetor unitário de células de associação definido pelo ponto de treinamento $\mathbf{s}_{(j)}$ tomado na iteração i e $\mathbf{x}_{(j_i)}$ sua respectiva saída desejada, isto é,

$$\mathbf{x}_{(j_i)} = \mathbf{f}(\mathbf{s}_{(j_i)}),$$

λ é um inteiro tal que $1 \leq \lambda \leq k$.

Logo, I_λ é a soma dos quadrados das distâncias entre o ponto $\mathbf{w}^{[k]}$ e os hiperplanos

$$\hat{\mathbf{a}}_{(j_i)}^T \mathbf{w} = \sqrt{C} \mathbf{x}_{(j_i)}$$

escolhidos nos λ últimos passos do treinamento.

Com isso, a lei de treinamento fica

$$\mathbf{w}^{[k]} = \mathbf{w}^{[k-1]} + \frac{\sum_{i=k-\lambda+1}^k (\sqrt{C} \mathbf{x}_{(j_i)} - \hat{\mathbf{a}}_{(j_i)}^T \mathbf{w}^{[k-1]}) \hat{\mathbf{a}}_{(j_i)} \hat{\mathbf{a}}_{(j_i)}^T}{\sum_{i=k-\lambda+1}^k (\hat{\mathbf{a}}_{(j_i)}^T \hat{\mathbf{a}}_{(j_i)})^2} \hat{\mathbf{a}}_{(j_i)}$$

4) treinamento no ponto com máximo erro ("ME")

Este algoritmo treina o **CMAC** no ponto que apresenta maior erro (em módulo), entre os λ últimos testados até a iteração k . O parâmetro λ é escolhido de forma que $1 \leq \lambda \leq k$.

Na i -ésima iteração, foi tomado o ponto $s_{(j_i)}$ do conjunto $\{s_{(1)}, s_{(2)}, \dots, s_{(m)}\}$ e o erro associado a esse ponto na k -ésima iteração é dado por

$$e_{(j_i)}^{[k]} = x_{(j_i)} - \hat{x}_{(j_i)}^{[k]} = x_{(j_i)} - \frac{1}{C} a_{(j_i)}^T v^{[k]}$$

Se $i \in \{k-\lambda+1, k-\lambda+2, \dots, k\}$ guardaremos os λ últimos erros anteriores à k -ésima iteração. Devemos selecionar o erro $e_{(j_i)}^{[k]}$ tal que

$$|e_{(j_i)}^{[k]}| \geq |e_{(j_1)}^{[k]}| \quad (4.3)$$

para $i \in \{k-\lambda+1, k-\lambda+2, \dots, k\}$. Se mais de um $e_{(j_i)}^{[k]}$ satisfizer a inequação (4.3), escolhe-se o erro definido pelo maior valor i .

5) ortonormalização parcial pelo processo de Gram-Schmidt ("GS")

O objetivo é armazenar a informação de um novo ponto de treinamento sem afetar o conteúdo já assimilado com os λ pontos anteriores. Para que isso aconteça, devemos modificar



$w^{(k-1)}$ na direção $\hat{d}_{(j)}$, que seja ortogonal a $\hat{a}_{(j)}$, $i \in \{k-\lambda+1, k-\lambda+2, \dots, k\}$ que são as células de associação definidas pelos λ últimos pontos onde o **CMAC** foi treinado até a iteração $(k-1)$.

O vetor $\hat{d}_{(j)}$ é obtido por meio dos vetores $d_{1(j)}, d_{2(j)}, \dots, d_{\lambda(j)}$ definidos por:

$$\hat{d}_{1(j_k)} = \hat{a}_{(j_{k-\lambda+1})}$$

$$d_{i(j_k)} = \hat{a}_{(j_{k-\lambda+i})} - \sum_{p=1}^{i-1} (\hat{a}_{(j_{k-\lambda+i})} \hat{d}_{p(j_k)}) \hat{d}_{p(j_k)}, \quad (2 \leq i \leq \lambda),$$

$$\hat{d}_{i(j_k)} = \frac{d_{i(j_k)}}{|d_{i(j_k)}|}, \quad |d_{i(j_k)}| \neq 0$$

para $i \in \{2, 3, \dots, \lambda\}$ sendo $\hat{d}_{1(j_k)}$ ortogonal aos vetores $\hat{a}_{(j_{k-\lambda+\mu})}$, $\mu \in \{1, 2, \dots, i-1\}$. Caso $d_{i(j_k)}$ seja zero, seu versor será nulo. A direção final que será utilizada para atualizar efetivamente os pesos é dada por

$$\hat{d}_{(j_k)} = \begin{cases} \hat{d}_{1(j_k)}, & |d_{1(j_k)}| \neq 0 \\ \hat{a}_{(j_k)}, & \text{caso contrário} \end{cases}$$

$|\hat{d}_{\lambda(j_k)}| = 0$ indica que a ortonormalização de **Gram-Schmidt** falhou, pois $\hat{a}_{(j_k)}$ é uma combinação linear de $\hat{a}_{(j_p)}$, $p \in \{k-\lambda+1, k-\lambda+2, \dots, k-1\}$.

A lei de treinamento é dada por

$$w^{[k]} = w^{[k-1]} + \frac{\sqrt{C} x_{(j_k)} - \hat{a}_{(j_k)}^T w^{[k-1]}}{\hat{a}_{(j_k)}^T d_{(j_k)}} \hat{d}_{(j_k)}.$$

Dentre os cinco algoritmos descritos, **Parks & Militzer** recomendam o algoritmo **ME** pois, apresenta boa velocidade de convergência e requer um moderado esforço computacional.

4.4. REQUISITOS DE MEMÓRIA

O **CMAC** apresenta a desvantagem de requerer enorme quantidade de memória nos casos com muitas entradas e saídas. O tamanho da memória requerida é dado pelo número de saídas, p , e pelo número de células de associação, Z (onde Z depende do número D , de intervalos por função q_i para cada variável de entrada e do parâmetro de generalização, C). A memória física associada ao exemplo da figura 4.2 deve armazenar um vetor w com $p \times Z = 48$ elementos. Aumentando uma entrada e uma saída, mantendo $C=3$ e $D=4$ temos $p \times Z = 384$ pesos.

Uma forma de se reduzir a memória física do **CMAC** emprega o código **HASH**. Este procedimento baseia-se em computar os endereços dos elementos de um pequeno vetor de pesos através de um gerador de números pseudoaleatórios que usa o endereço de uma célula de associação como argumento.



Embora o código **HASH** resolva o problema do tamanho da memória física, podem acontecer colisões[5][6][7], isto é, o mesmo peso ser endereçado a duas células de associação em a^* : tal peso será somado duas vezes e haverá diminuição da precisão da representação. Mas se para $s^{(1)}$ e $s^{(2)}$ afastados, algum peso é solicitado por $a^{*(1)}$ e $a^{*(2)}$, ocorre uma generalização indesejada o que interfere na convergência do treinamento[5][6]. Esses dois efeitos indesejáveis podem ser minimizados se aumentarmos a quantidade de intervalos da partição e a quantidade de pesos pois, aí, a sobreposição entre $a^{*(1)}$ e $a^{*(2)}$ será menor.

4.5. CMAC ALTERNATIVO

Kraft & Campagna[8] propuseram uma estrutura modificada para o **CMAC** usada na solução de problemas em que não é necessária uma grande memória. Considere a função $r=f(s)$, $s \in S \subset \mathbb{R}^2$, S compacto, $r \in \mathbb{R}$. Divide-se S em Z intervalos conforme a figura (4.4).

A cada célula associa-se um elemento numérico (peso). O vetor s pertence a um único intervalo a_i , $i \in \{1, 2, \dots, Z\}$. Para um vetor de entrada s , a estrutura fornece $\hat{x}=h(s)$ obtido pela média aritmética dos pesos contidos numa região centrada na célula a_i , que denominaremos Região Ativa (**RA**). A Região Ativa é



determinada pelo vetor de entrada, s , e por um número natural c que representa a quantidade de intervalos tomados no sentido positivo ou negativo do eixo de cada variável de entrada.

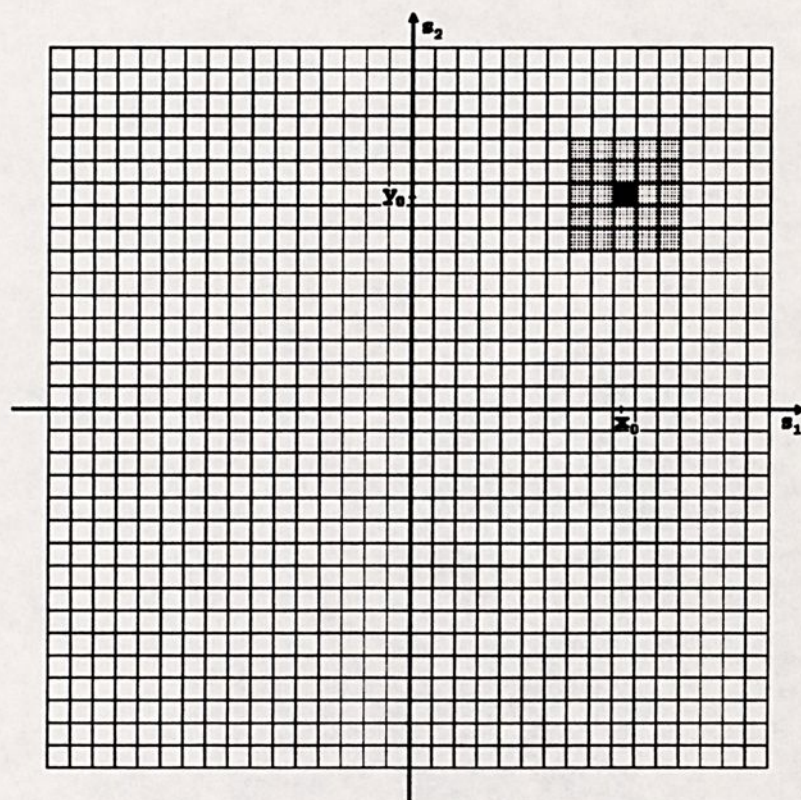


FIGURA 4.4: Conjunto S e a Região Ativa para $s=(x_0, y_0)$.

O conceito de Região Ativa no **CMAC** de **Kraft & Campagna** é similar ao conceito de conjunto de células de associação selecionadas, a^* , no **CMAC** de **Albus**. Então, por analogia, representaremos a quantidade de células da **RA** por C . E, ainda,

$$C = (2c+1)^N$$

sendo N a quantidade de elementos do vetor de entrada. Na figura

4.4, $c=2$, $N=2$ e a RA possui 25 células.

Dessa forma, se tomarmos $s^{(1)}$ e $s^{(2)}$ próximos, as respectivas RA podem ter até C células em comum e, aí, $\hat{r}^{(1)} = h(s^{(1)})$ é idêntico a $\hat{r}^{(2)} = h(s^{(2)})$. À medida que $s^{(1)}$ se afasta de $s^{(2)}$ as RA têm cada vez menos células em comum e $\hat{r}^{(1)}$ difere de $\hat{r}^{(2)}$. A taxa de generalização depende, como no CMAC original, do parâmetro C .

A saída dessa configuração é dada por

$$\hat{r} = \frac{1}{C} \sum w$$

sendo que w é um peso associado a qualquer célula pertencente à RA.

O treinamento consiste em modificar os pesos de forma que $\hat{r}^{(j)}$ se aproxime de $r^{(j)}$, quando $s^{(j)}$ é tomado em um conjunto finito de pontos. A regra de treinamento é baseada no algoritmo dos mínimos quadrados e consiste em

$$w^{[k+1]} = w^{[k]} + \frac{\beta}{C} (r - \hat{r}^{[k]}),$$

sendo $w^{[k]}$ um peso da RA, onde β é a taxa de aprendizado, e geralmente adota-se $0 < \beta \leq 1$. Este procedimento deve ser repetido para todos os pontos de treinamento.



Esta estrutura pode ser implementada facilmente quando não se requer muitos intervalos na partição feita em S . Sua maior desvantagem é a necessidade de grande quantidade de memória, pois não se usa nenhuma técnica para redução do número de pesos.

Observando a complexidade do problema a ser abordado e assim, a quantidade de intervalos necessários, esta estrutura pode ser generalizada para mais de duas variáveis de entrada. Se for necessário reproduzir p saídas, podemos utilizar p estruturas descritas acima em paralelo, para reproduzir as p saídas.



Capítulo 5

SISTEMAS DE CONTROLE AUTOMÁTICO COM O CMAC

5.1. INTRODUÇÃO

Em geral o controle de uma planta deve ser imune a perturbações. Com isso, torna-se necessário usar realimentação do erro para garantir uma maior robustez. Portanto, as variáveis de estado da planta, ou o erro calculado a partir dessas variáveis são entradas de um controlador baseado no **CMAC**.

Além disso, por exemplo, a trajetória que as variáveis de estado devem percorrer durante um intervalo de tempo e a forma como isso acontece, devem ser fornecidas através de informações de um nível mais alto.

As variáveis de entrada do **CMAC** pertencem, então, a duas classes distintas: variáveis que trazem informações de baixo nível, definindo o estado da planta e aquelas de nível mais alto, referentes à tarefa a ser realizada.

A seguir descreveremos duas aplicações do **CMAC** no controle de sistemas dinâmicos desconhecidos[8][9].



5.2. O CMAC como controlador por retroação

Kraft & Campagna[8] utilizaram o CMAC para controlar um sistema de primeira ordem dado pela equação (5.1):

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{a}\mathbf{x}_k + \mathbf{b}\mathbf{r}_k + \mathbf{f}(\mathbf{x}_k), \\ \mathbf{a}, \mathbf{b}, \mathbf{x} &\in \mathbb{R} \end{aligned} \quad (5.1)$$

segundo a estrutura dada na figura 5.1. A função $\mathbf{f}(\mathbf{x}_k)$ podia assumir duas formas distintas, permitindo, assim, associar dois casos a um mesmo problema de controle:

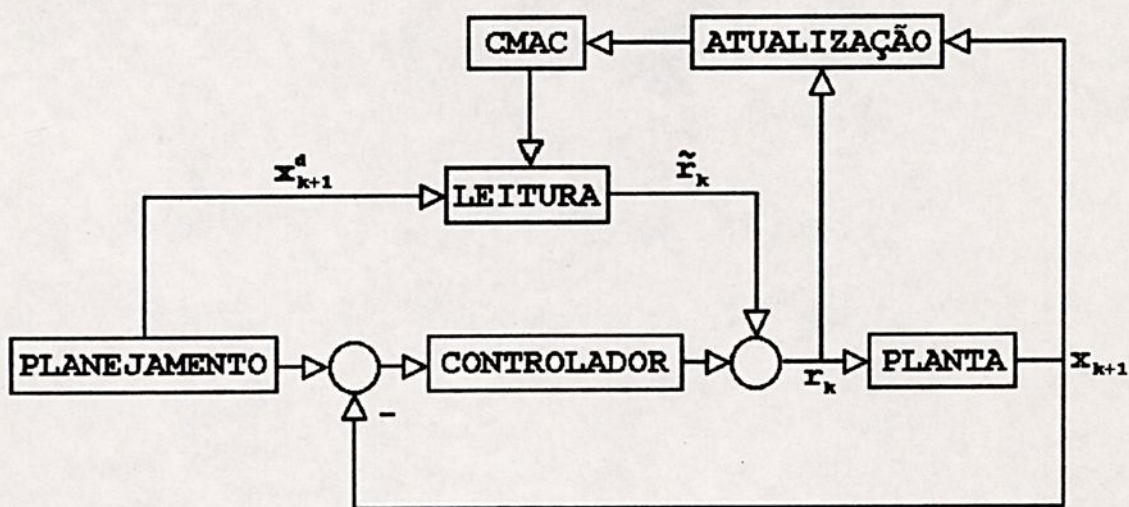


FIGURA 5.1: Controle de um sistema dinâmico de primeira ordem com CMAC.

→ $\mathbf{f}(\mathbf{x}_k) = -0,4(\mathbf{x}_k)^3$, ou então,

- $f(x_k)$ consistia de um ruído branco com média nula e uniformemente distribuído entre $-0,4$ e $0,4$.

Durante cada ciclo de controle, a saída x_k era medida e o valor desejado x_{k+1}^d era calculado ou recuperado de uma memória auxiliar. A saída $\hat{x}_k$ era obtida a partir da leitura da Região Ativa determinada por x_k e por x_{k+1}^d . O valor de $\hat{x}_k$ obtido era somado à saída de um controlador tipo ganho de erro de retroação para formar a excitação aplicada à planta. A Região Ativa do **CMAC**, definida por x_k e por x_{k+1} era, então, atualizada com o valor de x_k responsável pela transição $x_k \rightarrow x_{k+1}$ ($\beta < 1$).

Em consequência desse processo de treinamento, o **CMAC** armazenava o valor de $\hat{x}_k$ necessário para que x_k assumisse um valor x_{k+1}^d especificado para o ciclo de controle seguinte. Após alguns ciclos de controle, o aprendizado estava concluído, e apenas o **CMAC** fornecia a ação de controle aplicada à planta.

5.3. O **CMAC** treinado por regras

Handelman, Lane & Gelfand[9] estudaram o treinamento do **CMAC** supervisionado por um monitor baseado em regras. A tarefa consistia em controlar o sistema esquematizado na figura 5.2(a) de forma que as variáveis de estado atingissem um ponto previamente escolhido admitindo-se uma certa tolerância.



Na figura 5.2(b), e_i é o erro entre a inclinação final do bastão externo e a inclinação desejada e e_p é o erro referente ao posicionamento do seu centro sobre o eixo horizontal ao final da simulação.

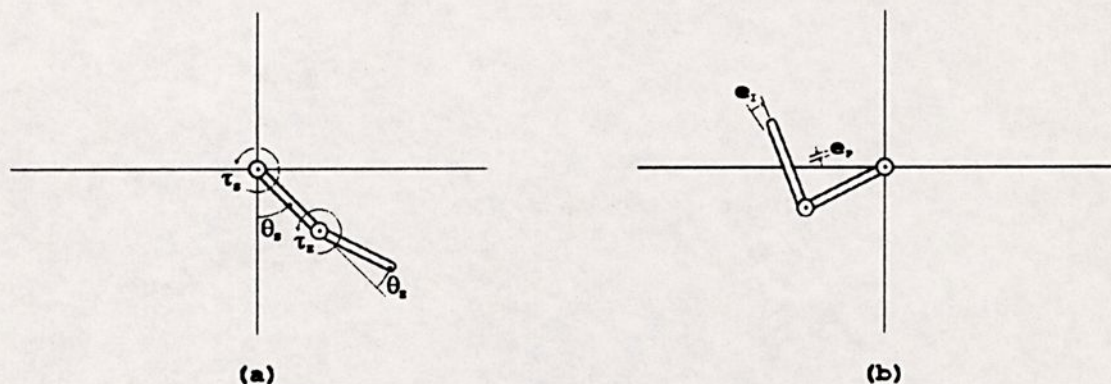


FIGURA 5.2: Sistema físico a ser controlado pelo CMAC ao final do treinamento. (a) sistema na posição inicial; (b) posição do sistema ao final da simulação.

Os torques τ_E e τ_S eram calculados através das equações (5.2):

$$\begin{aligned}\tau_E(k) &= K_{E_1}(k) \tau_0 - K_{E_2}(k) [c_1(\theta_E(k) - \theta_{E_0}(k)) + c_2 \omega_E(k)], \\ \omega_E(k) &= \frac{\partial \theta_E(k)}{\partial t}, \\ \tau_S(k) &= K_{S_1}(k) \tau_0 - K_{S_2}(k) [c_1(\theta_S(k) - \theta_{S_0}(k)) + c_2 \omega_S(k)], \\ \omega_S(k) &= \frac{\partial \theta_S(k)}{\partial t},\end{aligned}\tag{5.2}$$

sendo, $K_{E_1}(k)$, $K_{S_1}(k)$: ganhos associados ao torque base, τ_0 ;

$K_{E_2}(k)$, $K_{S_2}(k)$: ganhos associados às posições/velocidades das

articulações;

$\theta_{x_1}(k)$, $\theta_{x_2}(k)$: posições de referência dadas pela trajetória desejada;

c_1 , c_2 : fatores para correção de escala para as grandezas ângulo e velocidade.

