Universidade Federal do Amazonas
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Programa de Pós-Graduação em Informática

Metodologia para Classificação de Padrões de Consumo de Memória no Linux baseada em Mapas Auto-Organizáveis

Mauricio Tia Ni Gong Lin

Manaus – Amazonas
2006

Mauricio Tia Ni Gong Lin

Metodologia para Classificação de Padrões de Consumo de Memória no Linux baseada em Mapas Auto-Organizáveis

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Departamento de Ciência da Computação da Universidade Federal do Amazonas, como requisito para obtenção do Título de Mestre em Informática. Área de concentração: Inteligência Artificial.

Orientador: Edjard Mota

Mauricio Tia Ni Gong Lin

Metodologia para Classificação de Padrões de Consumo de Memória no Linux baseada em Mapas Auto-Organizáveis

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Departamento de Ciência da Computação da Universidade Federal do Amazonas, como requisito para obtenção do Título de Mestre em Informática. Área de concentração: Inteligência Artificial.

Banca Examinadora

Prof. Edjard Mota, Ph.D. – Orientador
Departamento de Ciência da Computação – UFAM/PPGI

Prof. Ilias Biris, Ph.D.
Departamento de Ciência da Computação – UFAM/PPGI

Prof. Dr. Raimundo Barreto
Departamento de Ciência da Computação – UFAM/PPGI

Manaus – Amazonas
2006

A minha família e namorada Helen.

Agradecimentos

Aos meus pais, pela educação proporcionada.

A minha namorada, Helen Nascimento, pela alegria, carinho e dedicação.

Ao meu orientador, Edjard Mota, pelo incentivo de ingressar no mestrado e confiança indiscutível no sucesso desta dissertação.

Ao meu chefe, Ilias Biris, pela paciência e apoio no desenvolvimento desta dissertação.

Ao meu mestre de Kung-Fu, Roberto Baptista, por ter me ensinado a importância dos estudos.

Aos colegas finlandeses: Fabritius Sampsa, Kimmo Hämäläinen e Juha Yrjölä, pela parceria de trabalho que influenciou no desenvolvimento desta dissertação.

Ao meu compadre, Ville Medeiros, pelas idéias e discussões realizadas na Finlândia.

Ao colega Daniel Petrini, pelas sugestões da dissertação.

Aos meus colegas de trabalho, pelo companheirismo e amizade.

Aos meus colegas de mestrado, pela ajuda oferecida nestes dois anos.

Ao amigo Beltimar Santos, por ter me emprestado o livro de redes neurais.

À Elienai Nogueira, pelo apoio administrativo.

Ao Linus Torvalds, por ter disponibilizado publicamente o código fonte do Linux que possibilitou a razão do tema desta dissertação.

A todas as pessoas que me ajudaram de alguma forma na conclusão deste trabalho, muito obrigado.

Conhecimento vem do seu instrutor, sabedoria vem do seu interior.

Bruce Lee

Resumo

A evolução do sistema operacional Linux possibilitou que o mesmo se tornasse o principal concorrente dos sistemas operacionais do mercado como o Windows da Microsoft e Solaris da Sun. Apesar de diversas funcionalidades e melhorias desenvolvidas no Linux, o problema relacionado à falta de memória e o mecanismo existente de solucioná-lo, chamado de OOM Killer, ainda é motivo de longas discussões na comunidade do kernel Linux.

A carência de pesquisas científicas relacionada ao algoritmo de seleção de processos do OOM Killer leva esta dissertação a propor um mecanismo de identificação e classificação de padrões de consumo de memória no Linux baseada no modelo de rede neural auto-organizável.

A ferramenta desenvolvida nesta dissertação mostra a possibilidade de utilizar Mapas Auto-Organizáveis para classificar e identificar os padrões de consumo de memória de determinadas aplicações inseridas em contextos de casos de uso.

Palavras-chave: Linux, Gerenciamento de Memória, Classificação de Padrões e Redes Neurais.

Abstract

The growth of Linux operating system has taken it to become a worthy competitor to commercial software such as Microsoft‘s Windows and Sun‘s Solaris. Although the development and the improvement of several Linux’s features, the problem related to Linux out of memory and the current mechanism used to solve it, named as OOM Killer, has brought a long discussion at Linux kernel community.

The lack of scientific works related to OOM Killer process selection algorithm motivates this dissertation to propose a mechanism for identifying and classifying memory consumption patterns of Linux applications. Such mechanism is based on a neural network technique known as Self Organizing Maps.

The development of a tool based on Self Organizing Maps presented the possibility of applying such approach for memory consumption patterns classification related to Linux applications use cases.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

A evolução do sistema operacional Linux, desde sua criação em 1991 por Linus Torvalds, permitiu que o mesmo se tornasse o principal concorrente dos sistemas Windows da Microsoft e Solaris da Sun em computadores servidores. Nos últimos anos, Linux tem sido considerado o sistema operacional de crescimento mais acelerado e conhecido em projetos de sistemas embarcados [1].

As últimas versões do *núcleo*, ou do inglês *kernel*, 2.6 do Linux incluem novas características que são excelentes para ambientes embarcados, como por exemplo: suporte a diversos modelos de memória e microcontroladores, melhoria de desempenho para aplicações de tempo real e aperfeiçoamento do sistema de entrada e saída (E/S) [2]. Requisições de memória são simplesmente rejeitadas quando não há memória disponível no sistema Operacional Unix, enquanto que no Linux uma solução diferente de tratar a *falta de memória (out of memory)* é utilizada. Embora novas melhorias são desenvolvidas para dispositivos embarcados, o problema relacionado à falta de memória no Linux e como solucioná-la através do *Out of Memory Killer (ou OOM Killer na forma abreviada)* ainda levanta longas discussões na comunidade do kernel Linux, conforme [3]. A falta de memória se torna mais acentuada em dispositivos com poucos recursos de processamento, como palmtops e telefones celulares [4].

A carência de trabalhos científicos relacionados com a falta de memória no Linux leva esta dissertação a propor um mecanismo de *identificar* e *classificar* o comportamento do consumo de memória. Modelos baseados em *redes neurais artificiais* podem prover uma forma de classificar os *padrões de consumo de memória* e verificar as aplicações responsáveis pelo consumo elevado de memória que ocasionam eventualmente a falta de memória no sistema.

1.1 Problema

A abordagem do OOM Killer foi desenvolvida por Rik Van Riel e melhorada por Andrea Arcangeli (ver Seção 2.1.8). Dentre as várias funções existentes no OOM Killer, a função que implementa o algoritmo de seleção de processos é considerada o elemento chave para determinar quais processos serão encerrados em uma situação de falta de memória.

O *algoritmo de seleção de processos* do OOM Killer, que escolhe quais processos devem ser encerrados quando há falta de memória, ainda pode ser melhorado. Dependendo da configuração do hardware e do conjunto de aplicações em execução, a heurística usada pelo atual OOM Killer pode não funcionar de maneira adequada em servidores e principalmente em dispositivos embarcados conforme o trabalho desenvolvido em [4].

O algoritmo de seleção foi desenvolvido de maneira empírica sem realizar antes uma análise ou estudo que possa proporcionar uma visão comportamental dos padrões de consumo de memória no Linux. O desenvolvimento de um mecanismo que forneça um modo de classificar padrões de consumo de memória é importante, pois assim novas abordagens de seleção no OOM Killer podem ser fundamentadas nos padrões classificados. A motivação da pesquisa deste trabalho é descrita a seguir.

1.2 Motivação

Apesar do esforço da comunidade do kernel Linux, as pesquisas científicas relacionadas ao problema do OOM Killer ainda estão em uma fase embrionária. A falta de trabalhos que relaciona modelos científicos com o algoritmo de seleção de processos do OOM Killer evidencia uma lacuna preocupante quando pensamos na possibilidade de adotar o Linux para sistemas com restrições de tempo ou aplicações críticas. Isto porque o funcionamento atual do OOM Killer não oferece uma seleção adequada dos processos. Sendo assim é importante propor uma implementação fundamentada em princípios computacionais e matemáticos.

A abordagem adotada para desenvolver este tipo de trabalho é utilizar um modelo de rede neural que possa fornecer um mecanismo para classificar o comportamento de consumo de memória de determinadas aplicações inseridas em alguns cenários de caso de uso. O modelo de rede neural auto-organizável pode fornecer resultados interessantes desde que o mesmo consiga mapear dezenas de milhares de dados de consumo de memória em um mapa topográfico.

A idéia do mapa topográfico consiste em mostrar a evolução do consumo de memória de uma aplicação durante um dado intervalo de tempo e assim identificar o comportamento

de consumo de memória da aplicação em um contexto de caso de uso. Vale ressaltar que os cenários de caso de uso devem simular situações reais para possibilitar uma análise e classificação realista dos dados coletados referentes ao consumo de memória.

1.3 Objetivos

O propósito desta dissertação consiste em propor uma metodologia baseada em redes neurais artificiais para classificar padrões de consumo de memória das aplicações do Linux. A idéia é utilizar a abordagem de redes neurais auto-organizáveis para fornecer um mapa topográfico da evolução do consumo de memória.

As informações de consumo de memória das aplicações devem ser coletadas para permitir o processo de classificação dos padrões de consumo de memória. A coleta dessas informações é baseada em cenários de casos de uso que visam simular situações reais no uso das aplicações. Assim os objetivos desta dissertação podem ser organizados como:

1. implementar uma rede neural auto-organizável que modele o comportamento do kernel para consumo de memória;
2. definir os cenários de casos de uso de consumo de memória;
3. automatizar a execução dos cenários definido no item 2;
4. coletar as informações do consumo de memória de cada cenário;
5. treinar a rede neural a partir das informações coletadas no item 4;
6. implementar uma ferramenta de classificação de padrões de consumo de memória baseado na rede neural implementada e treinada respectivamente nos itens 1 e 4.

Baseado nos objetivos citados anteriormente, esta dissertação pretende servir como referência para que outros algoritmos de seleção de processos de cunho mais científico possam ser construídos futuramente.

1.4 Estrutura da Dissertação

A estrutura dessa dissertação está organizada em 6 Capítulos. No Capítulo 2 são introduzidos alguns conceitos relacionados ao Linux, mostrando os seus subsistemas, destacando a parte de gerenciamento de memória e alguns trabalhos realizados no OOM Killer pela

comunidade do kernel. No Capítulo 3 são apresentados os conceitos de redes neurais artificiais, enfatizando o modelo de rede neural auto-organizável e o seu procedimento de aprendizagem.

No Capítulo 4 são discutidas as classes de padrões de consumo de memória e a maneira em que uma rede neural auto-organizável possa classificá-las. No Capítulo 5 são apresentados os resultados dessa dissertação, descrevendo os cenários de casos de uso projetados para prover os dados de treinamento da rede, alguns detalhes de implementação e do funcionamento da ferramenta que foi desenvolvida no decorrer desta dissertação. Finalmente no Capítulo 6 são apresentadas as conclusões deste trabalho e algumas sugestões de trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Este Capítulo tem o propósito de apresentar os estudos e as pesquisas realizadas que são relevantes para o desenvolvimento da proposta descrita na Seção 1.3. Inicialmente serão apresentados alguns conceitos de sistemas operacionais e algumas características inerentes ao sistema operacional Linux, principalmente as questões relacionadas ao comportamento de consumo de memória. Em seguida, uma abordagem de gerenciamento de memória para sistemas embarcados sem área de *paginação (ou do inglês swap)* é apresentada, mostrando os mecanismos utilizados e a estrutura da sua arquitetura.

2.1 Sistema Operacional Linux

Sistema operacional é um programa que atua como o intermediário entre o usuário e o hardware do computador, visando controlar e coordenar de maneira eficiente o uso dos recursos de hardware entre várias aplicações para diversos usuários [5, 6].

Nos sistemas operacionais modernos, o kernel reside em uma área protegida da memória e possui acesso total aos componentes de hardware. Este espaço da memória é chamado de espaço do kernel ou do inglês *kernel space*. Diferentemente, as aplicações de usuário são executadas no espaço do usuário ou *user space*. As aplicações enxergam apenas um subconjunto de recursos de hardware, pois no espaço do usuário o acesso às funcionalidades do hardware é restrito. Todo controle detalhado do hardware é de responsabilidade exclusiva do kernel [7].

Em sistemas Unix e similares, diversos processos concorrentes efetuam tarefas distintas. Cada processo consome recursos do sistema como bateria, memória, conectividade de redes ou algum outro tipo de recurso. As funções do kernel podem ser classificadas conforme descritas nas próximas Seções [8].

2.1.1 Gerenciamento de Processos

O kernel é encarregado de criar e destruir processos e de manipular as suas conexões com o mundo exterior (entrada e saída). Cada processo é um programa em execução [6]. Comunicação entre processos, através de sinais, pipes ou primitivas de comunicação de processos, é a base para a funcionalidade do sistema e também é tratado pelo kernel. Além disso, há o escalonador de processos que gerencia a utilização compartilhada do processador entre inúmeros processos. O escalonador de processos aloca o processador para o processo selecionado baseado em uma política de seleção, entre os diversos processos na memória que estão prontos para serem executados [5, 6].

2.1.2 Gerenciamento de Memória

A memória do computador é o principal recurso do sistema e a política utilizada para gerenciá-la é importante para o desempenho do sistema. O kernel gerencia os processos em um espaço de endereçamento virtual que utiliza o espaço disponível no disco rígido, ou outro tipo de memória secundária, como a extensão da memória física disponível. Quando o processador é solicitado para executar o processo, o espaço de endereçamento virtual do processo é mapeado para os endereços físicos correspondentes. O mapeamento entre o endereço virtual e físico é realizado por um dispositivo de hardware chamado de MMU (do inglês Memory Management Unit) [6]. O mapeamento entre o espaço de endereçamento virtual e físico é realizado através de tabelas de páginas.

2.1.3 Sistemas de Arquivo

Sistemas Unix e similares são baseados em conceitos de sistemas de arquivos, ou seja, quase tudo em Unix pode ser tratado como um arquivo. O kernel constrói um sistema de arquivos estruturados sobre dispositivos de hardware não estruturado e a abstração de arquivo resultante é usado extensivamente ao longo do sistema. Além disso, vários tipos de sistemas de arquivos são suportados pelo kernel do Linux como por exemplo o ext2, reiserfs, fat e etc.

2.1.4 Controle de Dispositivos

A maioria das operações de sistema interage eventualmente com um dispositivo físico. Com a exceção do processador, memória e algumas pouquíssimas entidades, qualquer operação de controle de dispositivos é efetuada através de códigos que são específicos do dispositivo manipulado. Tais códigos são chamados de driver de dispositivos. O kernel

deve possuir drivers de dispositivos para qualquer periférico presente no sistema para o seu funcionamento.

2.1.5 Redes

A comunicação de dados através da rede é gerenciada pelo sistema operacional, pois a maioria das operações em rede não são específicas de um processo. Os pacotes devem ser coletados, identificados e enviados antes de um processo tratá-lo. O sistema é encarregado de enviar os pacotes de dados através de programas e interfaces de rede e também controlar a execução dos programas conforme as suas atividades de redes. Além disso, questões de resolução de endereços e de roteamento são implementados também no kernel.

2.1.6 Sistema de Arquivos /proc

Ao invocar o comando `mount` no shell sem quaisquer argumentos, o shell mostrará os sistemas de arquivos atualmente montados no Linux. Uma das linhas impressas é similar como:

```
none on /proc type proc (rw)
```

O `/proc` é um sistema de arquivos especial. Observe que o primeiro campo, `none`, indica que este sistema de arquivos não está associado com nenhum dispositivo de hardware como o disco rígido. O `/proc` é uma janela do kernel que está em execução. Arquivos localizados no `/proc` não apresentam nenhuma correspondência com os arquivos reais de um dispositivo físico. Tais arquivos disponibilizam acesso aos parâmetros, estrutura de dados e estatísticas do kernel. Os conteúdos destes arquivos não são sempre blocos de dados fixos como os arquivos normais. O conteúdo de um arquivo do `/proc` é gerado pelo kernel quando uma leitura é feita. A escrita em um arquivo do `/proc` pode ser realizada para alterar a configuração do kernel em execução [9].

O exemplo abaixo lista os dados de um arquivo do `/proc`.

```
$ ls -l /proc/version
$ -r--r--r-- 1 root root 0 Feb 18 20:24 /proc/version
```

Observe que o tamanho do arquivo é zero, visto que o conteúdo do mesmo é gerado pelo kernel no momento do seu acesso. Portanto, o conceito de tamanho de arquivo não é aplicável para os arquivos do `/proc`. Além disso, a data e a hora de modificação

do arquivo sempre será igual à data e a hora atual do sistema. O conteúdo do arquivo `/proc/version` listado consiste basicamente de uma string que descreve a versão do kernel em execução. A leitura pode ser feita da mesma forma como se fosse ler um arquivo qualquer. Por exemplo, uma maneira trivial de exibir o conteúdo deste arquivo é através do comando `cat`.

```
$ cat /proc/version
Linux version 2.6.10 (mauricio@mzdhcp126145) (gcc version 3.3.5
(Debian 1:3.3.5-5)) #1 Thu Feb 17 17:15:11 AMT 2005
```

Da mesma forma, a escrita em um arquivo do `/proc`, pode ser realizada em qualquer outro arquivo normal.

```
$ echo 2 > /proc/sys/vm/overcommit_memory
```

O arquivo de `/proc` é comumente chamado de *entrada* (do inglês entry). A maioria das entradas do `/proc` é formatada legivelmente para o usuário e de um modo simples o suficiente para serem extraídas e lidas por aplicações. O `/proc` oferece informações variadas sobre o kernel, bem como os seus dispositivos e recursos. O comando `ps` e a ferramenta `top`, por exemplo, são aplicações que mostram informações dos processos através de leitura do `/proc`. Enfim, o `/proc` é um meio de comunicação entre o kernel e as aplicações do espaço de usuário, permitindo às aplicações mostrar o status do sistema através da leitura de um arquivo do /proc, assim como instruir o kernel de alterar o seu comportamento através da escrita em um arquivo do /proc.