Inicialmente, $K_{x_1}(k)$, $K_{x_2}(k)$, $K_{x_3}(k)$, $K_{x_4}(k)$, $\theta_{x_1}(k)$ e $\theta_{x_2}(k)$ eram parâmetros gerados pelo controlador baseado em regras e serviam como saídas desejadas durante o processo de treinamento do **CMAC**. Da mesma forma, as variáveis de estado da planta eram as entradas aplicadas ao controlador baseado em regras e, posteriormente, ao **CMAC**. A figura 5.3 ilustra este aspecto da arquitetura de controle.

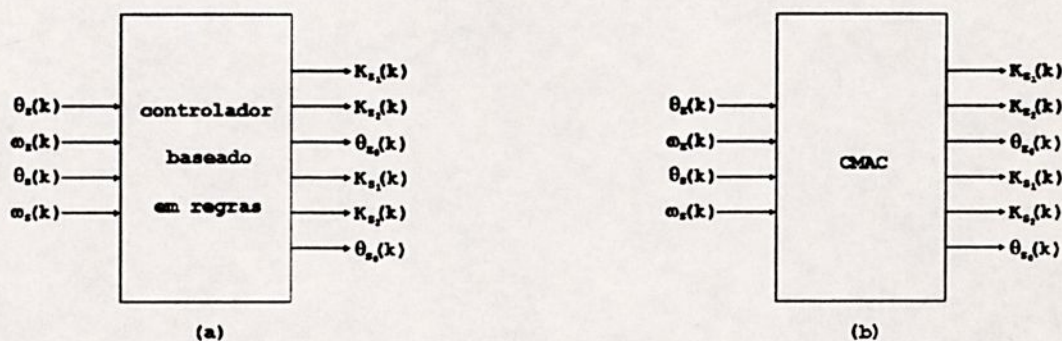


FIGURA 5.3: Os controladores geravam seis parâmetros como função do estado da planta. (a) inicialmente, durante o treinamento do **CMAC**, um controlador baseado em regras fornecia os ganhos e os ângulos de referência usados na equação 5.2; (b) quando as saídas do **CMAC** estivessem suficientemente próximas das saídas do controlador baseado em regras, o **CMAC** passava a fornecer os ganhos e os ângulos de referência para a equação 5.2.

Assim, o procedimento foi dividido em três fases:

- fase de processamento declarativo: as entradas do sistema



eram ponderadas por parâmetros gerados exclusivamente por um controlador baseado em regras, enquanto o **CMAC** era treinado de forma a reproduzir os mesmos parâmetros sob a supervisão do monitor baseado em regras;

- **fase de processamento misto:** as entradas do sistema eram ponderadas por parâmetros computados em conjunto pelo controlador baseado em regras e pelo **CMAC**, que ainda era submetido ao processo de treinamento;

- **fase de processamento reflexivo:** as entradas do sistema eram ponderadas exclusivamente por parâmetros obtidos nas saídas do **CMAC**, já treinado.



Capítulo 6

O CMAC COM PESOS DISCRETOS

De forma geral, o controle digital de sistemas envolve a conversão analógico-digital e digital-analógica de dados. Com isso, as variáveis amostradas e as ações de controle ficam restritas a conjuntos finitos de valores discretos.

Neste capítulo, propomos uma metodologia para reduzir a memória física do **CMAC**, baseada no caráter discreto de suas entradas e saídas.

Tolle & Ersü[10] abordaram a representação numérica dos pesos do **CMAC** em ponto fixo com a finalidade de redução de memória e de aumento da velocidade de processamento, em detrimento da precisão. Porém, cada peso era armazenado utilizando dois bytes (16 bits) e a memória física era reduzida principalmente devido ao uso do código **HASH**.

Nossa proposta é restringir os pesos e a saída do **CMAC** a conjuntos arbitrários de valores discretos, sendo que para as representações numéricas dos pesos e dos valores de saída pode ser usado um número de bits previamente especificado e dentro da



faixa 1 a 16 bits. Não é necessário discretizar as variáveis de entrada da mesma forma pois, a escolha dos campos de recepção já é feita por funções de saída discretas. Consideraremos que os dados amostrados são obtidos através de um conversor analógico digital tal como se mostra na figura (6.1).

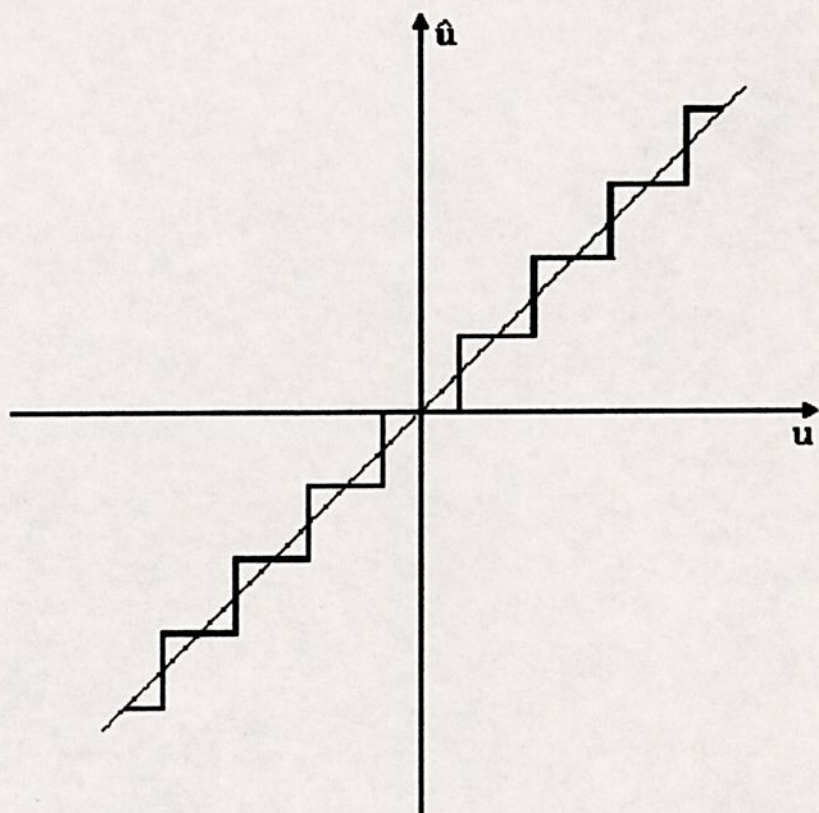


FIGURA 6.1: Tipo do conversor A/D empregado na discretização de dados amostrados.

Suponhamos que algum peso $w \in [w_{-m}, w_m]$ seja aproximado para um dos $(2m+1)$ valores possíveis $\hat{w}$, sendo

$$\hat{w} \in \{w_{-m}, w_{-m+1}, \dots, w_0, \dots, w_{m-1}, w_m\},$$

$$\Delta w = \frac{w_m - w_{-m}}{2m}, \quad (6.1)$$

$$w_v = w_0 + v\Delta w, \quad v \in \{-m, -m+1, \dots, 0, \dots, m-1, m\}.$$

Com isso, cada peso $w \in [w_{-m}, w_m]$ está associado a um número inteiro v correspondente ao $\hat{w}$ mais próximo de w . Assim, a memória requerida para o armazenamento de informações no **CMAC** com pesos discretos é reduzida devido ao fato de que armazenar um número inteiro v usando uma representação em ponto fixo com 16 bits, ou menos, requer uma memória menor do que um número representado em ponto flutuante que necessita de, pelo menos, 32 bits.

Escolhendo b bits para representar os $(2m+1)$ valores de v definidos na equação 6.1, deve-se obedecer a relação dada na equação 6.2 para a especificação de m :

$$2m+1 < 2^b \quad (6.2)$$

e é conveniente definirmos

$$2m+1 = 2^b - 1 \text{ ou } m = 2^{b-1} - 1. \quad (6.3)$$

De maneira análoga, suponha que a saída desejada para o **CMAC**, $r \in [r_{-n}, r_n]$, seja aproximada para um dos $(2n+1)$ valores possíveis $\hat{r}$ tal que,

$$t \in \{x_{-n}, x_{-n+1}, \dots, x_0, \dots, x_{n-1}, x_n\},$$

$$\Delta x = \frac{x_n - x_{-n}}{2n}, \quad (6.4)$$

$$x_i = x_0 + i\Delta x, \quad i \in \{-n, -n+1, \dots, 0, \dots, n-1, n\}.$$

Assim, cada valor $x \in [x_{-n}, x_n]$ está associado a um número inteiro i correspondente ao t mais próximo de x .

O próximo passo é escrever a saída do **CMAC** com pesos discretos, assim como um algoritmo de treinamento, a partir dos inteiros v , que definem os pesos, e i que definem as saídas, pois nosso objetivo é executar operações com números em ponto fixo.

Fazendo,

$$v_n - v_{-n} = K(x_n - x_{-n}), \quad K \geq 1$$

garantimos que os pesos tenham maior liberdade durante o processo de treinamento. Além disso, definiremos $w_0 = r_0 = 0$.

Com essas definições colocadas,

$$K\Delta x = K \frac{x_n - x_{-n}}{2n}$$

$$Kn\Delta x = K \frac{x_n - x_{-n}}{2} = \frac{v_n - v_{-n}}{2} = \Delta v.$$



Logo,

$$Kn\Delta r = m\Delta w. \quad (6.5)$$

Dado $s \in S$, uma entrada do **CMAC** e C a taxa de generalização, a saída correspondente $\hat{f}$ será dada pela soma dos pesos da Região Ativa, se este algoritmo for aplicado ao **CMAC** de **Kraft & Campagna**. Se o algoritmo baseado em pesos discretos for aplicado ao **CMAC** de **Albus**, sua saída pode ser escrita como a soma dos pesos correspondentes às posições não nulas de um vetor de células de associação.

Portanto, a saída do **CMAC** com pesos discretos, para a versão de **Kraft & Campagna** ou para a versão de **Albus**, pode ser escrita conforme a equação 6.6:

$$\hat{f} = \sum_{k=1}^C w_{v(k)}, \quad (6.6)$$

$$\hat{f} = \sum_{k=1}^C (w_0 + v_{(k)} \Delta w),$$

$$\hat{f} = [Cw_0 + \Delta w \sum_{k=1}^C v_{(k)}].$$

Definindo

$$S_1 = \sum_{k=1}^C v_{(k)},$$

temos que a saída do **CMAC** com pesos discretos será determinada



pela soma S_1 de C números inteiros $v_{(k)}$, $k=1,2,\dots,C$, conforme a equação 6.7:

$$\hat{f} = Cw_0 + S_1 \Delta w. \quad (6.7)$$

Supondo que encontramos os inteiros $v_{(1)}, v_{(2)}, \dots, v_{(C)}$ tais que $\hat{f} = f$, isto é, a saída do **CMAC** discreto coincide com a melhor aproximação discreta, $\hat{f}$, da saída desejada para o **CMAC**, r , segue que:

$$\hat{f} = r_0 + i\Delta r = Cw_0 + S_1 \Delta w = \hat{f}$$

$$i\Delta r = S_1 \Delta w$$

$$iKn\Delta r = S_1 Kn\Delta w$$

$$im\Delta w = S_1 Kn\Delta w.$$

Então, quando a convergência para um ponto de treinamento ocorre, a igualdade (6.8) é verificada:

$$im = S_1 Kn. \quad (6.8)$$

Logo, a cada passo de treinamento, a soma S_1 deve se comportar conforme a equação

$$S_1 \rightarrow \frac{im}{Kn},$$

e S_1 é uma soma de C números inteiros.

Assim, definiremos o erro inteiro devido à



apresentação do j -ésimo ponto de treinamento, $j=1,2,\dots,NP$, durante a t -ésima iteração do processo de treinamento, pela equação 6.9:

$$e_t^{(j)} = \frac{1^{(j)}_m}{Kn} - S_{1_t}^{(j)}. \quad (6.9)$$

Finalmente, por analogia com a regra **Delta de Widrow-Hoff**, definiremos o algoritmo para a atualização dos pesos discretos do **CMAC**, dado pela equação 6.10:

$$v_{t+1}^{(k)} = v_t^{(k)} + \frac{\beta}{C} e_t^{(j)}, \quad k = 1, 2, \dots, C. \quad (6.10)$$

β é a constante de aprendizado e, conforme exposto em 2.3.1, $0 < \beta \leq 1$.

O próximo capítulo apresenta resultados de simulações que comparam o desempenho deste algoritmo proposto com os outros algoritmos já existentes para o **CMAC**. Será mostrado que este novo algoritmo permite a reprodução de funções, com melhor precisão do que outros algoritmos propostos para o **CMAC**, utilizando uma memória física de tamanho similar.



Capítulo 7

APLICAÇÕES DO CMAC COM PESOS DISCRETOS

7.1. INTRODUÇÃO

Neste capítulo, descrevemos exemplos de reprodução de funções analiticamente conhecidas tomadas em [10]. Foi feita uma comparação entre as arquiteturas desenvolvidas por **Albus**, descrita no item 4.2, e por **Kraft & Campagna**, descrita no item 4.5, usando pesos contínuos, armazenados em ponto flutuante, e dispositivo para redução de memória física baseado em código **HASH** com suas respectivas versões baseadas em pesos discretos. As simulações mostraram que as novas estruturas apresentam melhores desempenhos com menores requisitos de memória física.

Este capítulo também apresenta resultados de controle por **modelo inverso generalizado** de um pêndulo forçado com um controlador neuronal tipo **CMAC** de **Albus** com pesos discretos. O objetivo é atingir um estado desejado através de uma trajetória conhecida após uma etapa de treinamento **off line**. Os resultados mostram que este objetivo é atingido de forma satisfatória.



7.2. REPRODUÇÃO DE FUNÇÕES ATRAVÉS DO CMAC DE KRAFT & CAMPAGNA

Consideremos as funções, com respectivas representações gráficas mostradas nas figuras (7.1), (7.2) e (7.3).

$$r = f_{\text{sen}}(x, y) = \text{sen}^2(x) \text{sen}^2(y), \quad (x, y) \in S = \{(x, y) \in \mathbb{R}^2, -\pi \leq x \leq \pi, -\pi \leq y \leq \pi\}$$

$$r = f_{\text{cos}}(x, y) = \frac{\cos(x^2 + y^2)}{\frac{1}{4}(x^2 + y^2)}, \quad (x, y) \in S = \{(x, y) \in \mathbb{R}^2, -\pi \leq x \leq \pi, -\pi \leq y \leq \pi\}$$

$$r = \text{canto}(x, y) = \begin{cases} 0,75x, & x < 0,3 \text{ e } y < 0,5 \\ 1, & \text{caso contrário.} \end{cases}$$

$$(x, y) \in S = \{(x, y) \in \mathbb{R}^2, -1 \leq x \leq 1, -1 \leq y \leq 1\}$$

Os valores de saída dessas três funções estão limitados ao intervalo $r \in [-1, 1]$, o que está ilustrado nas figuras (7.1) a (7.3). Supõe-se que os valores de r são obtidos através de um conversor A/D de 12 bits ($2n+1=4095$), o que proporciona 4095 valores discretos de r . O conjunto S é dividido em 784 subintervalos úteis conforme figuras (4.2) e (4.4). Foram executadas simulações em que os pesos estão confinados a conjuntos com $(2m+1)=4095$ ou $(2m+1)=127$ valores discretos devido a representações em 12 ou 7 bits, respectivamente. São comparados resultados com o CMAC de Kraft & Campagna utilizando uma memória física similar à ocupada pela configuração discreta. A mínima



quantidade de iterações executadas em cada caso é **100**, a fim de obtermos uma maior precisão na comparação dos tempos de estabilização do erro quadrático médio durante a etapa de treinamento. Define-se o tempo de estabilização como o tempo t_0 necessário para que

$$|EQM(t) - EQM_{FINAL}| < 0,01EQM_{FINAL}$$

sempre que $t \geq t_0$. Para cada caso, foi computado o erro quadrático médio entre os valores das funções originais e os valores fornecidos pelo CMAC ao final do treinamento, a partir de um conjunto de **60x60** pontos igualmente espaçados.

Os gráficos resultantes de cada simulação são baseados nos parâmetros cujas abreviaturas estão listadas abaixo:

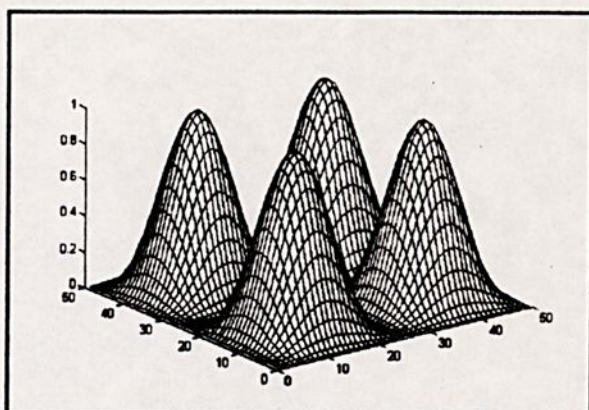


FIGURA 7.1: Função $f_{sen}()$ amostrada a partir de um conjunto de **50x50** pontos igualmente espaçados.

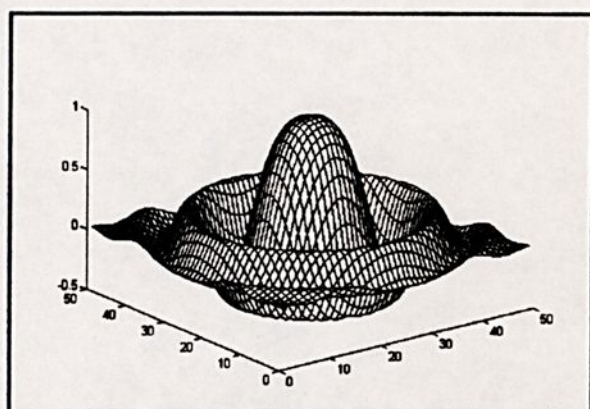


FIGURA 7.2: Função $f_{cos}()$ amostrada a partir de um conjunto com **50x50** pontos igualmente espaçados.

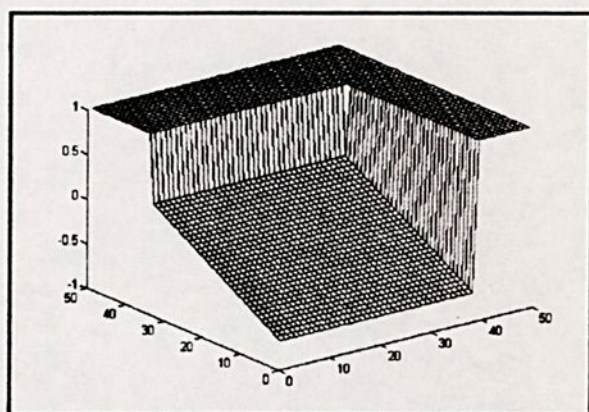


FIGURA 7.3: Função $canto()$ amostrada a partir de um conjunto com **50x50** pontos igualmente espaçados.

C: taxa de generalização;

TAB: especificação do número de intervalos da partição feita sobre o conjunto **S**;

NP,NPT: quantidade de pontos usados no treinamento;

MF: número de pesos do **CMAC** com dispositivo de redução de memória baseado no código **HASH**;

MEM: memória física requerida, em bytes;

BETA: taxa de aprendizado;

r_i: valor mínimo para a saída do **CMAC**;

r_s: valor máximo para a saída do **CMAC**;

w_i: valor mínimo que cada peso pode assumir;

w_s: valor máximo que cada peso pode assumir;

EQM: designa o erro quadrático médio da apresentação de todos os pontos de treinamento;

k: designa a iteração do processo de treinamento em andamento;

Nos gráficos a seguir, consideramos duas opções de quantidade de pontos usados para o treinamento: uma delas é alocar aleatoriamente um ponto em cada subintervalo da partição em **S**, formando um conjunto com **784** pontos; a outra alternativa considerada é distribuir aleatoriamente **200** pontos para o treinamento pelo conjunto **S**, com a garantia de que dois pontos não se encontram dentro do mesmo intervalo da partição.