2.1.7 Overcommit e Falta de Memória no Linux

Normalmente um programa do espaço de usuário, escrito na linguagem de programação C, reserva espaço na memória através da função `malloc()`. Se o valor do retorno for NULL, o programa identificará que não há mais memória disponível e efetuará ações apropriadas para tratar a exceção. Neste caso, a maioria dos programas imprime uma mensagem de erro e encerra a sua execução naquele instante, alguns programas precisam primeiramente liberar os arquivos bloqueados, e os programas mais inteligentes realizam uma coleta de lixo (garbage collection), limpando a memória, ou adaptam a sua execução à quantidade de memória disponível. Isso é como o Unix se comporta [10].

Linux não adota este padrão de comportamento com relação ao gerenciamento de

memória. O kernel aceita a maioria das solicitações de memória, na suposição de que as aplicações solicitam muita memória desnecessária. Em outras palavras, o kernel promete memória para as aplicações na esperança de que elas não vão precisar realmente de tanto recurso. Essa característica do Linux é conhecida como *overcommit*. Se essa suposição do kernel for satisfeita, então o sistema consegue executar mais programas na mesma memória ou executar um programa que exige muita memória virtual além do disponível. Caso a suposição não for satisfeita, logo situações desconfortáveis ocorrerão, pois o OOM Killer é disparado para encerrar a execução de algumas aplicações [11].

De acordo com a documentação do kernel [12], o kernel do Linux suporta atualmente três modos de tratamento de overcommit:

- No modo zero (0), o overcommit é baseado em uma heurística. Conforme a implementação dessa heurística, o kernel pode prometer ou não mais memória para as aplicações dependendo das condições de uso relativo á memória. Entretanto, na maioria dos casos, o kernel vai prometer mais memória do que a quantidade disponível. O modo zero é a configuração padrão do kernel.

- No modo um (1), o kernel sempre vai prometer mais memória, ou seja, nenhuma condição impedirá o kernel de recusar qualquer tipo de alocação de memória.

- No modo dois (2), o overcommit é desabilitado. Neste caso o kernel passa a aceitar alocações de memória que não exceda o tamanho do espaço de swap mais a porcentagem configurável (o padrão é 50%) da memória RAM. Por exemplo: se o tamanho do swap for 1024MB, a quantidade de memória RAM for de 512MB e a porcentagem configurável é o padrão, então alocações de memória não podem ser exceder (1024 + 256) MB. Neste modo o OOM Killer é desativado, pois o tamanho das alocações de memória nunca vai exceder o limite estabelecido. Entretanto as aplicações receberão mensagens de erro ao requisitar mais espaço de memória, aumentando a probabilidade das aplicações entrarem em um estado de falha.

Para definir ou alterar o modo de overcommit, basta informar o valor através do `/proc` como por exemplo:

```
$ echo 2 > /proc/sys/vm/overcommit_memory
```

Neste caso foi definido que o modo de overcommit é dois, ou seja, o overcommit é desativado. Para definir a porcentagem de memória quando o overcommit é desabilitado,

o valor é informado também através do `/proc`, como:

```
$ echo 80 > /proc/sys/vm/overcommit_ratio
```

No exemplo acima, foi definido que a porcentagem é de 80%.

2.1.8 Out of Memory Killer (OOM Killer)

Conforme mencionado na Seção 2.1.7, o *OOM Killer* é uma solução que existe no kernel para resolver o problema da falta de memória devido a sua característica de overcommit. Quando o sistema não tem memória disponível para alocar mais processos, a função `out_of_memory()` é chamada para tratar do problema. A Figura 2.1 ilustra, de maneira sucinta, o grafo de execução do out_of_memory(), segundo [11].

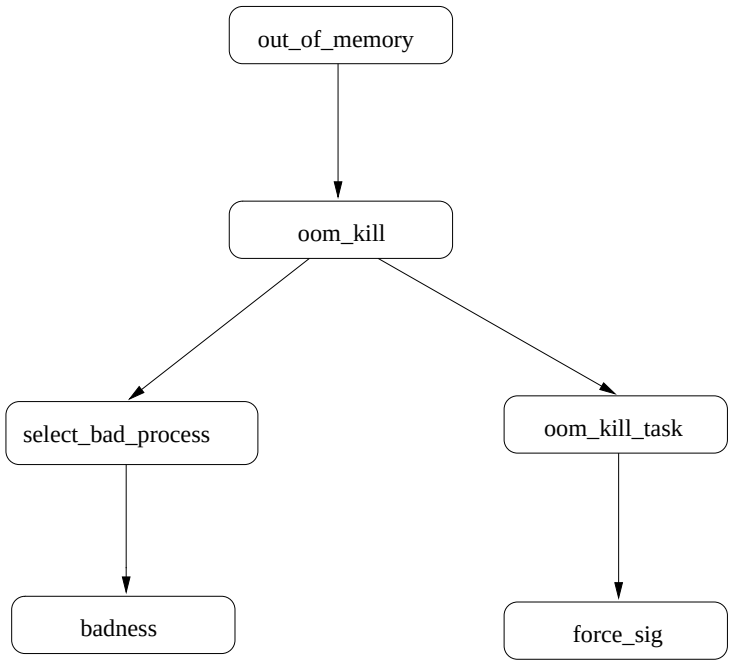

Figura 2.1: Grafo de chamadas esboçado de forma sucinta: out_of_memory().

As duas funções `select_bad_process()` e `oom_kill_task()` são responsáveis pela seleção e eliminação dos processos, respectivamente. A função `select_bad_process()` verifica as propriedades de cada processo e calcula uma pontuação que indica o nível de prioridade dele para ser encerrado. Quanto maior é a pontuação atribuída a um processo, maior probabilidade ele terá de ser finalizado. O cálculo dessa pontuação é baseado na Equação 2.1:

$$ponto\_do\_processo = \frac{memoria\_virtual\_do\_processo}{\sqrt{tempo\_de\_cpu} \times \sqrt[4]{tempo\_de\_execucao}} \quad (2.1)$$

Além disso, a função `select_bad_process()` verifica também outras variáveis, como a prioridade do processo e se o processo controla algum tipo de hardware, para efetuar o seu cálculo em sua plenitude. Uma vez selecionada o processo que será morto, a função `oom_kill_task()` é encarregada de enviar um sinal para matar o processo, bem como os threads relacionados a este processo.

A primeira versão do OOM Killer foi implementada por *Rik van Riel* em 1998 [13] e que persistiu até a última versão estável do kernel 2.6.10, ou seja, em 2005. Nessa implementação, antes de decidir matar os processos, é possível que o sistema não esteja no estado de falta de memória, pois o sistema pode estar simplesmente aguardando o término de uma operação de entrada e saída ou esperando a transferência das páginas da memória para a área de swap. Segundo [11], isso é realizado verificando as condições seguintes.

- Se existe espaço suficiente de swap (nr_swap_pages $>$ 0), então não há falta de memória.
- Se a duração desde a última falha for superior a 5 segundos, então não há falta de memória.
- Se houve falha dentro do último segundo, então há falta de memória.
- Se não ocorrer pelo menos 10 falhas nos últimos 5 segundos, então não há falta de memória.
- Se algum processo foi morto nos últimos 5 segundos, então não há falta de memória.

As condições citadas anteriormente são baseadas puramente em idéias empíricas. Quando o kernel identifica que não há meio de disponibilizar mais memória para os processos, a função `out_of_memory()` é invocada, no entanto antes de chamar a função `oom_kill()`, `out_of_memory()` verifica as condições que pode indicar a inexistência de falta de memória naquele momento, impedindo que `select_bad_process()` e `oom_kill_task()` sejam executados. Se o kernel anteriormente já identificou que não há mais memória disponível, então não teria razão para a função `out_of_memory()` também verificar se há ou não falta de memória. Basicamente, essas condições estranhas de temporizadores e contadores de falha são frutos de uma sincronização e programação concorrente que carecem de análise mais profunda e projeto mais bem elaborado.

Andrea Arcangeli e Thomas Gleixner modificaram o OOM Killer original [14], removendo o problema dos temporizadores e contadores de falha, solucionando o problema de sincronização que predominava no OOM Killer original [3]. Além disso, Arcangeli implementou uma interface que possibilita algoritmos de seleção de processos do OOM Killer no espaço de usuário interferirem no OOM Killer do kernel. Tabela 2.1 ilustra a comparação do OOM Killer original e do melhorado (novo) baseado no número de invocações do OOM Killer em cada experimento realizado. Observe que os números de invocações do OOM Killer modificado pelo Arcangeli e Gleixner são extremamente menores, quando comparado com o OOM Killer original.

Número de Invocações do OOM Killer	
Versão Original	*Versão Melhorada*
4547	20
1471	30
1624	22
2145	14
27990	2
3324	30
2382	30
3976	27
3430	3
731	32

Tabela 2.1: Comparação dos números de invocações da função `out_of_memory()`.

O problema de sincronização do OOM Killer original ocasionava inúmeras invocações desnecessárias da função `out_of_memory()`, por causa dos temporizadores e contadores de falha. Com a melhoria aplicada por Arcangeli e Gleixner, este overhead de invocações foi reduzido de modo significativo. No experimento comparativo entre o OOM Killer original e o do Arcangeli, a média do número de invocações da função `out_of_memory()` para 10 instâncias de testes, foi de 5162 invocações para o OOM Killer original e 21,3 invocações para o OOM Killer alterado pelo Arcangeli. Neste caso o número de invocações da função `out_of_memory()` reduziu aproximadamente 242 vezes, representando uma mudança importante do OOM Killer.

Apesar da melhoria proporcionada por Arcangeli e Gleixner, o funcionamento do OOM Killer continua sendo inadequado para sistemas de pequeno porte com pouco recurso de memória, pois a concepção do OOM Killer foi baseada nas necessidades existentes em computadores de mesa ou servidores, onde o tamanho da memória é consideravelmente grande quando comparado com dispositivos móveis como handhelds e telefones celulares.

A Seção seguinte descreve uma pesquisa desenvolvida que relaciona o mecanismo do OOM Killer com sistemas embarcados sem área de swap.

Além disso, a solução empírica desenvolvida por Rik van Riel evidencia uma necessidade de propor abordagens fundamentadas em princípios científicos que possam contribuir para o enriquecimento do algoritmo de seleção de processos do OOM Killer. O modelo de redes neurais apresentado na Seção 3.3 é usado para prover um novo mecanismo de classificação de processos baseado em padrões de consumo de memória, conforme descrito nos Capítulos 4 e 5.

2.2 Gerenciamento de Memória para Sistemas Embarcados sem Área de Swap

O conteúdo desta Seção consiste em um resumo do artigo publicado recentemente por [4] que apresenta uma estratégia de gerenciamento de alocação de memória para sistemas embarcados sem swap para evitar a latência acentuada do sistema e o disparo do OOM Killer. Este artigo apresenta os resultados parciais que fizeram parte da pesquisa desta dissertação.

2.2.1 Descrição da Abordagem de Gerenciamento de Memória

O OOM Killer do Linux não é invocado frequentemente em servidores e computadores de mesa, pois tais ambientes computacionais contêm memória física e espaço de swap suficiente para tornar a condição de falta de memória um evento raro de ocorrer. Entretanto sistemas embarcados sem área de swap tipicamente apresentam pouca memória principal, levando o OOM Killer a ser disparado com uma freqüência elevada.

Quando o sistema tem pouca memória livre disponível, as aplicações não funcionam apropriadamente devido à lentidão do sistema. O desempenho do sistema é afetada quando o uso da memória física está prestes a causar a condição de falta de memória ou quando toda a memória está sendo utilizada. Latências elevadas do sistema precisam ser reduzidas com o objetivo de evitar o desconforto no manuseio do sistema pelo usuário final.

Além disso, o algoritmo de seleção de processos do OOM Killer foi projetado para atender as necessidades de servidores e computadores de mesa, podendo não funcionar adequadamente em sistemas embarcados sem swap, pois a aplicação que está sendo visualizada e usada pelo usuário pode ser encerrada repentinamente.

A abordagem desenvolvida por [4], fornece dois mecanismos de gerenciamento de memória para sistemas embarcados sem área de swap. O primeiro mecanismo é aplicado para prevenir a latência acentuada do sistema em termos de uso de memória. Neste caso as alocações de memória são negadas e, portanto ocorre a falha de alocação de memória quando um limiar de consumo de memória pré-definida é alcançado. Este limiar é chamado de *Limite de Alocação de Memória* ou *MAT*, acrônimo de *Memory Allocation Threshold*.

O segundo mecanismo estabelece um limiar chamado *Limite de Sinalização* ou *ST*, acrônimo de *Signal Threshold*. Quando este limiar é alcançado, o kernel envia um sinal indicando pouca memória disponível ou *LMS (Low Memory Signal)* que deve ser recebido pelo espaço do usuário para liberar memória antes de atingir o MAT. Estes mecanismos são implementados no módulo do kernel, chamado de *Low Memory Watermark (ou LMW na forma abreviada)*, que tem os limites MAT e ST configuráveis através do sistema /proc.

Os recursos de memória devem ser gerenciados de modo diferente em dispositivos com restrição de memória para evitar a latência elevada do tempo de resposta do sistema. O mecanismo baseado no MAT é utilizado para evitar essa latência, pois as alocações de memória são negadas quando MAT é atingido e, portanto evitando a condição de falta de memória e o disparo do OOM Killer.

Antes de negar alocações de memória, processos podem ser encerrados para liberar memória. Isso é efetuado através da transmissão do LMS do espaço do kernel para o espaço do usuário, notificando as aplicações para liberarem memória. LMS é enviado de acordo com o valor do ST. O valor do ST deve ser menor que MAT, como ilustra a Figura 2.2, pois o LMS deve ocorrer antes da falha de alocação de memória.

Caso o envio do LMS seja efetuado com sucesso e a memória é liberada por receber o sinal, uma possível falha de alocação de memória é evitada. Um cenário interessante poderia envolver a sobreposição de janelas das aplicações A, B e C que estão em execução e consumindo memória gradativamente. Assumindo que a aplicação A é a que está sendo usada no momento pelo usuário, ao invés de negar alocações de memória para a referida aplicação A, seria mais viável tentar liberar a memória utilizada pelas aplicações B e C que não estão visíveis ao usuário, pois as janelas de B e C estão sobrepostas pela janela da aplicação A. Tal procedimento permite que o usuário continue interagindo com a aplicação A que está sendo usada atualmente.

Entretanto a falha de alocação de memória pode ser inevitável em algumas situações. Uma situação envolvendo o uso de uma única aplicação baseada em janela, que consume memória constantemente, pode ser um exemplo apropriado. Neste caso liberar a memória de outras aplicações baseadas em janelas não faz sentido, pois existe apenas uma aplicação

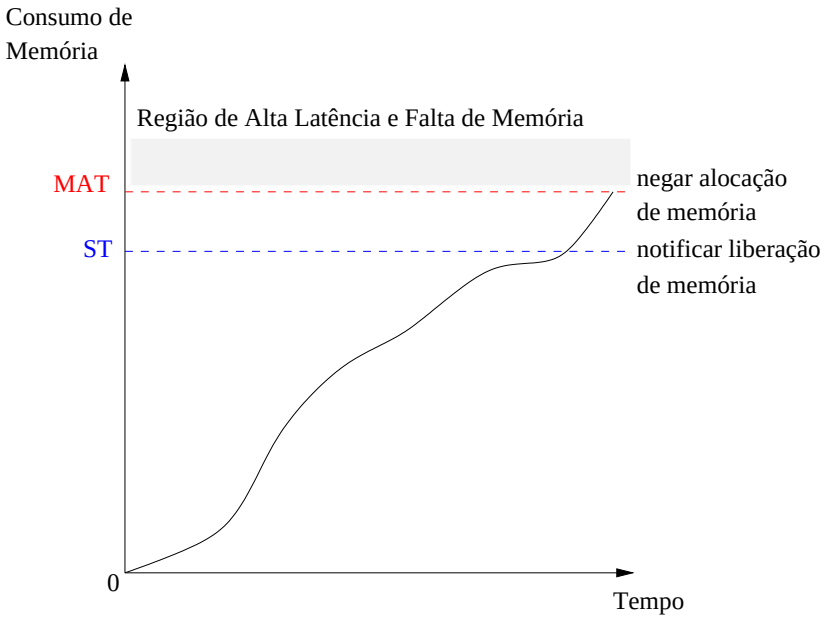

Figura 2.2: Limite de Sinalização é menor que o Limite de Alocação de Memória.

baseada em janela em execução. Logo a solução desejável seria negar as alocações de memória solicitadas pela referida aplicação.

Usando estes mecanismos, é possível identificar quando a memória alocada pode ser liberada ou a memória solicitada ser negada. Negar alocações de memória deve ocorrer somente quando a tentativa de liberar memória não pode ser procedida.

2.2.2 Arquitetura do LMW

Em termos de implementação, o LMW é um módulo de kernel baseado no framework chamado de *Linux Security Module (LSM)*. A implementação é baseada em uma heurística para verificar o limiar do consumo de memória física para negar alocações de memória ou notificar o espaço de usuário para liberar a memória.