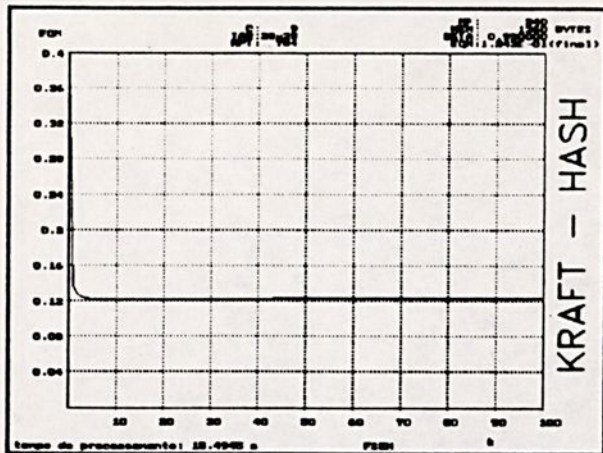


FIGURA 7.4: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

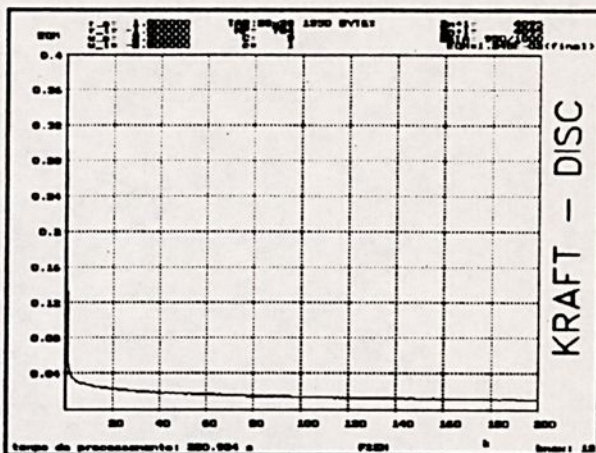


FIGURA 7.6: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

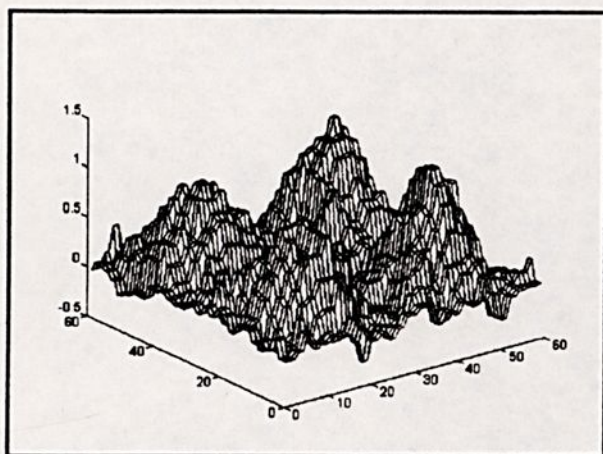


FIGURA 7.5: Aproximação resultante do treinamento representado na figura 7.4.

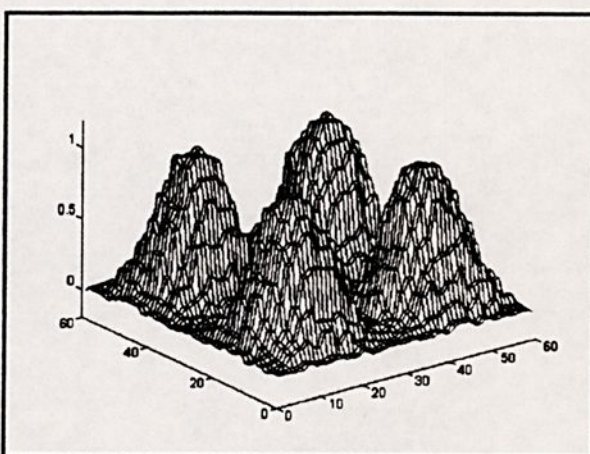


FIGURA 7.7: Aproximação resultante do treinamento representado na figura 7.6.

Tabela 7.1: Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $f_{sen}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 1350 bytes; a versão com código HASH necessita de 1360 bytes para armazenar 340 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,620 s	250,9 s
erro quadrático médio associado ao conjunto de pontos de teste	12,77 %	5,39 %

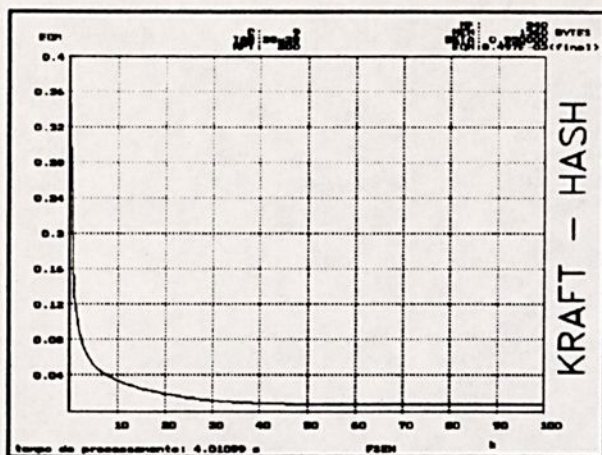


FIGURA 7.8: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

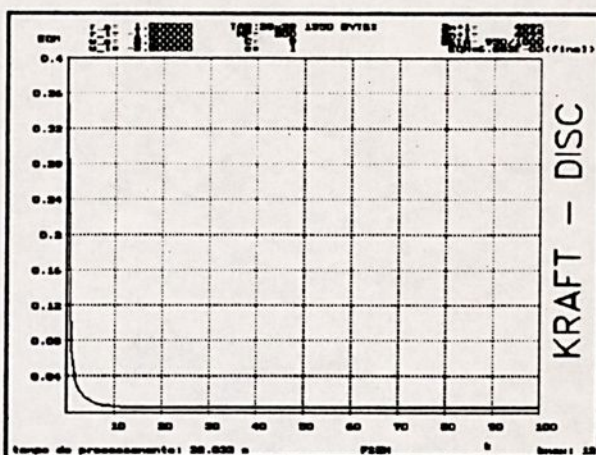


FIGURA 7.10: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

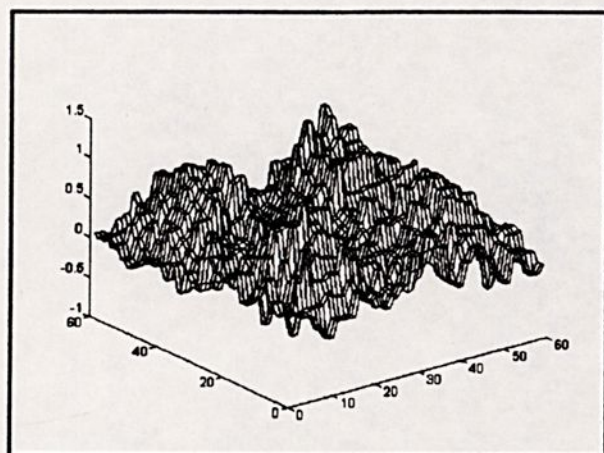


FIGURA 7.9: Aproximação resultante do treinamento representado na figura 7.8.

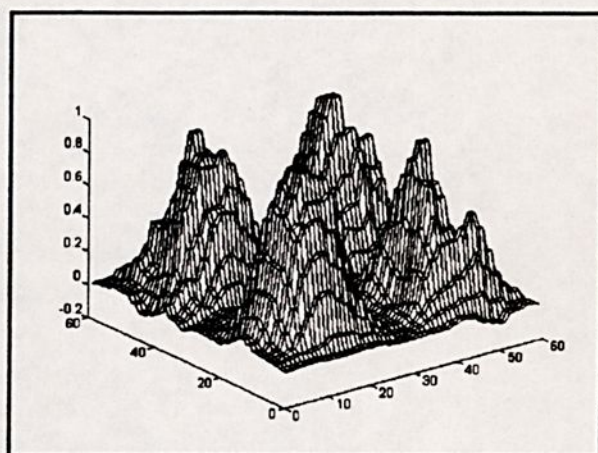


FIGURA 7.11: Aproximação resultante do treinamento representado na figura 7.10.

Tabela 7.2: Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $f_{sen}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 1350 bytes; a versão com código HASH necessita de 1360 bytes para armazenar 340 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	2,69 s	4,48 s
erro quadrático médio associado ao conjunto de pontos de teste	18,5 %	10,1 %

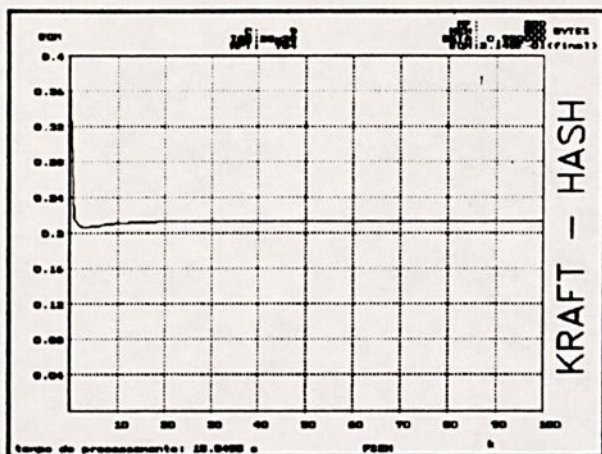


FIGURA 7.16: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

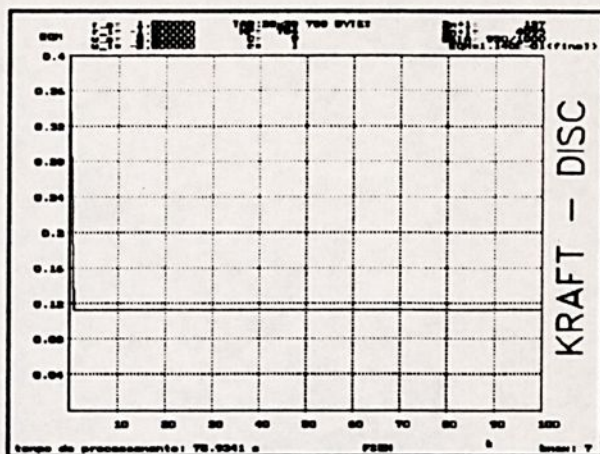


FIGURA 7.18: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

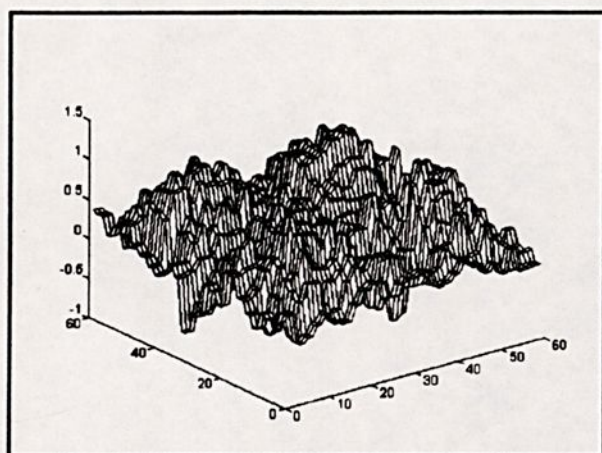


FIGURA 7.17: Aproximação resultante do treinamento representado na figura 7.16.

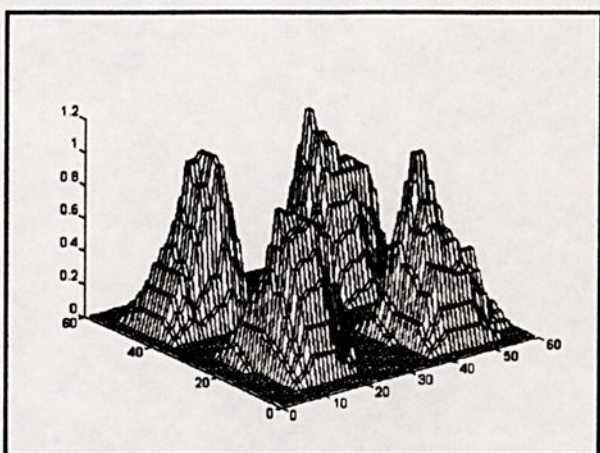


FIGURA 7.19: Aproximação resultante do treinamento representado na figura 7.18.

Tabela 7.3. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $f_{sen}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 788 bytes; a versão com código HASH necessita de 800 bytes para armazenar 200 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	1,71 s	0,760 s
erro quadrático médio associado ao conjunto de pontos de teste	21,5 %	12,4 %

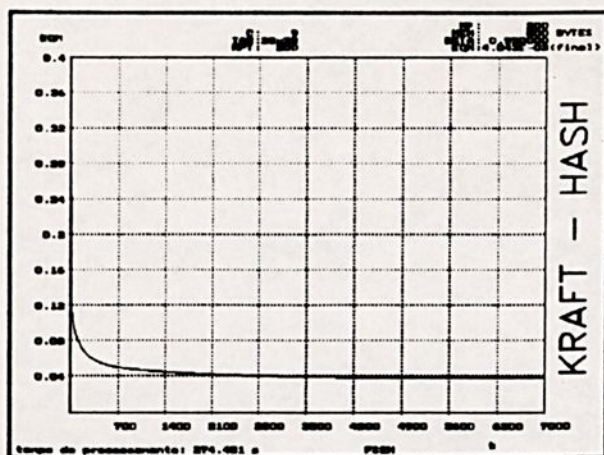


FIGURA 7.12: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

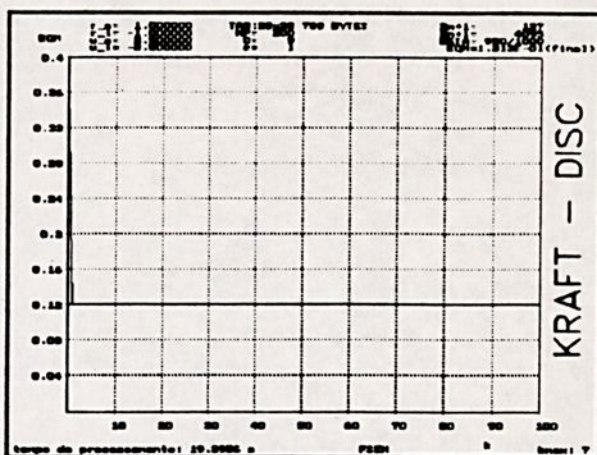


FIGURA 7.14: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

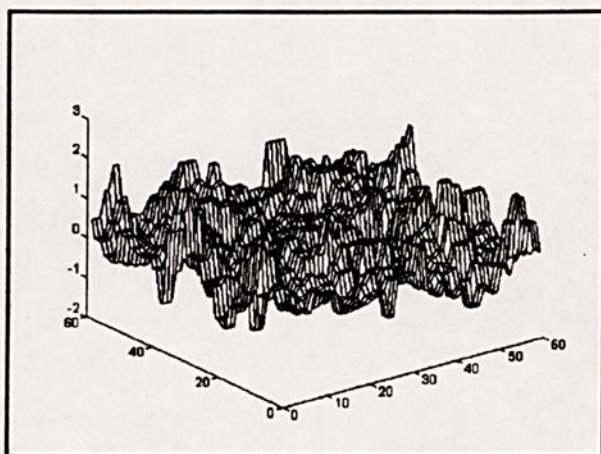


FIGURA 7.13: Aproximação resultante do treinamento representado na figura 7.12.

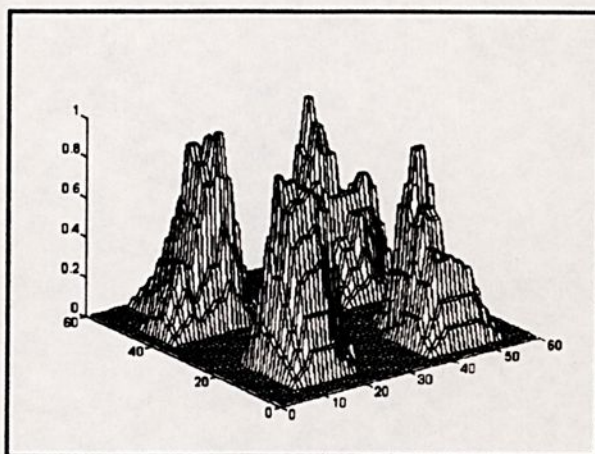


FIGURA 7.15: Aproximação resultante do treinamento representado na figura 7.14.

Tabela 7.4. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $f_{sen}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 788 bytes; a versão com código HASH necessita de 800 bytes para armazenar 200 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	150 s	0,194 s
erro quadrático médio associado ao conjunto de pontos de teste	91,3 %	17,6 %

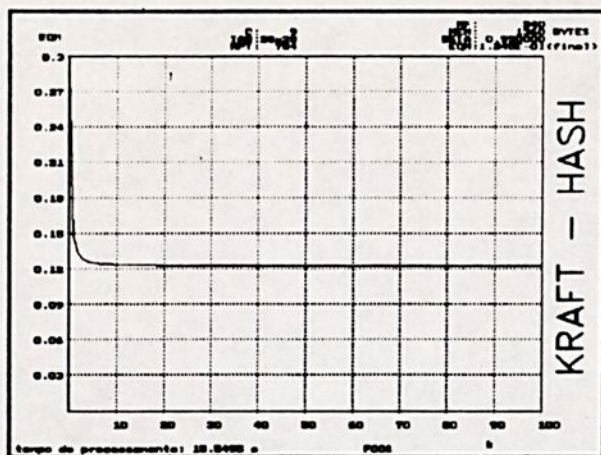


FIGURA 7.20: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

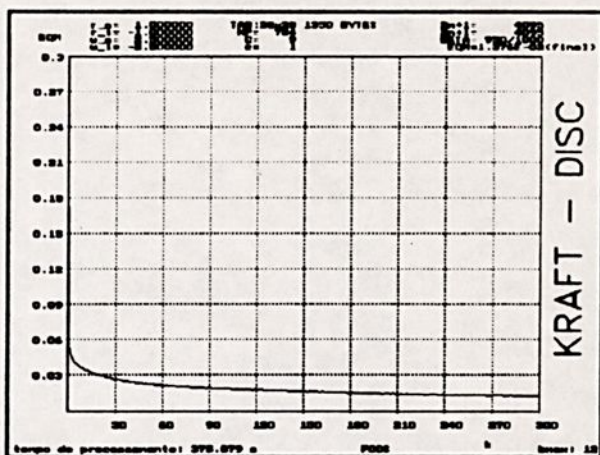


FIGURA 7.22: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

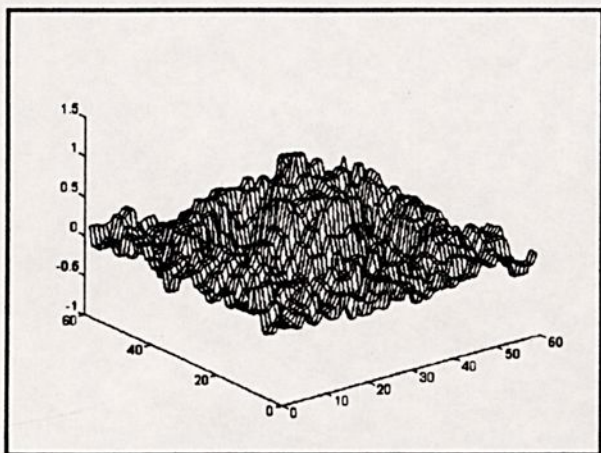


FIGURA 7.21: Aproximação resultante do treinamento representado na figura 7.20.

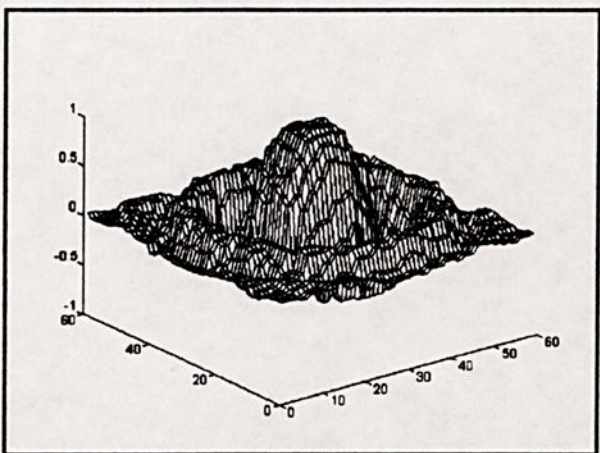


FIGURA 7.23: Aproximação resultante do treinamento representado na figura 7.22.