O programa do espaço de usuário recebe a notificação para tomar medidas de liberar memória. Vale ressaltar que o módulo do LMW não inclui nenhuma implementação no espaço de usuário, visto que o mesmo proporciona apenas uma forma de notificar o espaço de usuário através da *Camada de Evento do Kernel (ou do inglês Kernel Event Layer)*, quando o limiar ST é atingido. Equações 2.2 e 2.3 representam respectivamente os limites calculados para o MAT e o ST:

$$MAT = physical\_memory \times deny\_percentage \tag{2.2}$$

$$ST = physical\_memory \times notify\_percentage \tag{2.3}$$

onde *physical_memory* é a memória principal do sistema.

Os valores do *deny_percentage* e *notify_percentage* são parâmetros configuráveis do kernel que podem ser ajustados através da interface *sysctl*. Tais parâmetros representam as porcentagens aplicadas na memória física para obter os valores do MAT e ST; e são associados ao sistema de arquivos /proc que podem ser escritos e lidos, usando os comandos echo e cat, como exemplificado a seguir.

```
$ echo 100 > /proc/sys/vm/lowmem_deny_watermark
$ echo 90 > /proc/sys/vm/lowmem_notify_watermark
$ cat /proc/sys/vm/lowmem_deny_watermark
100
$ cat /proc/sys/vm/lowmem_notify_watermark
90
```

Assumindo que o tamanho total da memória física do sistema é de 64 MB, o valor do MAT estabelecido é $64MB \times 100\% = 64MB$, enquanto que o valor do ST é aproximadamente $64MB \times 90\% = 57MB$. Neste caso a falha de alocações de memória acontece somente quando o sistema apresentar 64 MB de memória sendo consumida, enquanto que o sistema envia o sinal ST para o espaço do usuário quando o mesmo apresentar 57 MB de memória sendo consumida. Os ajustes aplicados no MAT e ST devem ser efetuados cuidadosamente, procurando sempre otimizar o uso da memória e simultaneamente evitar alocações excessivas que possam causar latências indesejadas ou o disparo do OOM Killer.

A arquitetura do LMW é ilustrada na Figura 2.3. Basicamente o LMW sobrepõe o comportamento padrão do overcommit (ver Seção 2.1.7), levando o sistema a gerenciar as alocações de memória utilizando a heurística representada pelas Equações 2.2 e 2.3. Observe que a aplicação do espaço de usuário que recebe a notificação do kernel é chamado de *OOM Killer Alternativo*, que consiste em um modo de liberar memória, aplicando outros mecanismos e técnicas para selecionar processos que serão encerrados.

Através desse mecanismo de notificação proporcionado pelo LMW, várias abordagens de seleção de processos podem ser experimentadas para o OOM Killer Alternativo como, por exemplo, utilizar o modelo de rede neural da Seção 3.3 como uma técnica de classificação de aplicações.

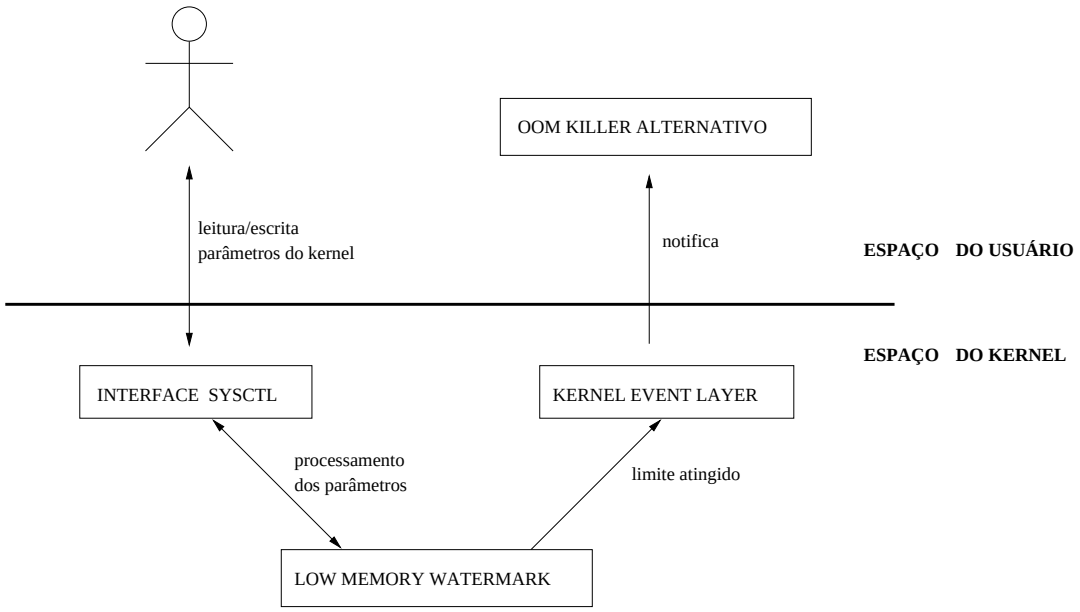

Figura 2.3: Arquitetura do LMW.

Sumário

Neste Capítulo foram descritos os conceitos do sistema operacional Linux, enfatizando o comportamento de gerenciamento de memória chamado de overcommit quando ocorre a falta de memória e a solução existente para tratá-la chamada de OOM Killer. Além disso, uma solução de falta de memória em sistemas embarcados sem swap é também discutida, visto que o OOM Killer não funciona adequadamente em tais sistemas.

No próximo Capítulo serão apresentados alguns fundamentos de redes neurais artificiais que são utilizados na etapa de implementação desta dissertação como parte dos objetivos descritos na Seção 1.3.

Capítulo 3

Redes Neurais Artificiais Auto-Organizáveis

Este Capítulo descreve os conceitos de redes neurais artificiais, mostrando alguns aspectos da aprendizagem ou treinamento supervisionado e não supervisionado. O modelo de rede neural auto-organizável é apresentado de modo detalhado, enfatizando as etapas do treinamento e as equações matemáticas envolvidas no processo de aprendizagem, bem como a introdução dos principais algoritmos relacionados.

3.1 Redes Neurais Artificiais

As *redes neurais artificiais* consistem em um método para solucionar problemas de inteligência artificial que são submetidas por um processo de aprendizagem com a finalidade de armazenar conhecimento experimental e torná-lo disponível para o uso.

Redes neurais artificiais, ou simplesmente redes neurais, é um grupo interconectado de neurônios artificiais que usa um modelo matemático ou computacional baseado no comportamento do cérebro humano para processamento de informações [15, 16, 17, 18].

A idéia é realizar o processamento de informações tendo como princípio a organização de neurônios do cérebro. Como o cérebro humano é capaz de aprender e tomar decisões baseadas na aprendizagem, as redes neurais artificiais devem fazer o mesmo. Assim, uma rede neural pode ser interpretada como um esquema de processamento capaz de armazenar conhecimento baseado em aprendizagem (experiência) e disponibilizar este conhecimento para a aplicação em questão [19].

A rede neural passa por um processo de treinamento, adquirindo a sistemática necessária para executar adequadamente o processo desejado dos dados fornecidos. Sendo assim, a rede neural é capaz de extrair regras básicas a partir de dados reais, diferindo

da computação programada, onde é necessário um conjunto de regras rígidas pré-fixadas e algoritmos [17].

Existem duas abordagens de treinamento, o supervisionado e o não supervisionado, que são descritas brevemente na próxima Seção.

3.2 Treinamento Supervisionado e Não Supervisionado

No treinamento *supervisionado*, os valores de *entrada* e da *saída desejada* são fornecidos para a rede. A rede processa então os valores de entrada e produz os valores de *saída real*. Um valor de erro é calculado baseado na comparação aplicada entre a resposta desejada fornecida para a rede e a resposta real produzida pela rede. O *valor de erro* é propagado de volta para a rede, ajustando os pesos sinápticos dos neurônios. Este procedimento é realizado inúmeras vezes até que os pesos ajustados da rede consigam prover uma resposta real satisfatória [17, 20].

As redes neurais mais conhecidas que utilizam o treinamento supervisionado são: perceptron de camada única alimentada adiante, perceptrons de múltiplas camadas alimentada adiante e redes recorrentes [17].

No treinamento *não supervisionado* ou *adaptativo*, somente os valores de *entrada* são fornecidos para a rede. A própria rede precisa decidir os critérios utilizados para agrupar e organizar os valores de entrada. Este procedimento geralmente é chamado de adaptação ou auto-organização. A rede possui habilidade de formar representações internas para codificar as características da entrada, criando automaticamente novas classes.

A rede neural mais conhecida que utiliza a aprendizagem não supervisionada é apresentada na Seção 3.3, que foi também implementada no desenvolvimento deste trabalho.

Independente do tipo de aprendizagem ou treinamento selecionado, toda rede é alimentada por uma seqüência de valores $x_1, x_2, \cdots, x_n$ na entrada e os pesos são ajustados de acordo com um modelo matemático durante a fase de treinamento. O processo de treinamento pode ser organizada nas etapas abaixo.

1. O primeiro padrão de entrada é apresentado para a rede.
2. Os pesos são ajustados para capacitar a rede de reconhecer o padrão fornecido.
3. O segundo padrão de entrada é apresentado para a rede e a etapa 2 é efetuada novamente.