Tabela 7.5. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $f_{\text{cos}}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 1350 bytes; a versão com código HASH necessita de 1360 bytes para armazenar 340 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	1,24 s	376 s
erro quadrático médio associado ao conjunto de pontos de teste	13,1 %	7,04 %

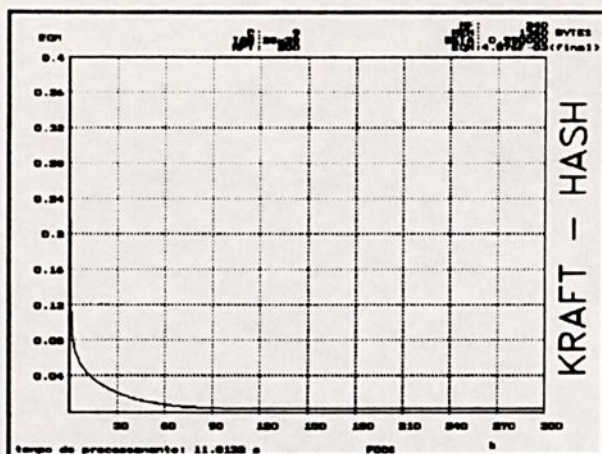


FIGURA 7.24: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

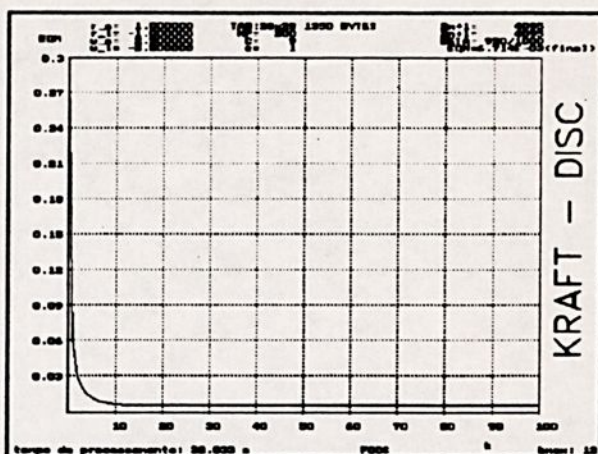


FIGURA 7.26: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

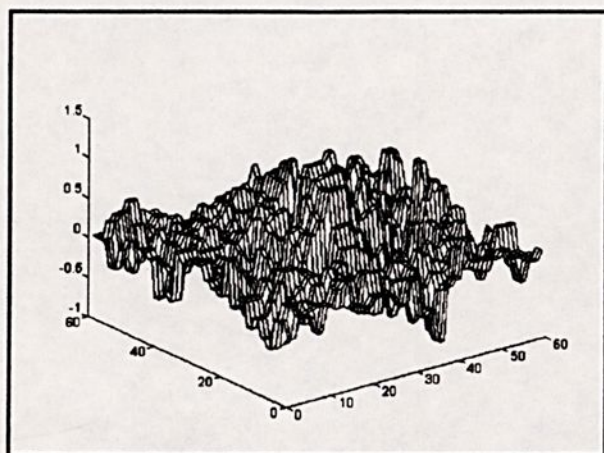


FIGURA 7.25: Aproximação resultante do treinamento representado na figura 7.24.

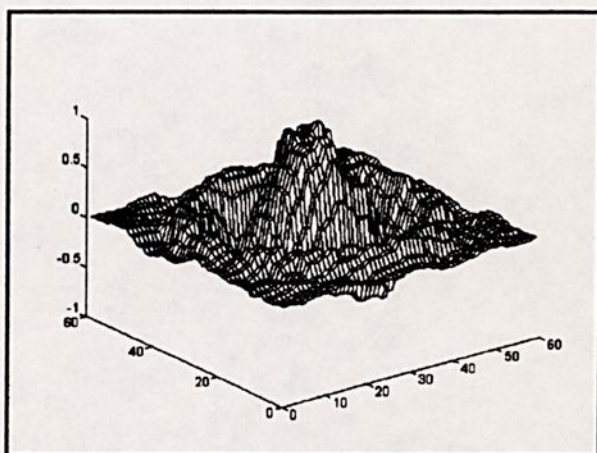


FIGURA 7.27: Aproximação resultante do treinamento representado na figura 7.26.

Tabela 7.6. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $f_{cos}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 1350 bytes; a versão com código HASH necessita de 1360 bytes para armazenar 340 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	5,75 s	4,80 s
erro quadrático médio associado ao conjunto de pontos de teste	22,2 %	10,4 %

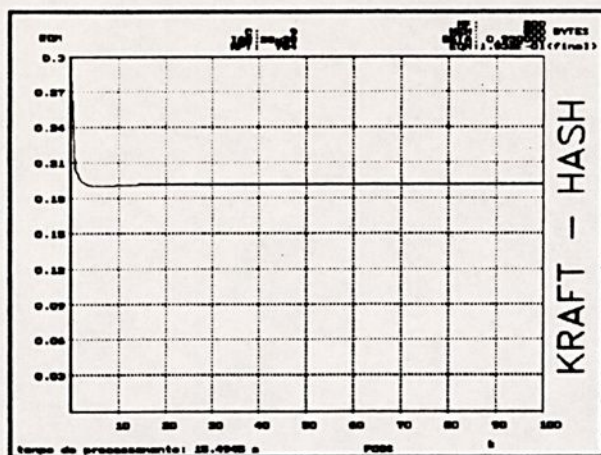


FIGURA 7.28: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

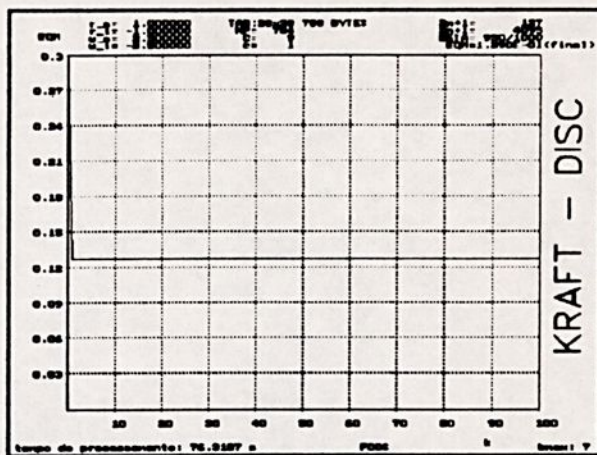


FIGURA 7.30: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

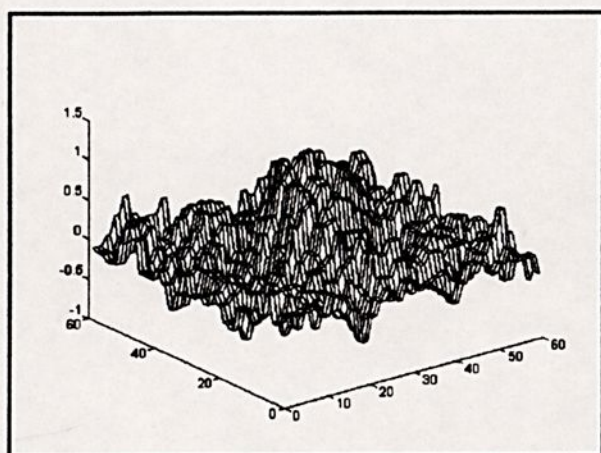


FIGURA 7.29: Aproximação resultante do treinamento representado na figura 7.28.

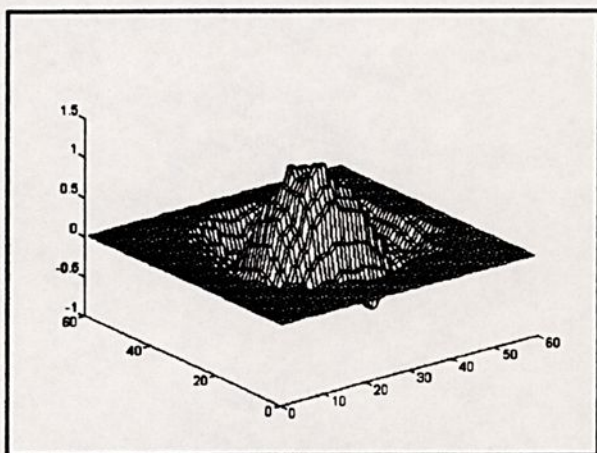


FIGURA 7.31: Aproximação resultante do treinamento representado na figura 7.30.

Tabela 7.7. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função `foos()` a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 788 bytes; a versão com código HASH necessita de 800 bytes para armazenar 200 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,465 s	0,763 s
erro quadrático médio associado ao conjunto de pontos de teste	19,5 %	13,0 %

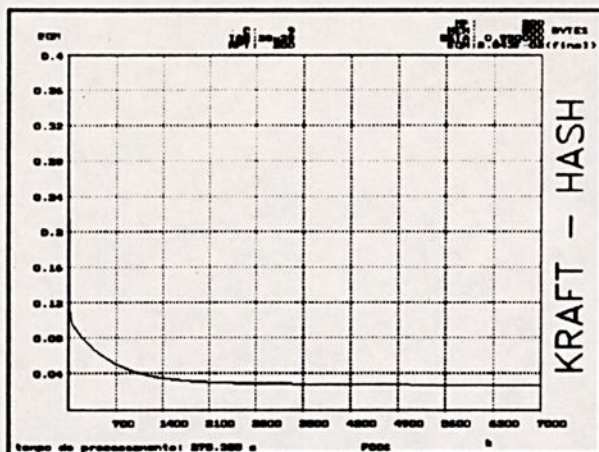


FIGURA 7.32 Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

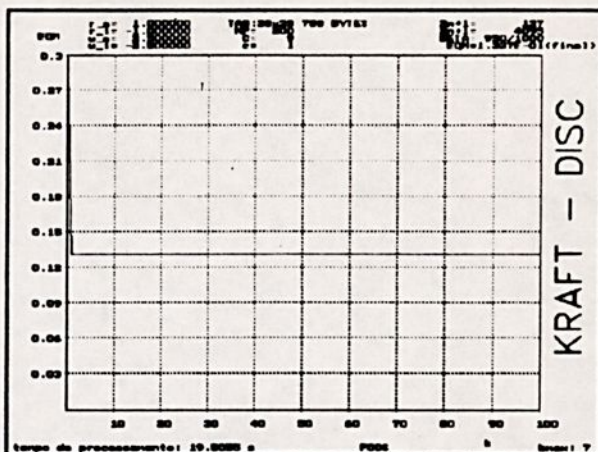


FIGURA 7.34: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

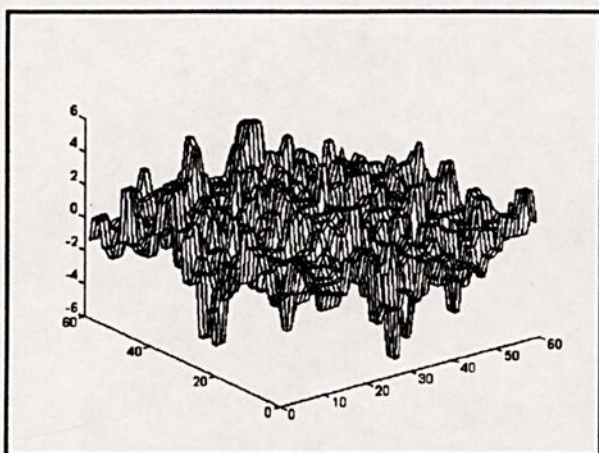


FIGURA 7.33: Aproximação resultante do treinamento representado na figura 7.32.

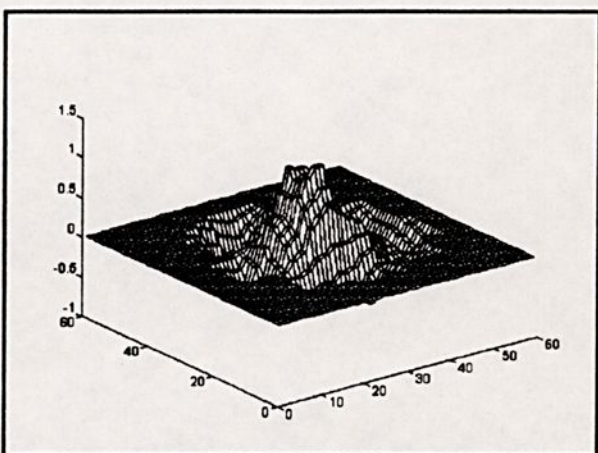


FIGURA 7.35: Aproximação resultante do treinamento representado na figura 7.34.

Tabela 7.8. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $f_{cos}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 788 bytes; a versão com código HASH necessita de 800 bytes para armazenar 200 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	246 s	0,195 s
erro quadrático médio associado ao conjunto de pontos de teste	118 %	16,0 %

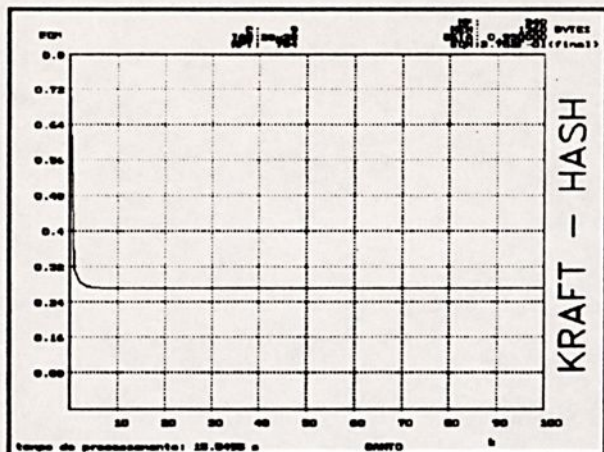


FIGURA 7.36: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

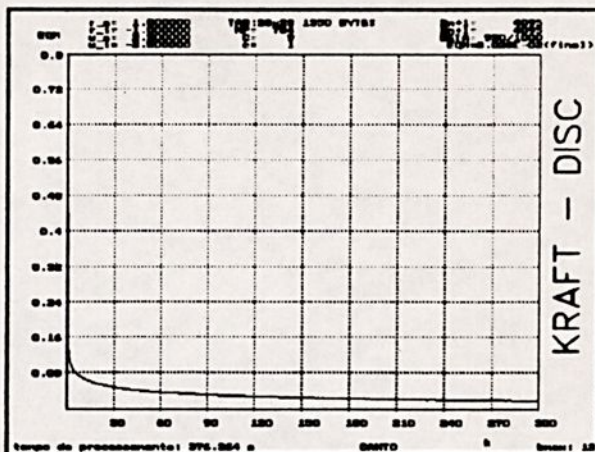


FIGURA 7.38: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

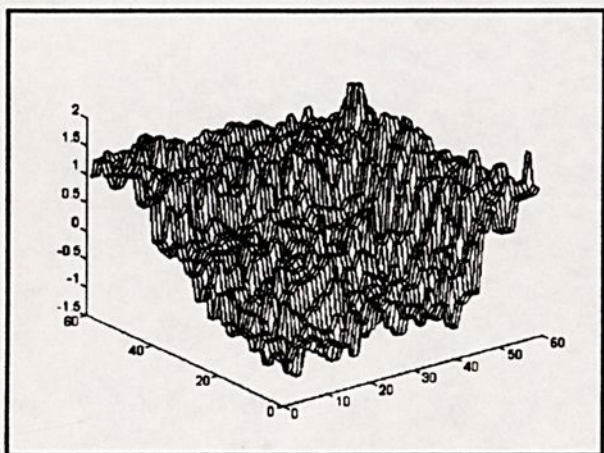


FIGURA 7.37: Aproximação resultante do treinamento representado na figura 7.36.

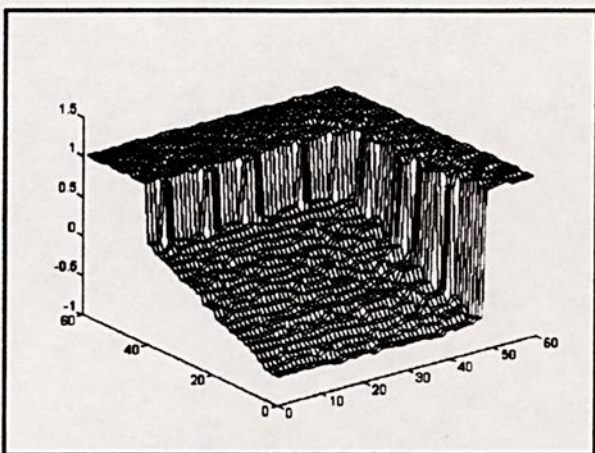


FIGURA 7.39: Aproximação resultante do treinamento representado na figura 7.38.

Tabela 7.9. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $\text{canto}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 1350 bytes; a versão com código HASH necessita de 1360 bytes para armazenar 340 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,777 s	367 s
erro quadrático médio associado ao conjunto de pontos de teste	29,1 %	16,1 %

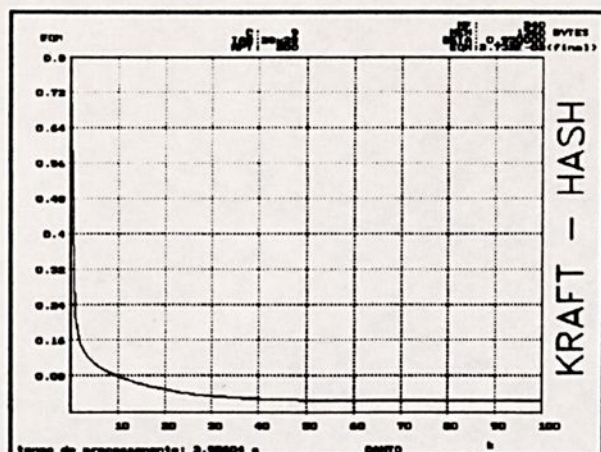


FIGURA 7.40: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

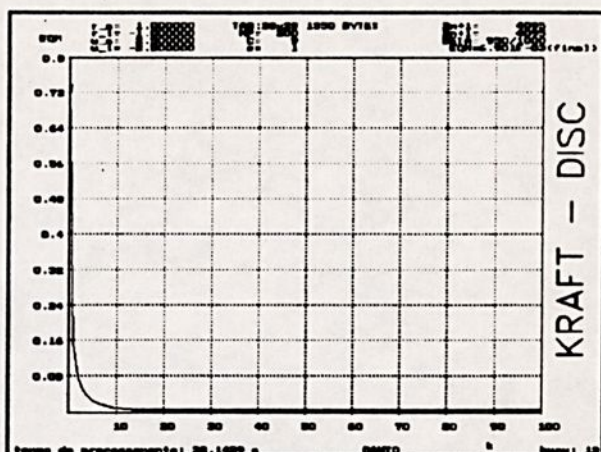


FIGURA 7.42: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

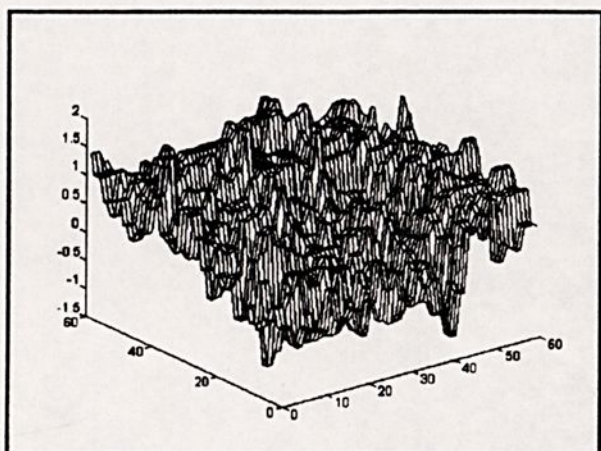


FIGURA 7.41: Aproximação resultante do treinamento representado na figura 7.40.

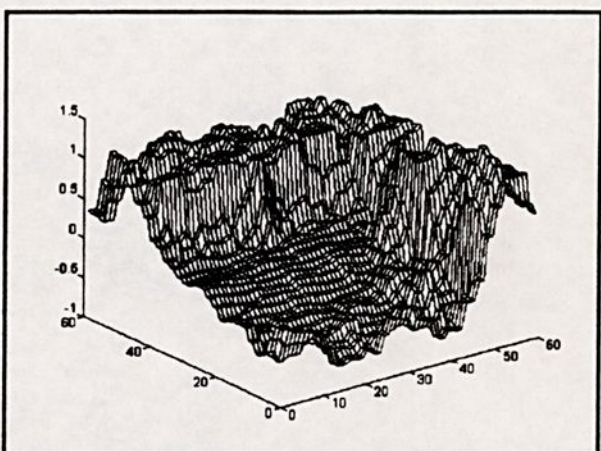


FIGURA 7.43: Aproximação resultante do treinamento representado na figura 7.42.

Tabela 7.10. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $\text{canto}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 1350 bytes; a versão com código HASH necessita de 1360 bytes para armazenar 340 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	2,57 s	6,75 s
erro quadrático médio associado ao conjunto de pontos de teste	42,3 %	26,2 %

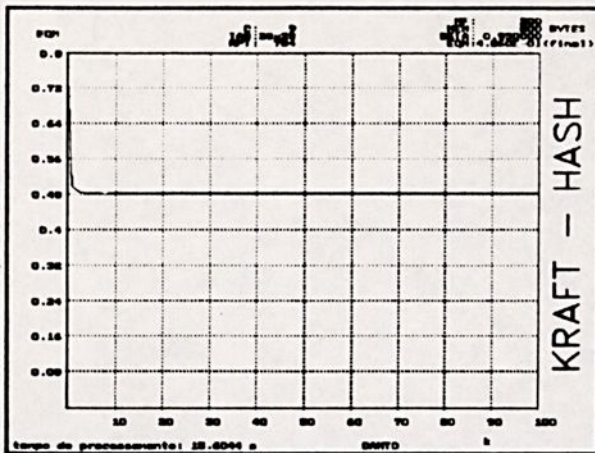


FIGURA 7.44: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

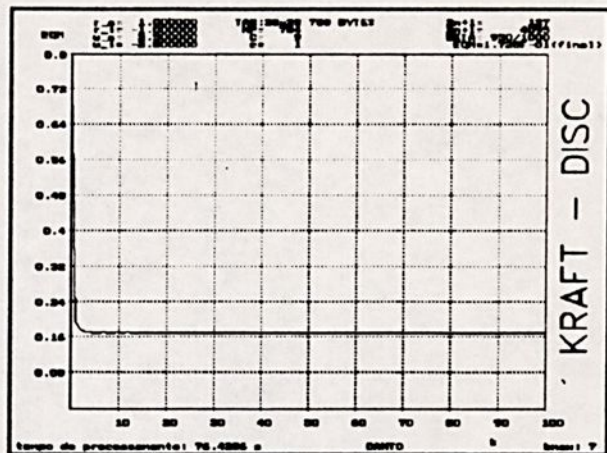


FIGURA 7.46: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

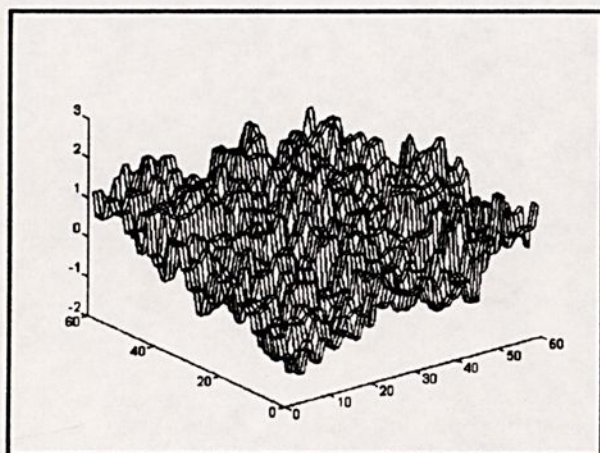


FIGURA 7.45: Aproximação resultante do treinamento representado na figura 7.44.

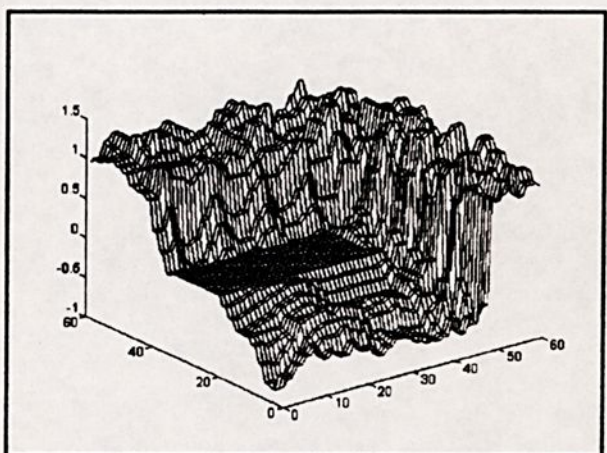


FIGURA 7.47: Aproximação resultante do treinamento representado na figura 7.46.

Tabela 7.11. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $\text{canto}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 788 bytes; a versão com código HASH necessita de 800 bytes para armazenar 200 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,311 s	9,94 s
erro quadrático médio associado ao conjunto de pontos de teste	48,7 %	21,4 %

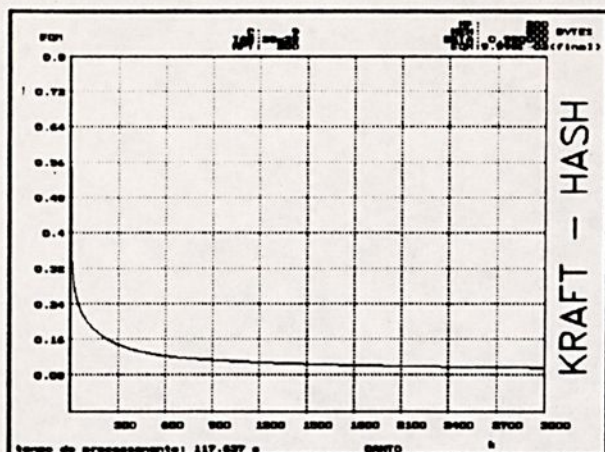


FIGURA 7.48: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

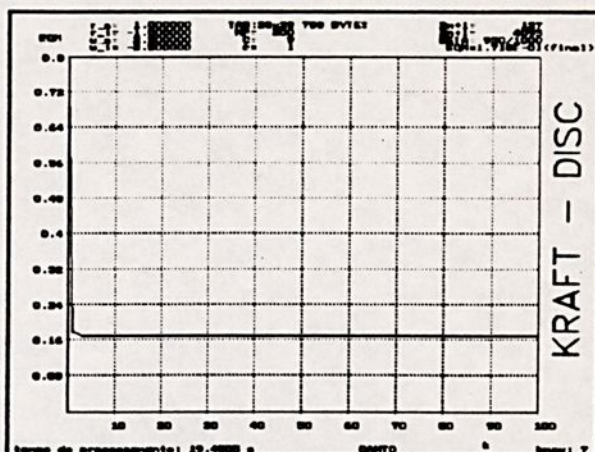


FIGURA 7.50: Gráfico do erro quadrático médio durante o treinamento do CMAC de Kraft e Campagna.

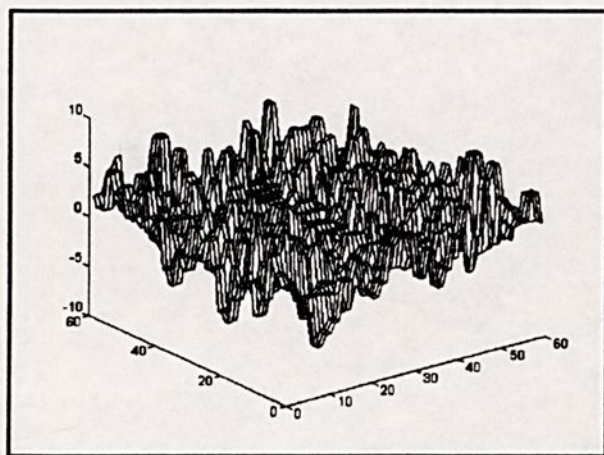


FIGURA 7.49: Aproximação resultante do treinamento representado na figura 7.48.

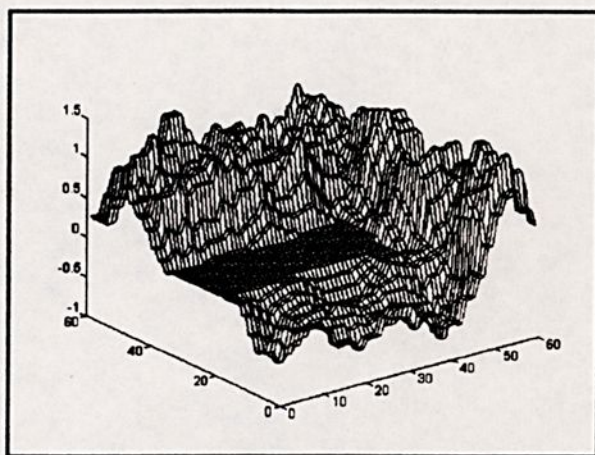


FIGURA 7.51: Aproximação resultante do treinamento representado na figura 7.50.

Tabela 7.12. Comparação dos resultados de treinamento do CMAC de Kraft e Campagna para reprodução da função $\text{canto}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 788 bytes; a versão com código HASH necessita de 800 bytes para armazenar 200 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	118 s	0,584 s
erro quadrático médio associado ao conjunto de pontos de teste	214 %	30,9 %

Os gráficos e tabelas mostram que, em **100%** dos casos, o uso de **12** bits para representação dos pesos oferece melhores resultados do que **7** bits.

Além disso, um conjunto com **784** pontos de treinamento proporcionou menores erros quadráticos do que os conseguidos com **200** pontos, porém, usando pesos discretos a diferença nem sempre é significativa (compare as tabelas 7.5 com 7.6 e 7.7 com 7.8, coluna **pesos discretos**).

A tabela 7.13 reúne todos os resultados das simulações, podemos comparar o uso de código **HASH** com o uso de pesos discretos: em todos os casos, a nova proposta para redução de memória física apresentou resultados mais precisos na reprodução das funções de teste.



Tabela 7.13: Resultados totais obtidos a partir das simulações representadas nas figuras 7.4 a 7.51.

fsen()		NP=784		NP=200	
		HASH	DISC	HASH	DISC
12 bits	estab	0,620 s	250,9 s	2,69 s	4,48 s
	EQM	12,77 %	5,39 %	18,5 %	10,1 %
7 bits	estab	1,71 s	0,760 s	150 s	0,194 s
	EQM	21,5 %	12,4 %	91,3 %	17,6 %

fcos()		NP=784		NP=200	
		HASH	DISC	HASH	DISC
12 bits	estab	1,24 s	376 s	5,75 s	4,80 s
	EQM	13,1 %	7,04 %	22,2 %	10,4 %
7 bits	estab	0,465 s	0,763 s	246 s	0,195 s
	EQM	19,5 %	13,0 %	118 %	16,0 %

canto()		NP=784		NP=200	
		HASH	DISC	HASH	DISC
12 bits	estab	0,777 s	367 s	2,57 s	6,75 s
	EQM	29,1 %	16,1 %	42,3 %	26,2 %
7 bits	estab	0,311 s	9,94 s	118 s	0,584 s
	EQM	48,7 %	21,4 %	214 %	30,9 %



7.3. REPRODUÇÃO DE FUNÇÕES ATRAVÉS DO CMAC DE ALBUS

Neste tópico apresentamos resultados de simulações para a reprodução das mesmas funções citadas no item anterior, a saber:

$$r = f_{\text{sen}}(x, y) = \text{sen}^2(x) \text{sen}^2(y), \quad (x, y) \in S = \{(x, y) \in \mathbb{R}^2, -\pi \leq x \leq \pi, -\pi \leq y \leq \pi\}$$

$$r = f_{\text{cos}}(x, y) = \frac{\cos(x^2 + y^2)}{e^{\frac{1}{4}(x^2 + y^2)}}, \quad (x, y) \in S = \{(x, y) \in \mathbb{R}^2, -\pi \leq x \leq \pi, -\pi \leq y \leq \pi\}$$

$$r = \text{canto}(x, y) = \begin{cases} 0,75x, & x < 0,3 \text{ e } y < 0,5 \\ 1, & \text{caso contrário} \end{cases},$$

$$(x, y) \in S = \{(x, y) \in \mathbb{R}^2, -1 \leq x \leq 1, -1 \leq y \leq 1\}$$

com o CMAC de Albus nas versões com código **HASH** e com pesos discretos.

Os gráficos resultantes de cada simulação apresentam os parâmetros utilizados:

C: taxa de generalização;

TAB: especificação do número de intervalos da partição feita sobre **S**;

NP, NPT: quantidade de pontos usados no treinamento;

MF: número de pesos do CMAC;

MEM: memória física requerida;

BETA: taxa de aprendizado;



r_i: valor mínimo para a saída do **CMAC**;

r_s: valor máximo para a saída do **CMAC**;

w_i: valor mínimo que cada peso pode assumir;

w_s: valor máximo que cada peso pode assumir;

EQM: designa o erro quadrático médio da apresentação de todos os pontos de treinamento;

k: designa a iteração em andamento.



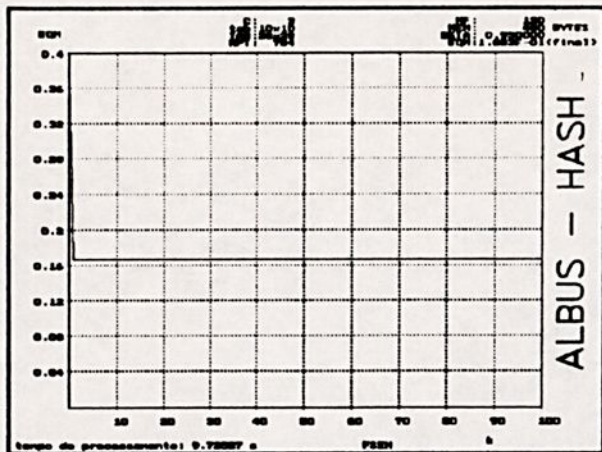


FIGURA 7.52: Gráfico do erro quadrático médio durante o treinamento do CMAC de Albus.

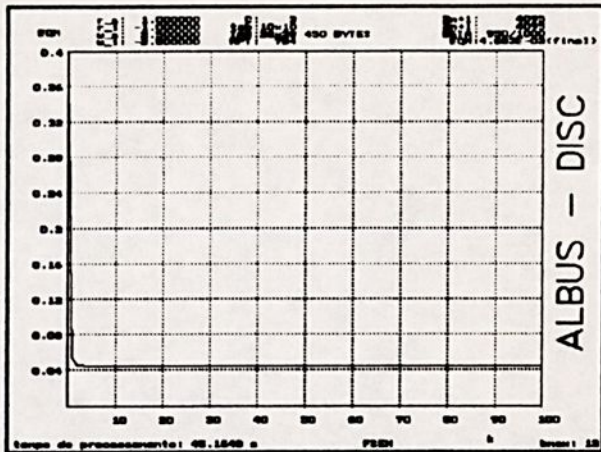


FIGURA 7.54: Gráfico do erro quadrático médio durante o treinamento do CMAC de Albus.

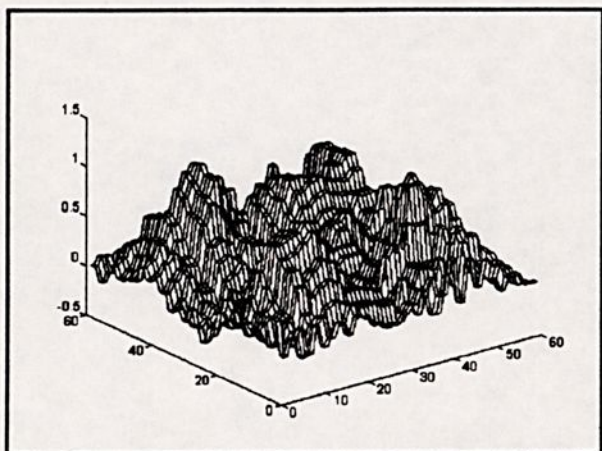


FIGURA 7.53: Aproximação resultante do treinamento representado na figura 7.52.

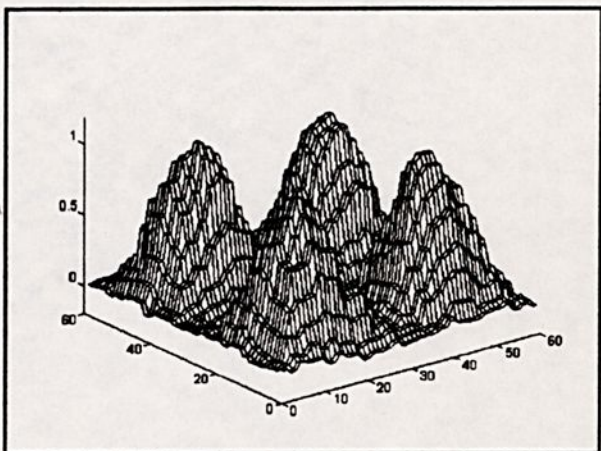


FIGURA 7.55: Aproximação resultante do treinamento representado na figura 7.54.

Tabela 7.14. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função $f_{sen}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 450 bytes; a versão com código HASH necessita de 480 bytes para armazenar 120 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,0973 s	1,35 s
erro quadrático médio associado ao conjunto de pontos de teste	17,2 %	6,49 %

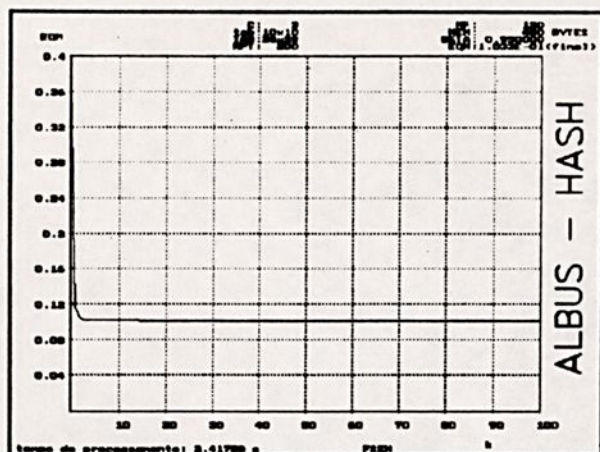


FIGURA 7.56: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

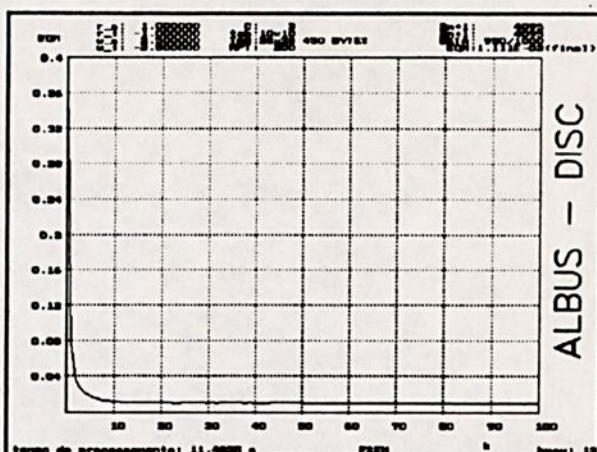


FIGURA 7.58: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

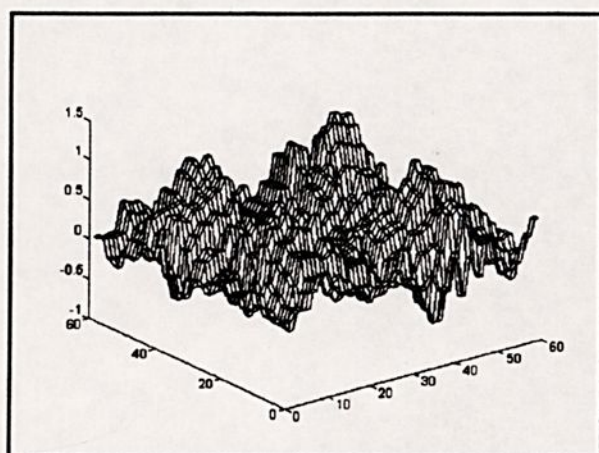


FIGURA 7.57: Aproximação resultante do treinamento representado na figura 7.56.

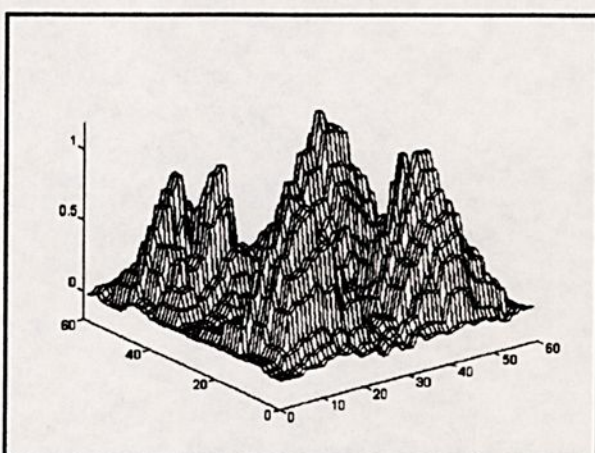


FIGURA 7.59: Aproximação resultante do treinamento representado na figura 7.58.