4. O mesmo é aplicado para todos os outros padrões.

5. O procedimento de 1 até 4 é executada novamente centenas ou milhares de vezes até encontrar uma configuração de pesos sinápticos capaz de reconhecer todos os padrões fornecidos no treinamento.

A Seção seguinte ilustra o treinamento de uma rede auto-organizável de uma forma mais detalhada, assim como as características inerentes à este modelo de rede neural.

3.3 Mapas Auto-Organizáveis

Mapas Auto-Organizáveis ou simplesmente *SOM (do inglês Self Organizing Maps)* é uma rede neural artificial auto-organizável, de aprendizagem não supervisionada, baseada em grades de neurônios artificais onde os pesos são adaptados em conformidade com os vetores de entrada fornecidos durante o treinamento. Foi desenvolvido pelo professor Teuvo Kohonen e as vezes é chamado de mapa de Kohonen [21, 22, 23, 24].

No SOM, os neurônios estão colocados em nós de uma grade unidimensional ou bidimensional. Mapas de dimensões mais elevadas são também possíveis, embora não sejam tão comuns, pois o SOM tem o propósito de representar dados multidimensionais em espaços de dimensões menores [17, 21, 22]. Os neurônios são seletivamente sintonizados a vários padrões de entrada ou classes de padrões de entrada no decorrer de um processo de aprendizagem ou treinamento.

As localizações dos neurônios assim sintonizados se tornam ordenadas entre si de modo que um sistema de coordenadas significativos para características diferentes de entrada é criado sobre a grade. Portanto um mapa auto-organizável é caracterizado pela formação de uma mapa topográfico dos padrões de entrada no qual as localizações espaciais, i.e., coordenadas dos neurônios na grade, são indicativas das características estatísticas intrínsecas contidas nos padrões de entrada, por isso o nome *mapa auto-organizável* [17].

Um exemplo comum usado para mostrar os princípios de SOM é o mapeamento de cores a partir de seus 3 componentes: vermelho, verde e azul ou RGB (red, green and blue) em um espaço bidimensional. A Figura 3.1 ilustra um exemplo de SOM treinado para reconhecer padrões de cores. As cores são fornecidas para a rede como vetores de 3 dimensões, uma dimensão para cada componente de cor, e a rede foi treinada para representá-los em um espaço de 2 dimensões. Observe que além do agrupamento de cores em regiões distintas, as regiões de propriedades similares estão localizadas de forma adjacente.

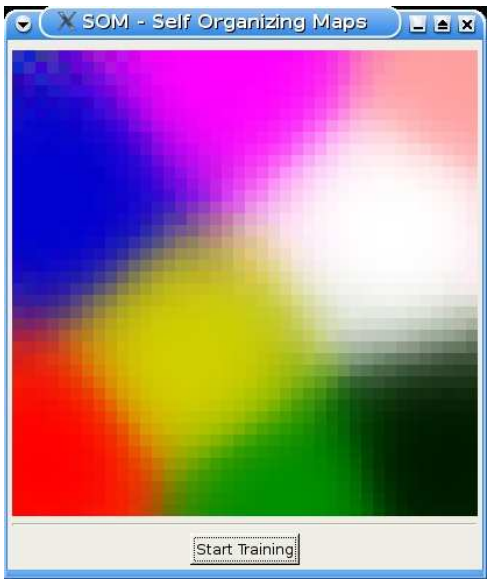

Figura 3.1: Um exemplo de SOM treinado para classificação de cores.

Com o propósito de descrever a arquitetura da rede, um exemplo bidimensional de SOM é apresentado a seguir. A rede é criada a partir de uma grade bidimensional de nós, onde cada nó que representa um certo neurônio é conectado com a camada de entrada. Figura 3.2 mostra um exemplo simples de rede Kohonen de nós 4×4 conectados com a camada de entrada, representando um vetor bidimensional.

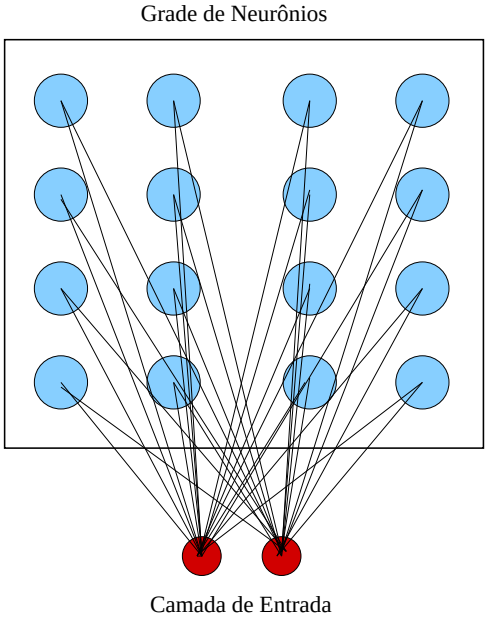

Figura 3.2: Uma rede simples de Mapas Auto-Organizáveis de Kohonen.

Cada nó da grade tem uma posição topológica específica representada por uma coordenada (x, y) e contém um vetor de pesos sinápticos de dimensão igual ao vetor de entrada. Ou seja, caso os dados de treinamento consistem de vetores x de dimensão n:

$$x = [x_1, x_2, x_3, ..., x_n]$$

Então cada nó ou neurônio j da grade contém um vetor correspondente de peso w_j com a mesma dimensão n:

$$w_j = [w_{j1}, w_{j2}, w_{j3}, ..., w_{jn}]$$

A rede SOM ilustrada na Figura 3.1 apresenta uma grade de tamanho 40×40. Cada nó da grade possui 3 pesos, cada um representando um componente RBG. Além disso, cada nó é representado por uma célula retangular, quando desenhado na interface gráfica do aplicativo.

O algoritmo responsável pela formação do SOM primeiramente começa inicializando os pesos sinápticos da grade. Isto pode ser feito atribuindo-lhes valores pequenos obtidos de um gerador de números aleatórios. Realizando dessa maneira, nenhuma organização prévia é imposta ao mapa de características. Uma vez que a grade tenha sido apropriadamente inicializada, há três processos importantes envolvidos na constituição do SOM, como descrito brevemente aqui [17].

- *Competição:* Para cada padrão de entrada fornecida, os neurônios da grade calculam seus respectivos valores baseada em uma função discriminante. Esta função proporciona a base para a competição entre os neurônios. O neurônio com o maior valor da função discriminante é considerado como vencedor da competição.

- *Cooperação:* O neurônio vencedor estabelece a localização espacial de uma vizinhança topológica de neurônios ativos, provendo dessa forma o meio para a cooperação entre os neurônios vizinhos.

- *Adaptação Sináptica:* Permite que os neurônios excitados aumentem seus valores individuais da função discriminante em relação ao padrão de entrada através de ajustes feitos nos respectivos pesos sinápticos.

O procedimento de treinamento é constituído de várias etapas que são repetidas várias vezes, como apresentado a seguir.

Inicialização: Cada neurônio tem os seus pesos sinápticos inicializados aleatoriamente. Geralmente os pesos são inicializados entre 0 e 1, ou seja, $0 < w < 1$.

Seleção de Vetor de Entrada: Um vetor de entrada é escolhido do conjunto de dados de treinamento e apresentado para a grade de neurônios.

Competição de Neurônios: Todos os pesos de todos os neurônios são calculados para determinar o neurônio mais semelhante em relação vetor de entrada. Este procedimento de comparação é chamado de casamento por similaridade, visto que o neurônio mais similar em relação ao vetor de entrada é selecionado. O neurônio selecionado é considerado como o neurônio vencedor e chamado de *Unidade de Melhor Casamento ou BMU (do inglês Best Matching Unit).*

Cooperação de Neurônios: O *raio da vizinhança topológica* do BMU é calculado. O valor do raio assume inicialmente um valor elevado, geralmente tem o mesmo o raio da grade, mas diminui a cada iteração do treinamento. Os neurônios localizados dentro deste raio são considerados os vizinhos do BMU.

Adaptação Sináptica: Os pesos sinápticos de cada neurônio vizinho do BMU são ajustados para torná-los similares ao vetor de entrada. Os neurônios vizinhos mais próximos do BMU têm os seus pesos alterados de modo mais significativo.

Repetição: O segundo passo é retomado novamente selecionando um novo vetor de entrada e os passos subseqüentes são então executados.

O detalhamento do algoritmo de SOM é descrito a seguir, conforme [22, 17, 25].

Antes de realizar o treinamento do SOM, os pesos sinápticos de cada neurônio da grade precisam ser inicializados como apresentados no Algoritmo 3.1. A grade de neurônios é representada por uma matriz bidimensional nomeada de `grids`. A função `rand_float()` gera um valor aleatório no intervalo de 0 e 1 para o vetor peso de cada neurônio.