Tabela 7.15. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função $f_{sen}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 450 bytes; a versão com código HASH necessita de 480 bytes para armazenar 120 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,0725 s	9,19 s
erro quadrático médio associado ao conjunto de pontos de teste	18,5 %	10,9 %

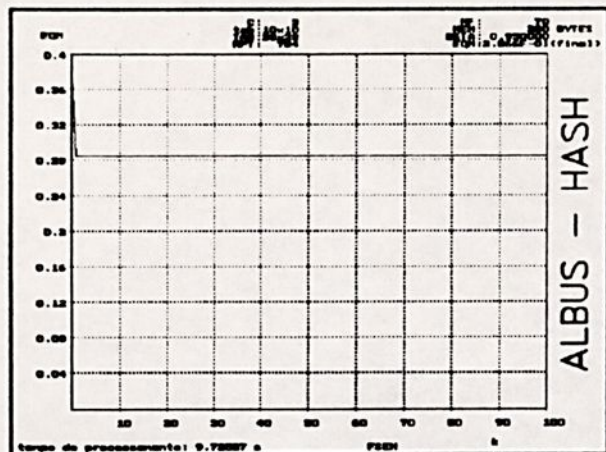


FIGURA 7.60: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

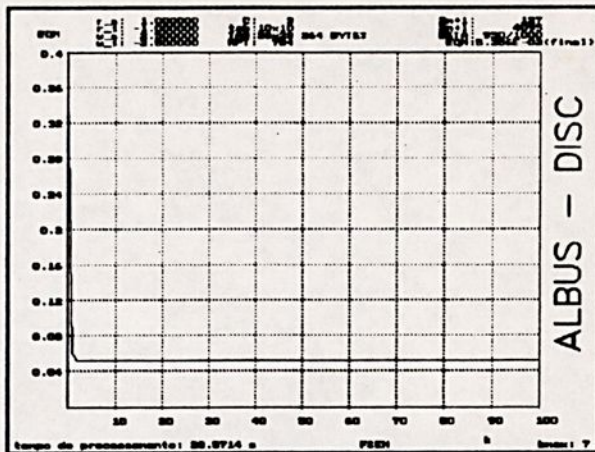


FIGURA 7.62: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

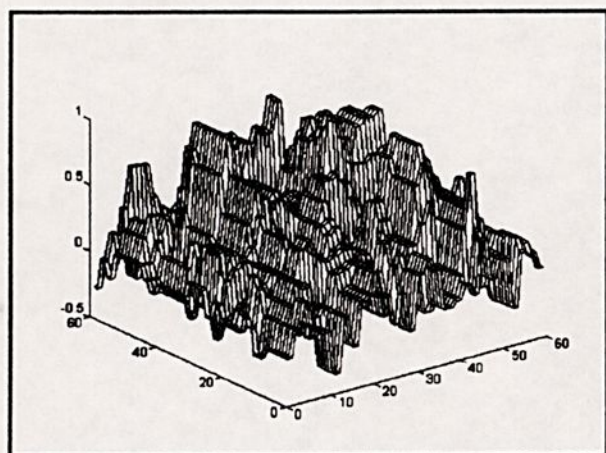


FIGURA 7.61: Aproximação resultante do treinamento representado na figura 7.60.

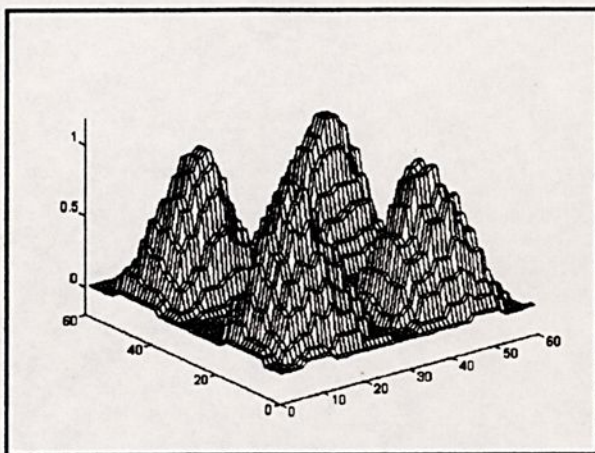


FIGURA 7.63: Aproximação resultante do treinamento representado na figura 7.62.

Tabela 7.16. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função $fsen()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 264 bytes; a versão com código HASH necessita de 280 bytes para armazenar 70 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,0973 s	0,571 s
erro quadrático médio associado ao conjunto de pontos de teste	29,0 %	6,47 %

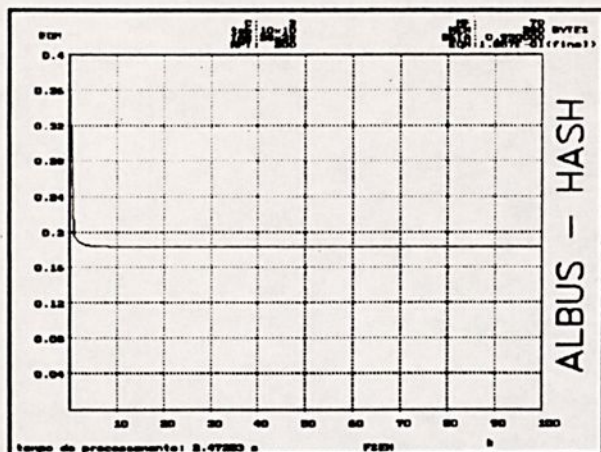


FIGURA 7.64: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

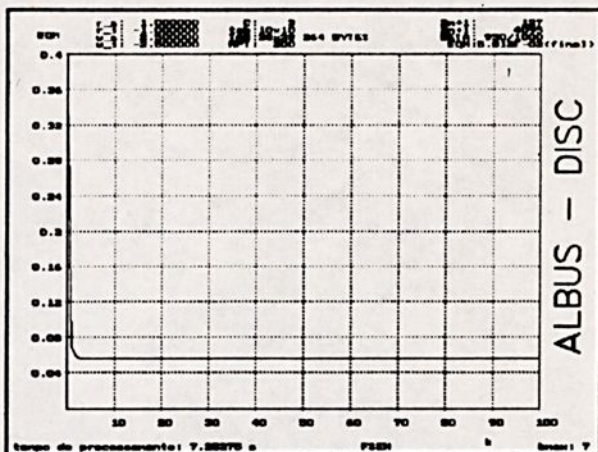


FIGURA 7.66: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

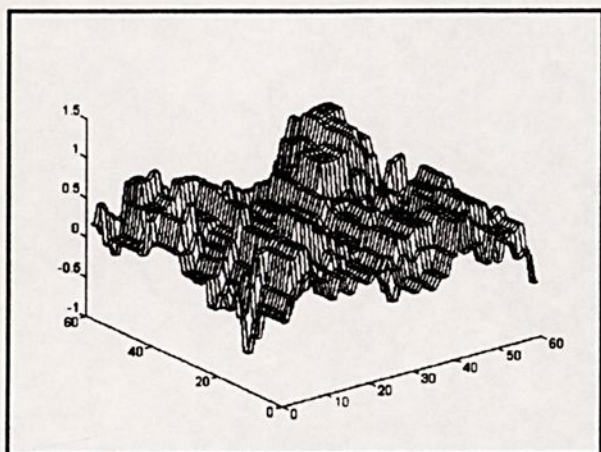


FIGURA 7.65: Aproximação resultante do treinamento representado na figura 7.64.

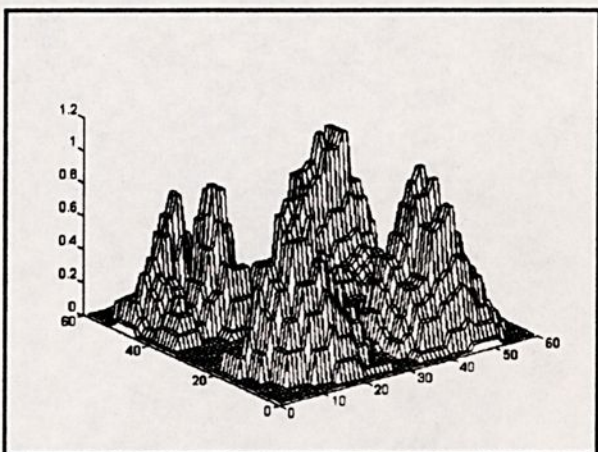


FIGURA 7.67: Aproximação resultante do treinamento representado na figura 7.66:

Tabela 7.17. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função $fsen()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 264 bytes; a versão com código HASH necessita de 280 bytes para armazenar 70 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,0989 s	0,218 s
erro quadrático médio associado ao conjunto de pontos de teste	26,0 %	13,9 %

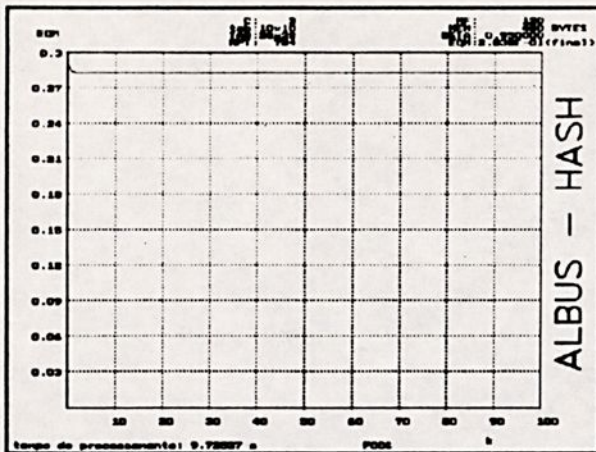


FIGURA 7.68: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

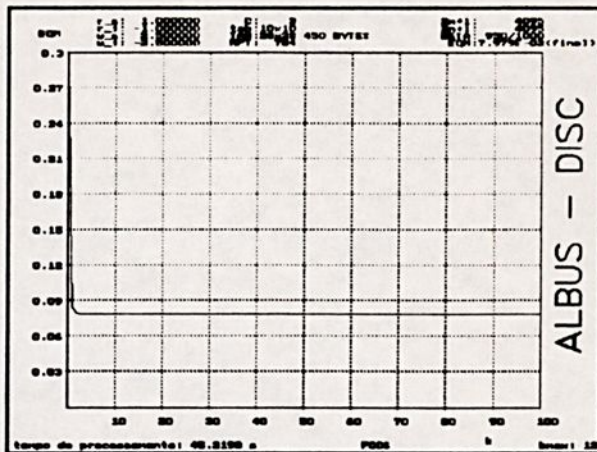


FIGURA 7.70: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

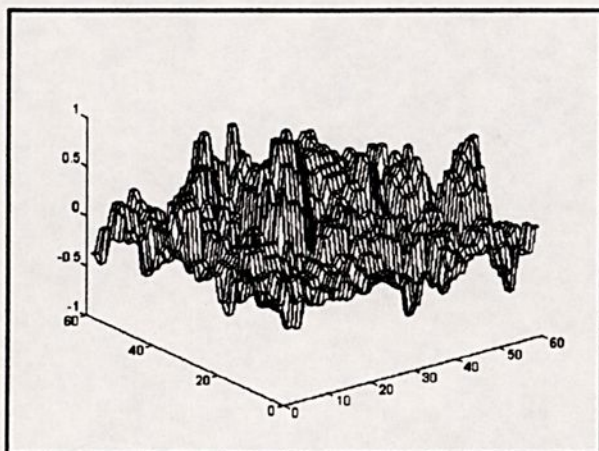


FIGURA 7.69: Aproximação resultante do treinamento representado na figura 7.68.

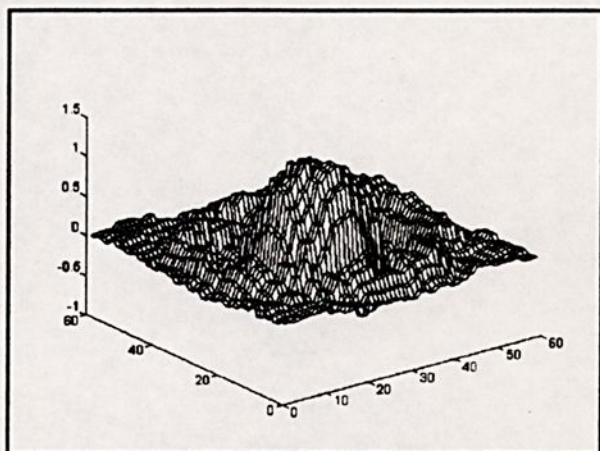


FIGURA 7.71: Aproximação resultante do treinamento representado na figura 7.70.

Tabela 7.18. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função $f_{cos}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 450 bytes; a versão com código HASH necessita de 480 bytes para armazenar 120 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,0973 s	0,904 s
erro quadrático médio associado ao conjunto de pontos de teste	28,1 %	9,40 %

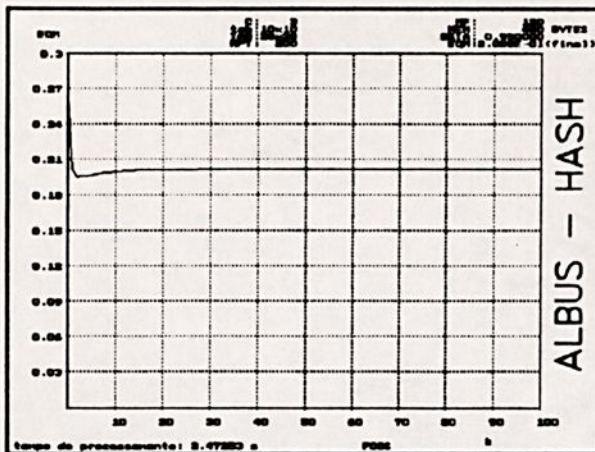


FIGURA 7.72: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

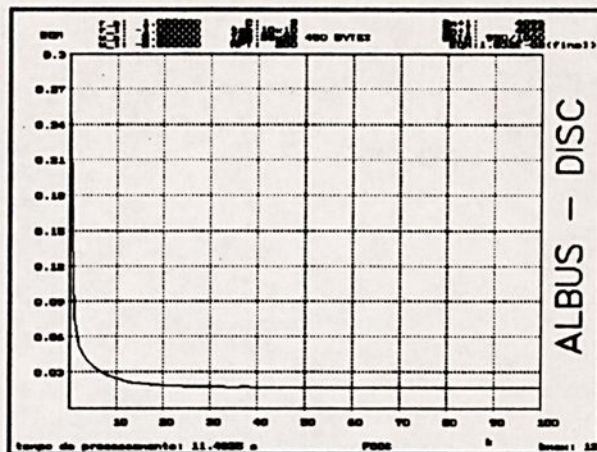


FIGURA 7.74: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

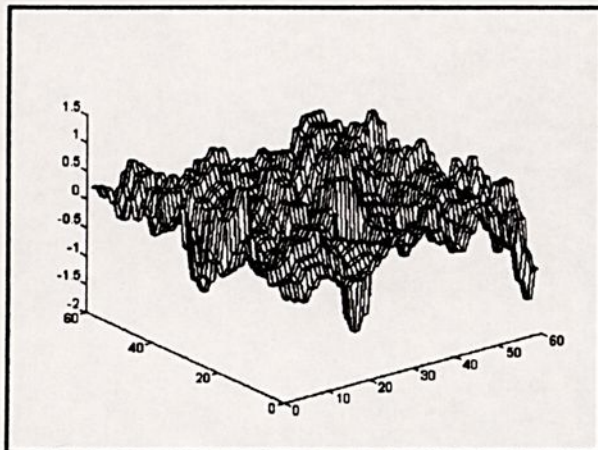


FIGURA 7.73: Aproximação resultante do treinamento representado na figura 7.72.

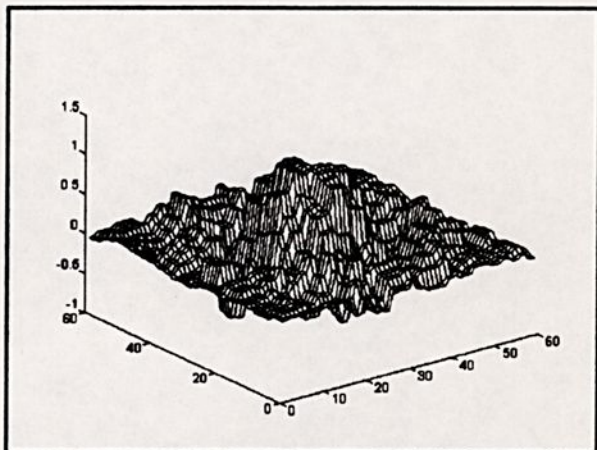


FIGURA 7.75: Aproximação resultante do treinamento representado na figura 7.74.

Tabela 7.19. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função $f_{\text{cos}}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 450 bytes; a versão com código HASH necessita de 480 bytes para armazenar 120 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,272 s	5,28 s
erro quadrático médio associado ao conjunto de pontos de teste	31,5 %	14,7 %

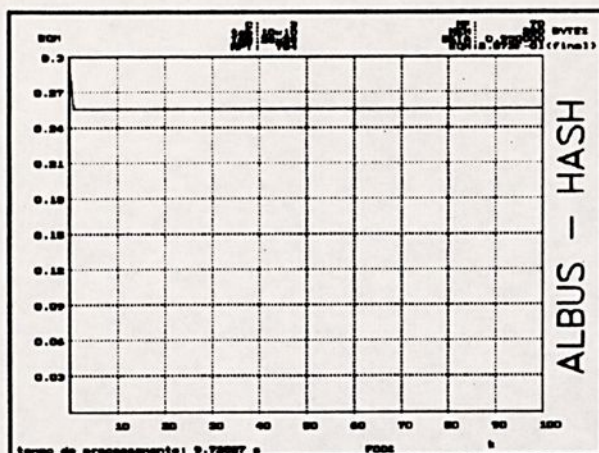


FIGURA 7.76: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

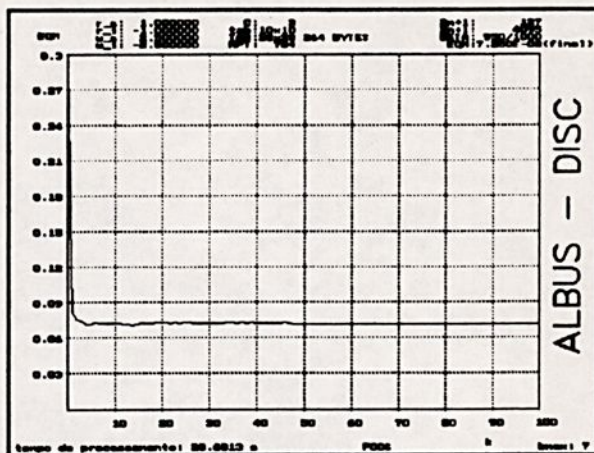


FIGURA 7.78: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

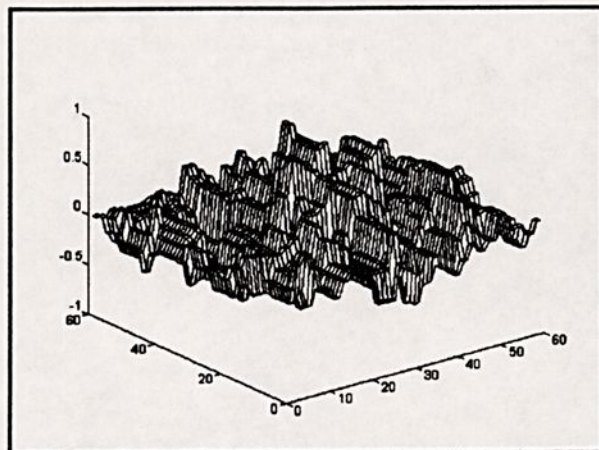


FIGURA 7.77: Aproximação resultante do treinamento representado na figura 7.76.

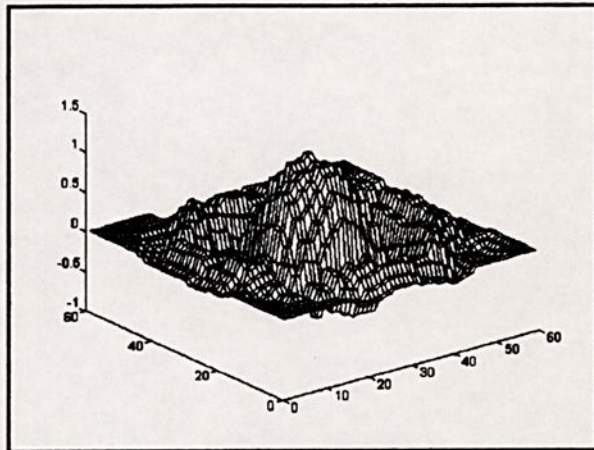


FIGURA 7.79: Aproximação resultante do treinamento representado na figura 7.78.

Tabela 7.20. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função $f_{\text{foo}}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 264 bytes; a versão com código HASH necessita de 280 bytes para armazenar 70 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,0973 s	13,5 s
erro quadrático médio associado ao conjunto de pontos de teste	25,7 %	8,61 %

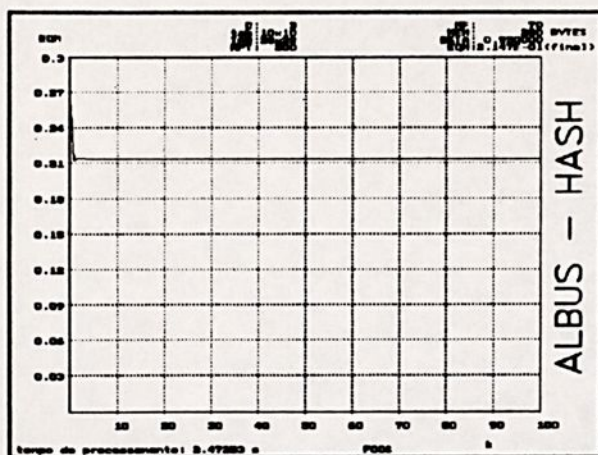


FIGURA 7.80: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

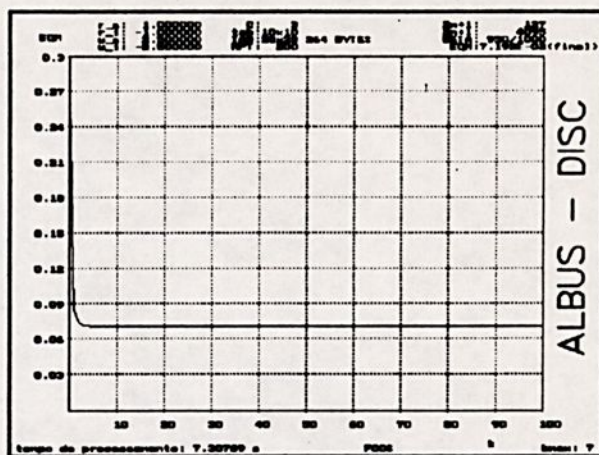


FIGURA 7.82: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

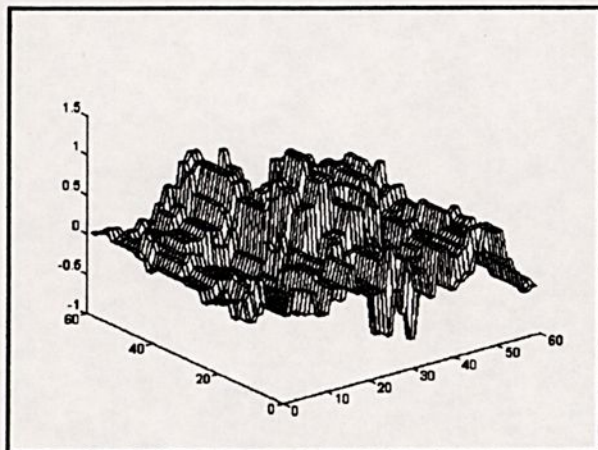


FIGURA 7.81: Aproximação resultante do treinamento representado na figura 7.80.

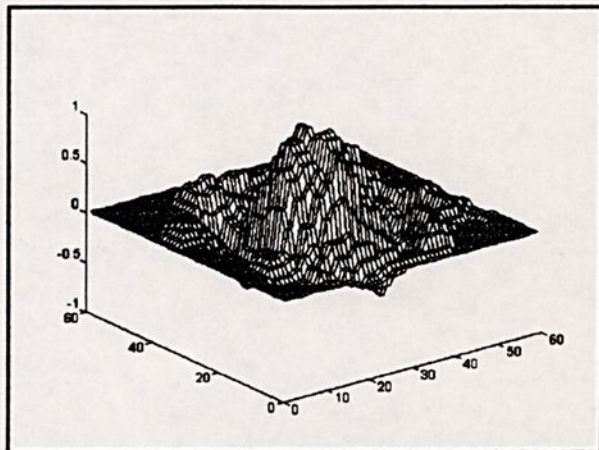


FIGURA 7.83: Aproximação resultante do treinamento representado na figura 7.82.

Tabela 7.21. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função $f_{\text{cos}}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 264 bytes; a versão com código HASH necessita de 280 bytes para armazenar 70 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,0247 s	0,219 s
erro quadrático médio associado ao conjunto de pontos de teste	28,4 %	13,8 %

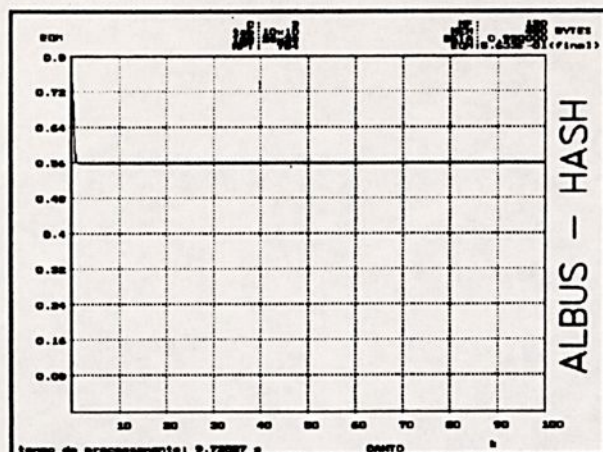


FIGURA 7.84: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

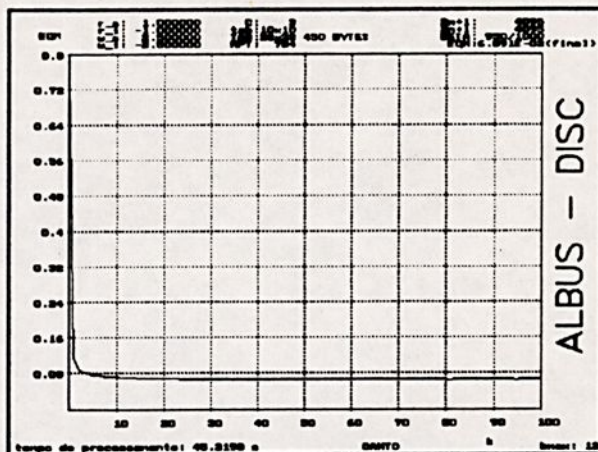


FIGURA 7.86: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

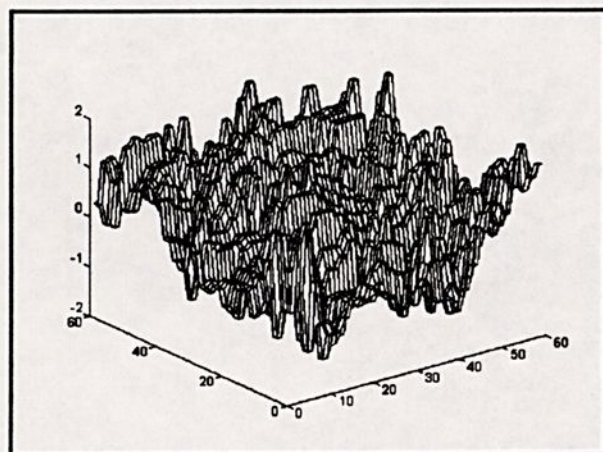


FIGURA 7.85: Aproximação resultante do treinamento representado na figura 7.84.

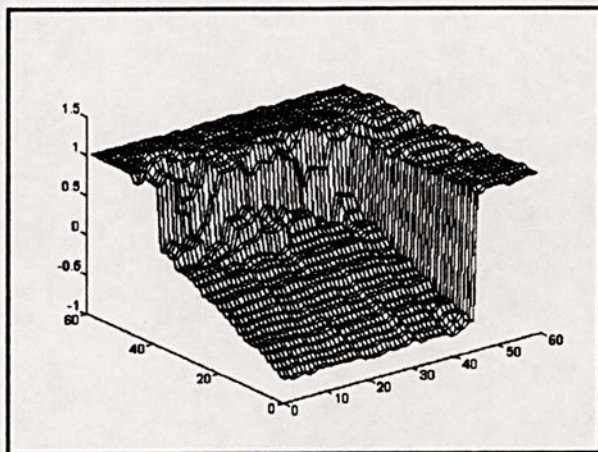


FIGURA 7.87: Aproximação resultante do treinamento representado na figura 7.86.

Tabela 7.22. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função $f_{\text{canto}}()$ a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 450 bytes; a versão com código HASH necessita de 480 bytes para armazenar 120 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,0973 s	20,1 s
erro quadrático médio associado ao conjunto de pontos de teste	55,9 %	8,16 %

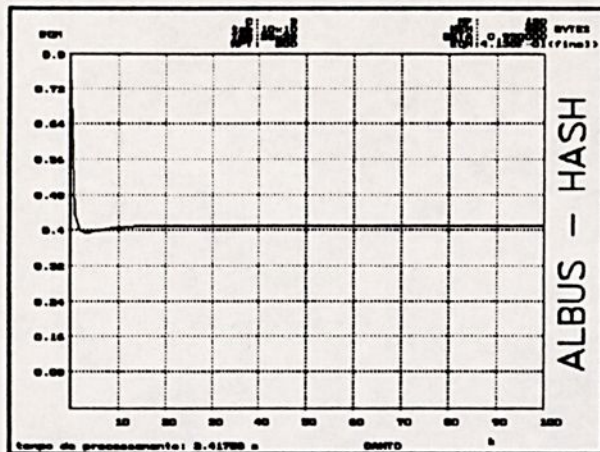


FIGURA 7.88: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

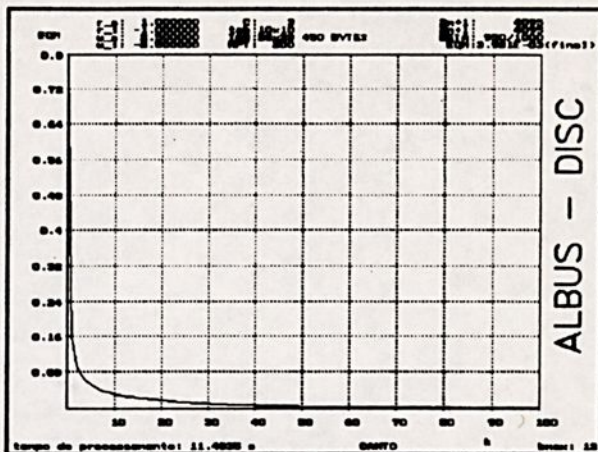


FIGURA 7.90: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

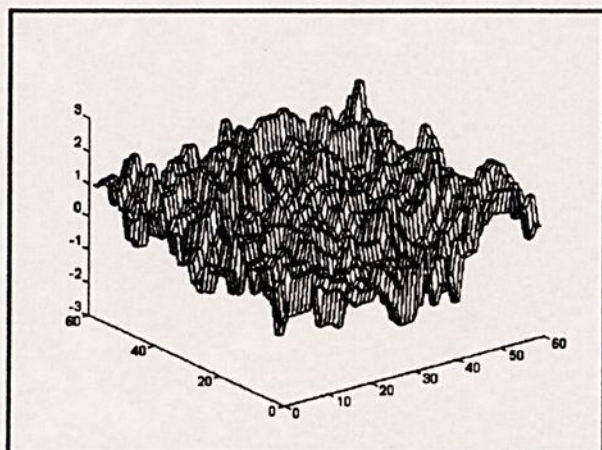


FIGURA 7.89: Aproximação resultante do treinamento representado na figura 7.88.

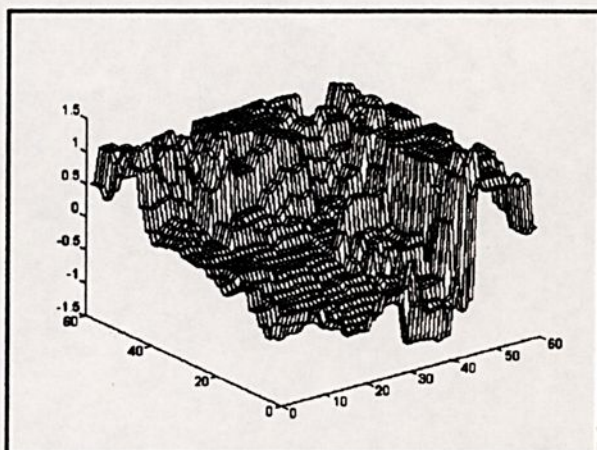


FIGURA 7.91: Aproximação resultante do treinamento representado na figura 7.90.

Tabela 7.23. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função $f_{\text{canto}}()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 12 bits requerendo 450 bytes; a versão com código HASH necessita de 480 bytes para armazenar 120 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,242 s	11,5 s
erro quadrático médio associado ao conjunto de pontos de teste	67,2 %	25,2 %

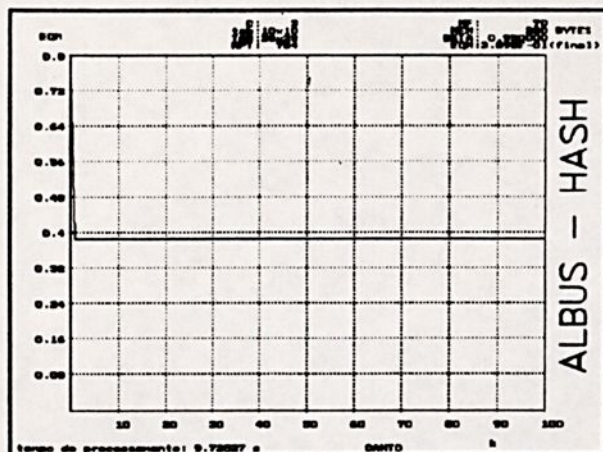


FIGURA 7.92: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

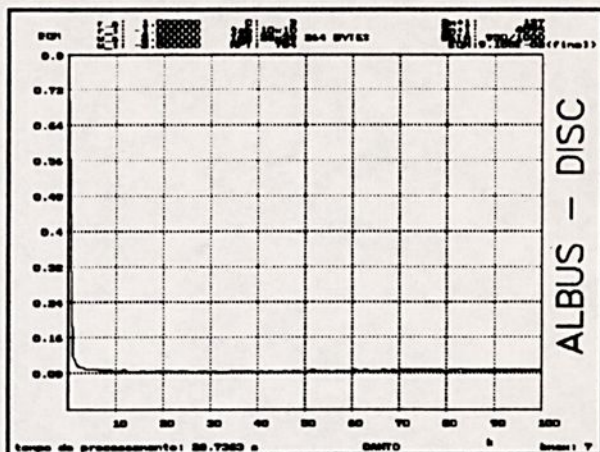


FIGURA 7.94: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

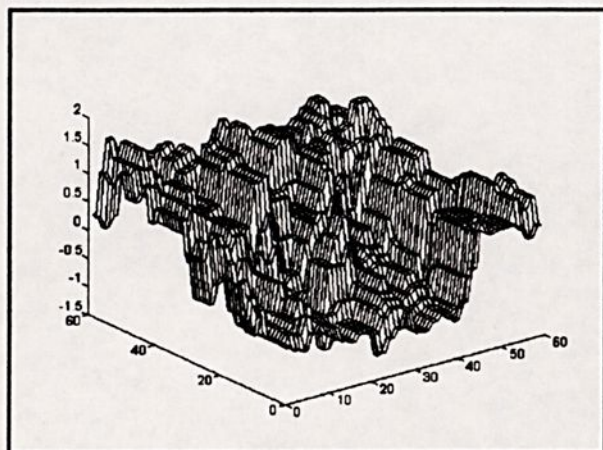


FIGURA 7.93: Aproximação resultante do treinamento representado na figura 7.92.

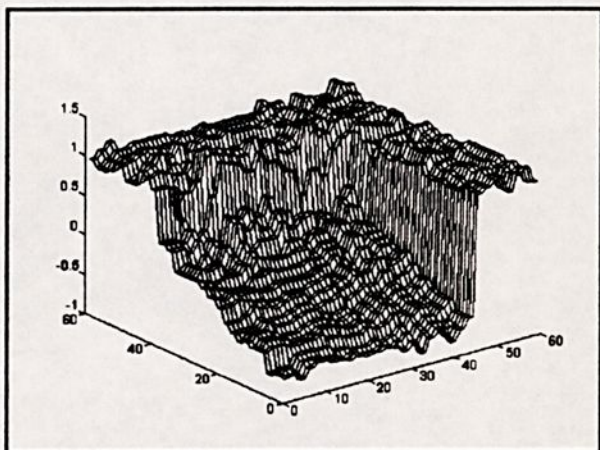


FIGURA 7.95: Aproximação resultante do treinamento representado na figura 7.94.

Tabela 7.24. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função `fcanto()` a partir de um conjunto de 784 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 264 bytes; a versão com código HASH necessita de 280 bytes para armazenar 70 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EGM	0,0973 s	28,7 s
erro quadrático médio associado ao conjunto de pontos de teste	39,3 %	10,2 %

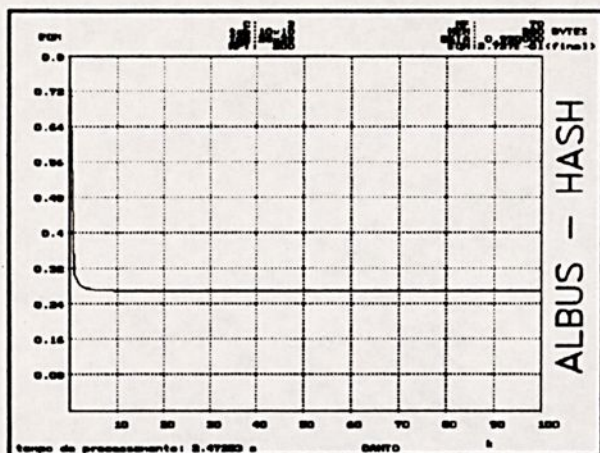


FIGURA 7.96: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

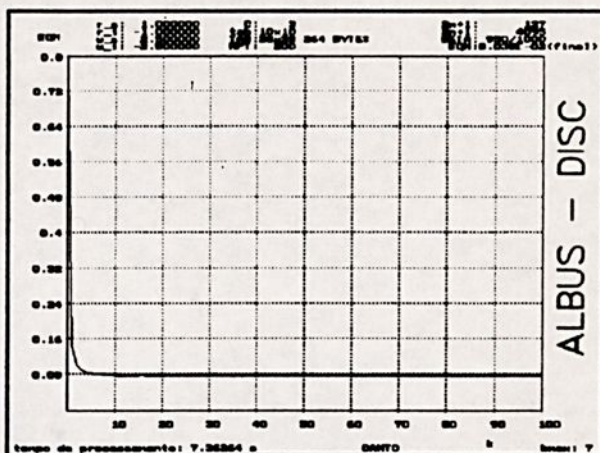


FIGURA 7.98: Gráfico do erro quadrático médio do treinamento do CMAC de Albus.

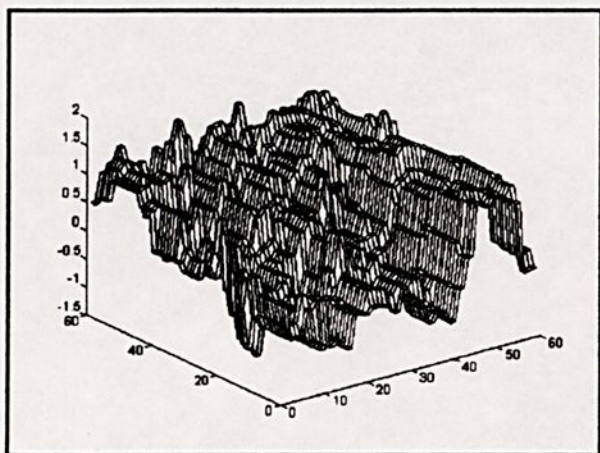


FIGURA 7.97: Aproximação resultante do treinamento representado na figura 7.96.

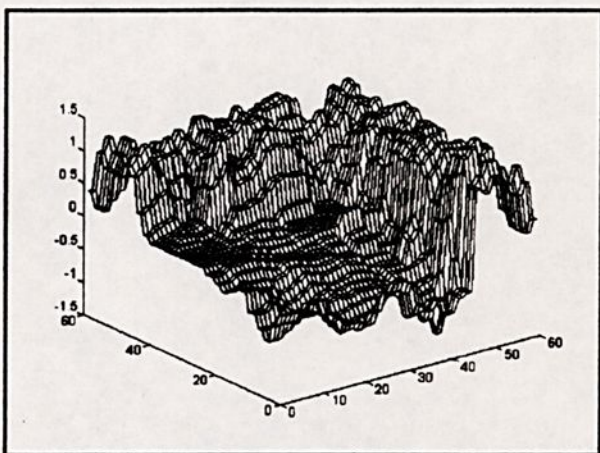


FIGURA 7.99: Aproximação resultante do treinamento representado na figura 7.98.