A forma para determinar o BMU é acessar todos os neurônios da grade e calcular a *distância Euclidiana* entre o vetor peso de cada neurônio e o vetor de entrada atual, como mostra o Algoritmo 3.2. O neurônio de vetor peso mais próximo do vetor de entrada, ou seja, menor distância Euclidiana, é considerado como o neurônio vencedor ou BMU. Neste caso a distância Euclidiana é a função discriminante neste processo competitivo de neurônios e calculado como ilustra a Equação 3.1:

$$dist_j = \sqrt{\sum_{i=1}^{n}(x_i - w_i)^2} \tag{3.1}$$

que é matematicamente equivalente a Equação 3.2:

Algoritmo 3.1 Inicialização dos pesos sinápticos dos neurônios da grade.

```
void init_grid(struct som_node * grids[][GRIDS_YSIZE]) {
    int i, j, k;
    srand(time(0));
    for (i=0; i<GRIDS_XSIZE; i++) {
        for (j=0; j<GRIDS_YSIZE; j++) {
            grids[i][j] = (struct som_node *)
                    malloc(sizeof(struct som_node));
            grids[i][j]->xp = i;
            grids[i][j]->yp = j;
            for (k=0; k<WEIGHTS_SIZE; k++) {
                grids[i][j]->weights[k] = rand_float();
            }
        }
    }
}
```

Algoritmo 3.2 Pesquisa o neurônio vencedor (BMU) durante o processo competitivo.

```
struct som_node * get_bmu(double input_vector[],
                      int len_input,
                      struct som_node * grids[][GRIDS_YSIZE]
                      ) {
    struct som_node *bmu = grids[0][0];
    double best_dist = euclidean_dist(input_vector,
                                      bmu->weights,
                                      len_input);
    double new_dist;
    int i, j;
    for (i=0; i<GRIDS_XSIZE; i++) {
        for (j=0; j<GRIDS_YSIZE; j++) {
            new_dist = euclidean_dist(input_vector,
                                      grids[i][j]->weights,
                                      len_input);
            if (new_dist < best_dist) {
                bmu = grids[i][j];
                best_dist = new_dist;
            }
        }
    }
    return bmu;
}
```

$$dist_j^2 = \sum_{i=1}^{n}(x_i - w_i)^2 \tag{3.2}$$

onde $dist_j$ é a distância entre o vetor peso w do neurônio j e o vetor de entrada x de dimensionalidade n.

Em termos implementacionais, a Equação 3.2 é mais eficiente, visto que a função raiz quadrada não é utilizada na codificação. Além disso, o valor $dist_j^2$ é sempre utilizado durante o treinamento, portanto é preferível manter o valor da distância na sua forma quadrática. Algoritmo 3.3 mostra a implementação da Equação 3.2.

Algoritmo 3.3 Cálculo da distância Euclidiana.

```
double euclidean_dist(double *input, double *weights, int len) {
    double summation = 0;
    double temp;
    int i;
    for (i=0; i<len; i++) {
        temp = (input[i]-weights[i]) * (input[i]-weights[i]);
        summation += temp;
    }
    return summation;
}
```

A cada iteração do treinamento, após o BMU ter sido determinado, o passo seguinte consiste em calcular quais são os neurônios localizados na vizinhança topológica do BMU. Tais neurônios terão os seus pesos sinápticos alterados posteriormente durante o processo de adaptação sináptica. O raio da vizinhança topológica é calculado e depois uma comparação é feita em cada neurônio da grade para verificar se o mesmo está localizado ou não na vizinhança topológica atual. A Figura 3.3 ilustra um exemplo de vizinhança topológica determinada no início do treinamento da rede. O BMU é representado pela cor vermelha e a vizinhança topológica, representada pela área sombreada, está centralizada em torno do BMU encontrado.

Uma característica da rede SOM é que a área da vizinhança topológica diminui ao longo do treinamento da rede. A cada iteração t, a função de decaimento exponencial é calculada para permitir o reajuste do tamanho da vizinhança do BMU como mostrada na Equação 3.3:

$$\sigma(t) = \sigma_0 e^{(-\frac{t}{\lambda})} \tag{3.3}$$

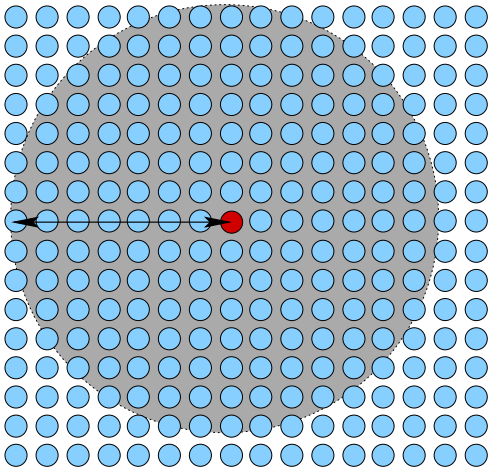

Figura 3.3: Vizinhança topológica de um BMU representada pela área sombreada.

onde σ_0 representa o raio σ no tempo t_0 que tem o mesmo valor do raio da grade, e λ é uma constante de tempo. A variável t representa um dado instante do treinamento, ou melhor, um passo de tempo da iteração. A Figura 3.4 ilustra o modo como a vizinhança topológica diminui durante o treinamento da rede, assumindo que a vizinhança permanece centralizada em torno do mesmo neurônio. Na prática o BMU pode ser qualquer neurônio da grade, conforme o vetor de entrada que está sendo apresentado para a rede. A implementação da função de decaimento exponencial é apresentada no Algoritmo 3.4.

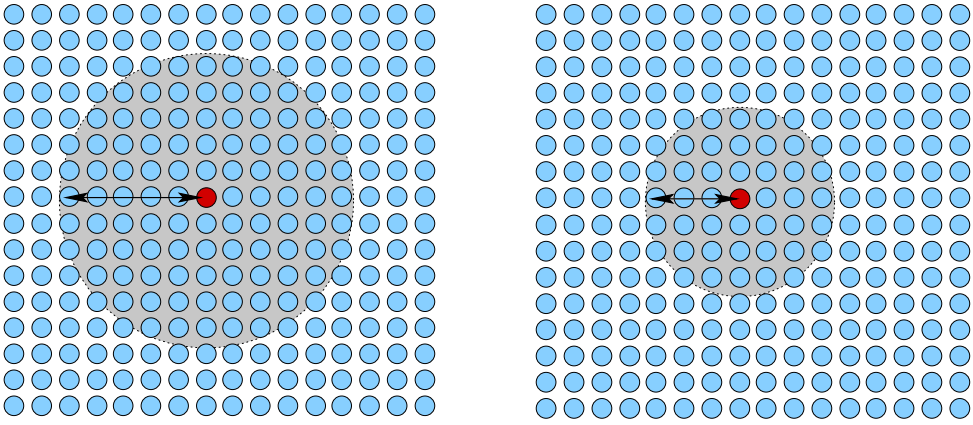

Figura 3.4: O tamanho da vizinhança topológica diminui no decorrer do treinamento.

Para que a grade seja auto-organizável, é necessário que o vetor de peso sináptico w_j do neurônio j da grade se modifique em relação ao vetor de entrada. Todos os neurônios localizados na vizinhança topológica, incluindo o próprio BMU, têm os seus pesos sinápticos alterados de acordo com a Equação 3.4:

Algoritmo 3.4 Função de decaimento exponencial utilizada para o cálculo da vizinhança topológica.

```
double neighborhood_radius(double init_radius,
                           double iteration,
                           double time_constant) {
    return init_radius * exp(-iteration/time_constant);
}
```

$$W_j(t+1) = W_j(t) + \eta(t)\Theta_j(t)(V(t) - W_j(t)) \tag{3.4}$$

onde t representa o instante atual, $\eta(t)$ é um *parâmetro da taxa de aprendizagem* variável no tempo que decresce gradualmente com o aumento do tempo t e $\Theta_j(t)$ é uma função variável no tempo que determina o *grau de aprendizagem* do neurônio j baseada na distância relativa com o BMU. O procedimento de adaptação ou ajuste sináptico é demonstrado no Algoritmo 3.5.

Algoritmo 3.5 Ajuste dos pesos sinápticos de cada neurônio localizado na vizinhança topológica.

```
void adjust_weights(struct som_node * node,
                    struct input_pattern * input,
                    int len_input,
                    double learning_rate,
                    double influence) {
    double node_weight, input_weight;
    int i;
    for (i=0; i<len_input; i++) {
        node_weight = node->weights[i];
        input_weight = input->weights[i];
        node_weight += influence * learning_rate *
                (input_weight-node_weight);
        node->weights[i] = node_weight;
    }
}
```

Analogamente, a Equação 3.5 que define o parâmetro da taxa de aprendizagem é uma função de decaimento exponencial idêntica a Equação 3.3 para cálculo do raio da vizinhança. A taxa de aprendizagem inicial $\eta(0)$ é definido em 0.1 e diminui ao longo do tempo, atingindo um valor próximo de zero nas últimas iterações do treinamento.

$$\eta(t) = \eta_0 e^{(-\frac{t}{\lambda})} \tag{3.5}$$

A taxa de aprendizagem é implementada pelo Algoritmo 3.6.