Tabela 7.25. Comparação dos resultados de treinamento do CMAC de Albus para reprodução da função $fcanto()$ a partir de um conjunto de 200 pontos de treinamento. Os pesos discretos são armazenados com 7 bits requerendo 264 bytes; a versão com código HASH necessita de 280 bytes para armazenar 70 pesos tipo ponto flutuante de 4 bytes cada um.

	código HASH	pesos discretos
tempo de estabilização do EQM	0,148 s	1,25 s
erro quadrático médio associado ao conjunto de pontos de teste	40,4 %	26,3 %

Analogamente à comparação feita para o **CMAC** de **Kraft & Campagna**, os gráficos e tabelas apresentados mostram que, na grande maioria dos casos, representar pesos com **12** bits oferece maior precisão do que com **7** bits.

Em todos os casos em que o treinamento é realizado com **784** pontos obtém-se uma melhor qualidade de reprodução das funções de teste relativamente ao treinamento com **200** pontos, embora essa diferença seja pouco significativa em quase todos os casos, o que pode ser observado nas tabelas (exceto nas tabelas 7.22 e 7.23).

A tabela 7.26 mostra que foram obtidos menores erros quadráticos médios através do dispositivo de redução de memória baseado em pesos discretos e isso pode ser observado em todos os casos simulados.



Tabela 7.26: Resultados totais obtidos a partir das simulações representadas nas figuras 7.52 a 7.99.

fsen()		NP=784		NP=200	
		HASH	DISC	HASH	DISC
12 bits	estab	0,0973 s	1,35 s	0,0725 s	9,19 s
	EQM	17,2 %	6,49 %	18,5 %	10,9 %
7 bits	estab	0,0973 s	0,571 s	0,0989 s	0,218 s
	EQM	29,0 %	6,47 %	26,0 %	13,9 %

fcos()		NP=784		NP=200	
		HASH	DISC	HASH	DISC
12 bits	estab	0,0973 s	0,904 s	0,272 s	5,28 s
	EQM	28,1 %	9,40 %	31,5 %	14,7 %
7 bits	estab	0,0973 s	13,5 s	0,0247 s	0,219 s
	EQM	25,7 %	8,61 %	28,4 %	13,8 %

canto()		NP=784		NP=200	
		HASH	DISC	HASH	DISC
12 bits	estab	0,0973 s	20,1 s	0,242 s	11,5 s
	EQM	55,9 %	8,16 %	67,2 %	25,2 %
7 bits	estab	0,0973 s	28,7 s	0,148 s	1,25 s
	EQM	39,3 %	10,2 %	40,4 %	26,3 %



7.4. EXEMPLO DE CONTROLE POR MODELO INVERSO

Descrevemos aqui um exemplo de rastreamento de uma trajetória conhecida por um pêndulo forçado excitado pela saída de um controlador tipo **CMAC** de **Albus** com pesos discretos.

O objetivo é que o pêndulo permaneça na posição inicial escolhida θ_i durante o intervalo $[0, t_i]$, $t_i=1s$ e, logo em seguida, se movimente até a posição final θ_f desejada, num intervalo de tempo estipulado $\Delta t_s=2s$ e permaneça em θ_f desde $t_f=t_i+\Delta t_s$ até o final do tempo de simulação.

O modelo matemático discreto do pêndulo forçado é dado por:

$$\alpha_{k+1} = f(\theta_k, \omega_k, T_k)$$

$$\alpha_{k+1} = -\frac{B}{ml^2} \omega_k - \frac{g}{l} \sin \theta_k + \frac{T_k}{ml^2} \quad (7.1)$$

$$\alpha_k = \frac{d^2 \theta_k}{dt^2}$$

$$\omega_k = \frac{d\theta_k}{dt}$$

sendo θ_k o ângulo formado pelo braço do pêndulo com a linha vertical na posição inferior no instante k , positivo quando medido no sentido anti-horário. A integração no tempo é feita pelo método de **Euler**, conforme as equações (7.2), com passo de



$$\begin{aligned}\omega_{k+1} &= \omega_k + \alpha_{k+1} \tau \\ \theta_{k+1} &= \theta_k + \omega_{k+1} \tau\end{aligned}\tag{7.2}$$

integração $\tau=0,005s$. As amostragens das variáveis de estado e de α_k ocorrem a cada $P=4$ passos de integração da equação (7.1), assim, a cada incremento de k , κ é incrementado $P=4$ vezes. Isto significa que o valor da excitação pode mudar a cada $0,02s$. Supõe-se que o torque T_k é fornecido por um atuador tipo relé e que pode assumir os valores $+1,5Nm$ ou $-1,5Nm$, ao ser aplicado à planta. Impõe-se que a posição do pêndulo siga um arco de senóide de um valor mínimo para um valor máximo (ou de um máximo para um mínimo) e que a velocidade siga, portanto, um arco de cossenóide assumindo valor nulo em t_i e em $t_f=t_i+\Delta t_s$. Entre t_i e t_f , os arcos de senóide e de cossenóide são repartidos em intervalos de $0,1s$ originando uma seqüência de valores desejados θ_{k+1}^d e ω_{k+1}^d que devem ser seguidos pela planta. Como um incremento em k ocorre a cada $0,02s$, consideraremos θ_{k+1}^d e ω_{k+1}^d constantes dentro de um intervalo de $0,1s$, embora k seja incrementado cinco vezes nesse mesmo intervalo.

Durante a etapa de treinamento, o **CMAC** deve assimilar a função dada na equação 7.3:

$$T_k = h(\theta_k, \omega_k, \alpha_{k+1}).\tag{7.3}$$



A equação 7.3 descreve a seguinte situação: "estando o pêndulo forçado na posição θ_k com velocidade ω_k , qual deve ser o torque aplicado no instante k para que a aceleração do pêndulo no instante $k+1$ seja α_{k+1} ". Assim, basta determinarmos o valor de α_{k+1} necessário para que o pêndulo atinja $(\theta_{k+1}^d, \omega_{k+1}^d)$, estando em (θ_k, ω_k) . Esta tarefa é realizada através da equação 7.4. Esta equação foi inspirada em [12] e é de natureza empírica.

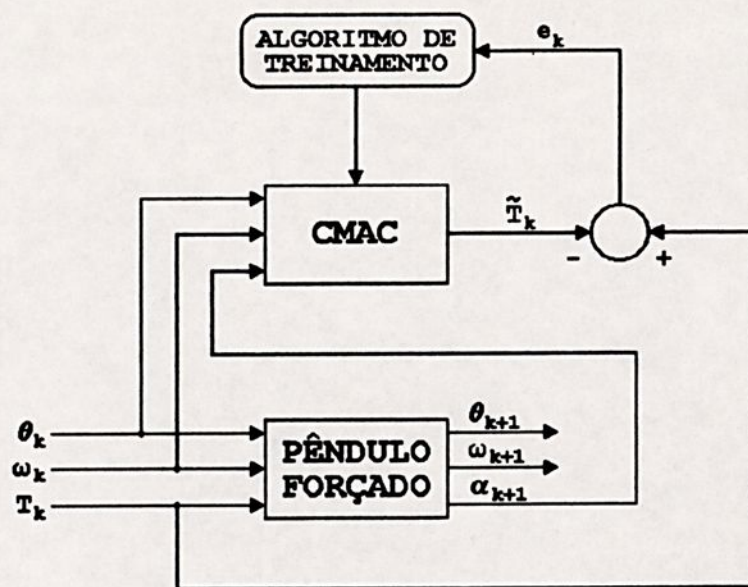
$$\alpha_{k+1}^d = H(\theta_k, \omega_k, \theta_{k+1}^d, \omega_{k+1}^d) = \frac{\alpha_{\max}}{2} \left(\frac{\theta_{k+1}^d - \theta_k}{\theta_{\max}} + 0,3 \frac{\omega_{k+1}^d - \omega_k}{\omega_{\max}} \right). \quad (7.4)$$

A figura 7.100 ilustra os diagramas de blocos do processo de treinamento do **CMAC** para que este reproduza a função $h(\cdot)$ e do processo de controle do pêndulo forçado para que este siga a trajetória definida a partir da informação assimilada pelo **CMAC**.

Os valores discretos dos pesos e da saída do **CMAC**, o valor da constante de generalização (C), da taxa de aprendizado (β , BETA), da memória requerida (em bytes), de θ_i , θ_f são dados na figura 7.101.

Os pesos são definidos por números representados com 3 bits. E ainda, os valores desejados para as saídas do **CMAC** são amostrados por um conversor A/D de 12 bits. As figuras 7.101 e 7.102 mostram os resultados obtidos com a simulação de um





(a)

FIGURA 7.100(a): Diagrama de blocos usado para o treinamento do **CMAC** a partir de entradas e saídas amostrados diretamente da planta.

pêndulo forçado com os seguintes parâmetros, supostamente invariantes no tempo:

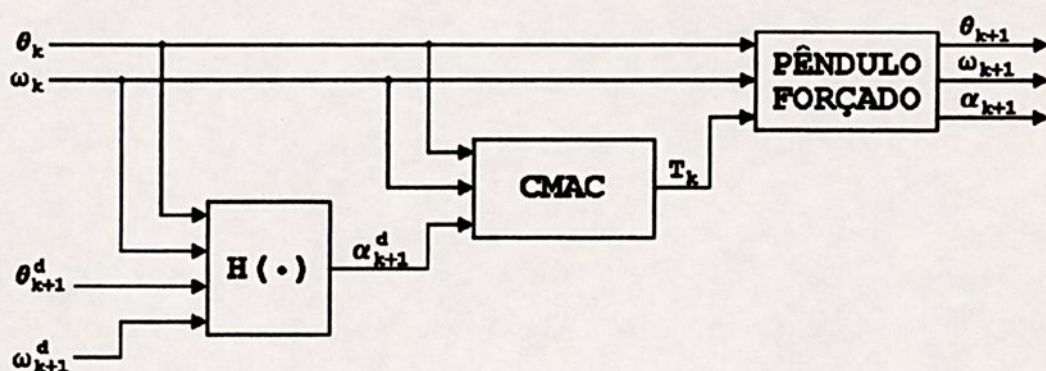
$$m = 0,1\text{kg};$$

$$l = 1\text{m};$$

$$g = 9,8\text{N/kg};$$

$$B = 0,03\text{Nm/(rad/s)}.$$

O treinamento do **CMAC** é feito conforme a seguinte sequência de passos:



(b)

FIGURA 7.100(b): Diagrama de blocos para o controle do pêndulo forçado a partir das excitações fornecidas pelo CMAC já treinado. O objetivo é que a planta siga uma sequência de valores de posição e de velocidade previamente especificados.

- posiciona-se o pêndulo, parado, numa posição arbitrária escolhida entre -270° e $+270^\circ$;
- aplica-se torques escolhidos aleatoriamente no conjunto $\{-1,5; +1,5\} [Nm]$, a cada $0,02s$ durante $1s$ ou até que ocorra $|\theta| \geq \theta_{\max} = 270^\circ$ ou $|\omega| \geq \omega_{\max} = 1500^\circ/s$;
- repete-se este procedimento por 100 vezes.

Ao final do treinamento, encontramos o seguinte

conjunto de pesos finais:

$\hat{w} = -1$, se $-270^\circ < \theta < 0$, $-1500^\circ/s < \omega < 0$ e $-5000^\circ/s^2 < \alpha < 0$,

$\hat{w} = +1$, se $-270^\circ < \theta < 0$, $-1500^\circ/s < \omega < 0$ e $0 \leq \alpha < +5000^\circ/s^2$,

$\hat{w} = -1$, se $-270^\circ < \theta < 0$, $0 \leq \omega < +1500^\circ/s$ e $-5000^\circ/s^2 < \alpha < 0$,

$\hat{w} = +1$, se $-270^\circ < \theta < 0$, $0 \leq \omega < +1500^\circ/s$ e $0 \leq \alpha < +5000^\circ/s^2$,

$\hat{w} = -1$, se $0 \leq \theta < +270^\circ$, $-1500^\circ/s < \omega < 0$ e $-5000^\circ/s^2 < \alpha < 0$,

$\hat{w} = +1$, se $0 \leq \theta < +270^\circ$, $-1500^\circ/s < \omega < 0$ e $0 \leq \alpha < +5000^\circ/s^2$,

$\hat{w} = -1$, se $0 \leq \theta < +270^\circ$, $0 \leq \omega < +1500^\circ/s$ e $-5000^\circ/s^2 < \alpha < 0$

$\hat{w} = +1$, se $0 \leq \theta < +270^\circ$, $0 \leq \omega < +1500^\circ/s$ e $0 \leq \alpha < +5000^\circ/s^2$.

A figura 7.103 ilustra o pêndulo forçado usado para a simulação. O sentido indicado pelas setas é o positivo, e os parâmetros listados anteriormente estão indicados.



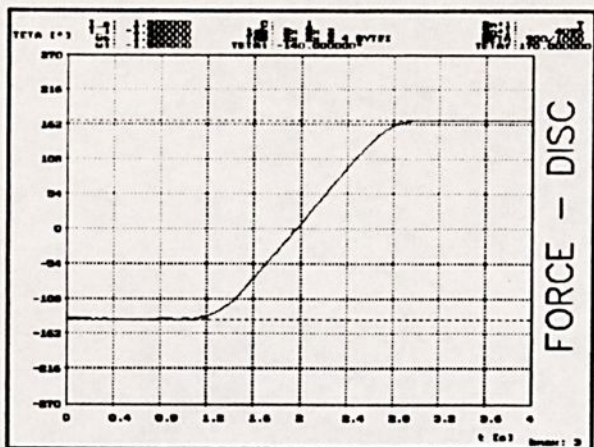


FIGURA 7.101: Comportamento da posição do pêndulo forçado durante a ação do controlador baseado no CMAC de Albus com pesos discretos.

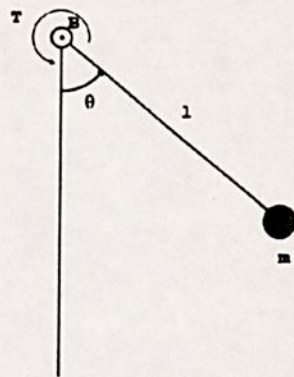


FIGURA 7.103: Diagrama esquemático do pêndulo forçado simulado.

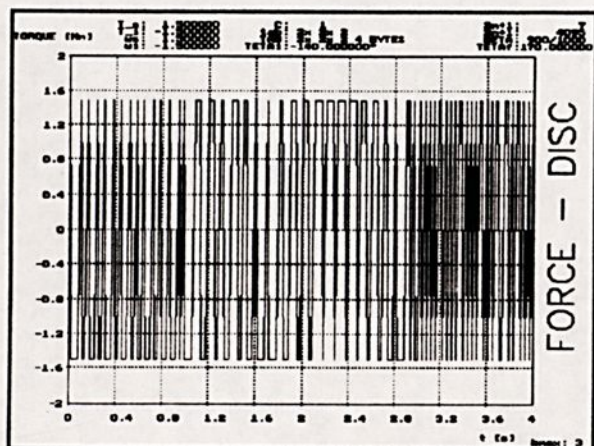


FIGURA 7.102: Comportamento do torque aplicado ao pêndulo forçado pelo CMAC, durante a simulação.

Capítulo 8

CONCLUSÕES

O **Cerebellar Model Articulation Controller** foi apresentado e analisado em suas vantagens e desvantagens. As versões criadas por **Albus** e por **Kraft & Campagna** mostraram-se eficientes na reprodução de funções analiticamente conhecidas. De forma geral, a configuração proposta por **Kraft & Campagna** oferece melhores resultados quando se define a mesma partição no conjunto de entrada da função a identificar, basta comparar as tabelas (7.1) com (7.13), (7.2) com (7.14) e sucessivamente. Isto é explicado pelo fato de que o **CMAC** de **Kraft & Campagna** usa uma memória física maior para representar a mesma informação. O **CMAC** proposto por **Albus** é mais adequado para as aplicações em que os pesos são representados com menor número de bits, já que possui um menor número de pesos e cada um, individualmente, é adaptado um maior número de vezes em relação aos pesos do **CMAC** de **Kraft & Campagna**.

Comparações entre o uso de código **HASH** ou de pesos discretos como dispositivos de redução de memória física requerida mostram que em **100%** dos casos de identificação de



funções abordados o uso de pesos discretos oferece um menor erro de representação apesar da maior lentidão decorrente da representação de valores, por **software**, com quantidade de bits que não sejam padrões definidos por compiladores ou hardware convencionais. Como a nova configuração proposta apresenta maiores tempos de estabilização do erro quadrático médio, recomenda-se o seu uso em identificação **off line** antes da aplicação efetiva em controle de sistemas (treinamento generalizado).

Percebemos que o **CMAC** possui muitas aplicações voltadas direta ou indiretamente para o controle de sistemas, mesmo que não se conheça métodos, além da própria experimentação, que auxiliem na escolha de parâmetros tais como: número de divisões por variável, taxa de generalização (C), quantidade de pesos necessários ou tamanho da representação de cada peso.

A estrutura que propomos pode ser aprimorada pois, necessita de um algoritmo específico para a busca de valores discretos mais próximos, na medida do possível, dos valores desejados.

Em resumo, consideramos que com esse trabalho um passo a mais foi dado a fim de aumentar a aplicabilidade do **CMAC**, por três motivos: (i) podemos obter resultados semelhantes aos que já haviam sido conseguidos, usando uma memória física similar;



(ii)pela descrição em português do **CMAC**, não disponível na literatura; (iii) finalmente por ser, provavelmente o primeiro estudo sobre o assunto no Brasil.

Convém colocar que este trabalho, mais especificamente o estudo apresentado no capítulo 6, **O CMAC COM PESOS DISCRETOS**, foi aceito e apresentado no **VI International Fuzzy Systems Association World Congress**, São Paulo 22-28 de julho de 1995[14].



Capítulo 9

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] **Widrow B. & Lehr M. A.**, "30 Years of Adaptive Neural Networks: Perceptron, Madaline and Backpropagation", Proceedings of IEEE, vol. 78, nº 9, september 1990.
- [2] **Lippmann, R. P.**, "An Introduction to Computing with Neural Nets", IEEE ASSP Magazine, april 1987.
- [3] **Hunt, K. J., Sbarbaro, D., Zbikowski, R. & Gawthrop, P. J.**, "Neural Networks for Control Systems - a Survey", Automatica, vol. 28, nº 6, pp 1083 - 1112, 1992.
- [4] **Psaltis, D., Sideris, A. & Yamamura A. A.**, "A Multilayered Neural Network Controller", IEEE Control Systems Magazine, pp 17 - 21, april 1988.
- [5] **Albus, J. S.** "A New Approach to Manipulator Control: the Cerebellar Model Articulation Controller (CMAC)", Trans. of the ASME, J. Dynamic Syst. Meas. Contr., vol. 97, pp 220-227, september 1975.



- [6] **Albus, J. S.** "Mechanisms of Planning and Problem Solving in the Brain", *Mathematical Biosciences*, pp 247 - 293, 1979.
- [7] **Yiu Fai Wong & Athanasios Sideris**, "Learning Convergence in the Cerebellar Model Articulation Controller", *IEEE Trans. on Neural Networks*, vol. 3, n° 1, january 1992.
- [8] **Kraft L. G. & Campagna D. P.**, "A Comparison between CMAC Neural Network Control and two Traditional Adaptive Control Systems", *IEEE Control Systems Magazine*, pp 36 - 43, april 1990.
- [9] **Handelman, D. A., Lane, S. H. & Gelfand, J. J.**, "Integrating Neural Networks and Knowledge Based Systems for Intelligent Robotic Control", *IEEE Control Systems Magazine*, pp 77 - 86, april 1990.
- [10] **Tolle, H. & Ersü**, "Neurocontrol", Springer Verlag, Heidelberg, 1992.
- [11] **Levin, A., U. & Narendra K., S.**, "Control of Nonlinear Neural Networks: Controllability and Stabilization", *IEEE Transactions on Neural Networks*, vol. 4, n° 2, march 1993.
- [12] **Saerens M., Soquet, A.** "Neural Controller Based on Backpropagation Algorithm", *IEE Proceedings-F*, vol. 138,



- [13] **Parks, P.C., Militzer, J.** "A Comparison of Five Algorithms for the Training of CMAC Memories for Learning Control Systems", *Automatica*, vol 28, n° 5, pp 1027-1035, 1992.
- [14] **Bordon, M.J., Teixeira, M.C.M.** "Improving Memory Requirement Using Cerebellar Model Articulation Controller with Discrete Weights", VI International Fuzzy Association World Congress, São Paulo, 22-28 de julho de 1995, pp 599-602.



DEPARTAMENTO DE ENGENHARIA ELÉTRICA
AV. BRASIL, 56 - CAIXA POSTAL 31
15378-000 ILHA SOLTEIRA - SP