`START_LEARNING_RATE` e `NUM_ITERATIONS` são macros que correspondem respectivamente a taxa de aprendizagem inicial e número de iterações do treinamento.

Algoritmo 3.6 Cálculo da taxa de aprendizagem.

```
learning_rate = START_LEARNING_RATE *
                exp(-(double)iteration/NUM_ITERATIONS);
```

Além da influência do parâmetro da taxa de aprendizagem no ajuste dos pesos sinápticos, a adaptação sináptica é também influenciada pela proporção da distância entre o neurônio j e o BMU. Isso é representado pela Equação 3.6:

$$\Theta_j(t) = e^{(-\frac{dist_j^2}{2\sigma^2(t)})} \tag{3.6}$$

onde $dist_j$ corresponde a distância Euclidiana entre o neurônio j e o BMU, e $\sigma(t)$ é o raio obtido pelo cálculo da vizinhança topológica, como apresentado na Equação 3.3. Observe que $\Theta_j(t)$ decresce gradualmente ao longo do tempo. A implementação da Equação 3.6 é mostrada no Algoritmo 3.7. O parâmetro `distance` da função é instanciado como a distância Euclidiana na sua forma quadrática, diferente do parâmetro `radius` que precisa ainda ser calculado.

Algoritmo 3.7 Cálculo da quantidade de influência de um neurônio na sua adaptação sináptica.

```
double get_influence(double distance, double radius) {
    double radius_sq = radius * radius;
    return exp(-(distance)/(2 * radius_sq));
}
```

De acordo com [22], as redes SOM são aplicadas em várias áreas tais como: recuperação de informação em base de dados bibliográficos [26], sistema de classificação de imagens [27, 28], diagnóstico médico, modelagem ambiental, compressão de dados e reconhecimento de fala. Seções 5.2 e 5.3 apresentam como a rede SOM pode ser utilizada para classificar padrões de consumo de memória de aplicações em um sistema Linux.

Sumário

Neste Capítulo foram apresentados alguns conceitos de redes neurais artificiais, enfatizando o modelo de rede neural auto-organizável chamado de Mapas Auto-Organizáveis. A abordagem baseada em Mapas Auto-Organizáveis foi descrita neste Capítulo de maneira detalhada, pois é o modelo de rede neural escolhido para classificar padrões de consumo de memória nesta dissertação.

No Capítulo seguinte serão discutidos os padrões de consumo de memória em termos de tamanho e ritmo de alocação e liberação de memória, bem como os tipos de dados que possibilitam a rede neural auto-organizável realizar a classificação dos padrões.

Capítulo 4

Classificação de Padrões de Consumo de Memória

A maneira como os processos alocam memória pode ser analisada conforme a quantidade de células ou blocos alocados e liberados. É difícil prever a quantidade de memória consumida por uma dada aplicação, pois não há um comportamento padrão de solicitação do uso de memória.

Este Capítulo apresenta uma maneira de classificar o consumo de memória de acordo com o tamanho das células que estão sendo alocadas e liberadas, assim como a freqüência ou ritmo que isso acontece. O propósito de mostrar as classes de consumo de memória é encontrar um modelo que forneça uma representação comportamental da evolução do consumo de memória de uma determinada aplicação. Seção 4.1 descreve os padrões de consumo de memória conforme [29] e a Seção 4.2 apresenta uma representação das classes de padrões de consumo de memória baseada em um modelo de rede neural auto-organizável.

4.1 Padrões de Consumo de Memória

Não existe uma definição absoluta em termos de quantidade de memória alocada ou liberada, pois afirmar que o sistema tem pouca ou muita memória disponível são medidas qualitativas que dependem do hardware que está sendo usado e dos tipos de aplicações que estão em execução. Entretanto, uma representação abstrata de padrões de alocação e liberação de memória pode ser definida independentemente das restrições de hardware, como apresentada a seguir e ilustrada na Figura 4.1.

1. Células pequenas de memória: cada alocação ou liberação de memória corresponde a uma célula ou bloco pequeno de memória. Por exemplo: assumindo que 4 KB é o

tamanho de uma célula pequena, um programa que só aloca ou libera 4 KB várias vezes pertence a este tipo de comportamento.

2. Células grandes de memória: cada alocação ou liberação de memória corresponde a uma célula ou bloco grande de memória. Por exemplo: assumindo que 16 KB é o tamanho de uma célula grande, um programa que só aloca ou libera 16 KB várias vezes pertence a este tipo de comportamento.

3. Células pequenas e grandes de memória organizadas de forma intercalada: a alocação ou liberação de memória é alternadamente pequena e grande. Por exemplo: assumindo que 4 KB e 16 KB são respectivamente o tamanho de uma célula pequena e grande, um programa que aloca ou libera 4KB e depois 16 KB sempre de forma intercalada pertence a este tipo de comportamento.

4. Seqüências alternadas de células pequenas seguidas de células grandes de memória, onde i, j e k representam o intervalo da quantidade de células para cada seqüência.

5. Seqüências alternadas de células grandes seguidas de células pequenas de memória, onde i, j e k representam o intervalo da quantidade de células para cada seqüência.

6. Seqüências aleatórias de células pequenas e grandes de memória. Neste caso a alocação ou liberação de memória tem um comportamento imprevisível.

Além disso, cada padrão mencionado pode apresentar uma freqüência diferente de consumo de memória, ou seja, um ritmo variado em que a alocação e a liberação de memória se procedem. Suponha que as aplicações A e B pertencem ao comportamento de consumo de memória do item 1, portanto ambos alocam células pequenas de memória de mesmo tamanho. No entanto a aplicação A efetua a alocação de memória a cada 1s enquanto que a aplicação B efetua a alocação a cada 2s. Após o tempo decorrido de 10s, certamente o consumo de memória da aplicação A é maior que B, assumindo que neste intervalo nenhumas das aplicações tiveram a memória liberada.

Portanto a quantidade de memória consumida e o ritmo em que o consumo diminui ou aumenta, são variáveis que indicam a evolução e o comportamento do consumo de memória e interessantes de serem abordadas em um modelo de representação de padrões de consumo de memória. Visto que o padrão de consumo de memória pode assumir formas variáveis e aleatórias em termos de tamanho de células alocadas e ritmo de alocação, um modelo baseado em redes neurais auto-organizáveis pode ser capaz de mapear e classificar as diversas combinações possíveis de padrão de consumo de memória, que é apresentado na próxima Seção.

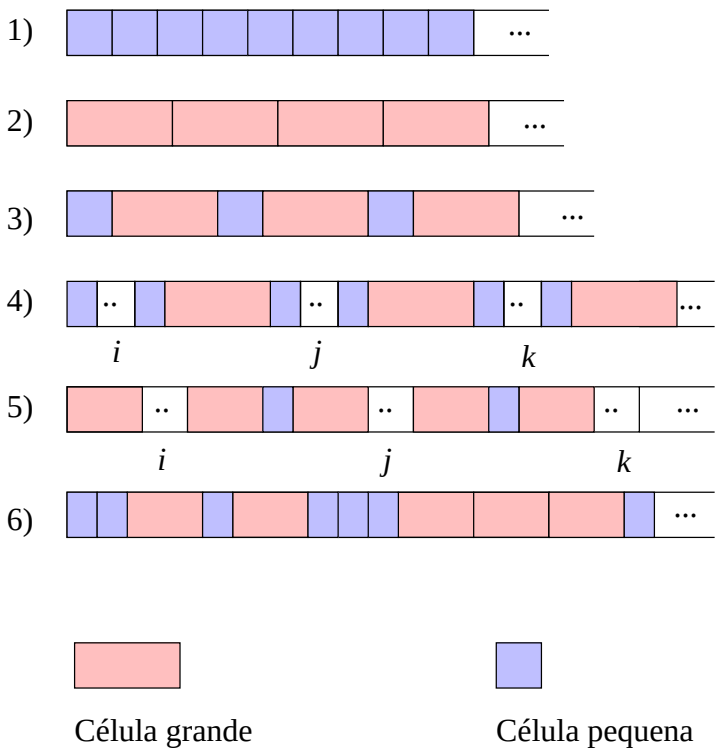

Figura 4.1: Representação abstrata de padrões de consumo de memória em termos de tamanhos de células.

4.2 Classificação de Padrões de Consumo de Memória baseado em Redes Neurais Auto-Organizáveis

Conforme os padrões de consumo de memória apresentada na Seção 4.1, a forma encontrada para classificar os padrões de consumo de memória consiste em utilizar os seguintes dados:

- quantidade de páginas físicas alocadas que representa a *quantidade de memória física consumida*;
- *variação do consumo de memória (VCM)* que é utilizada para indicar o ritmo em que o consumo de memória está aumentando ou diminuindo. Esta variação é calculada através da razão entre a diferença da quantidade de memória física consumida e o intervalo de tempo decorrido, conforme a Equação 4.1;

$$VCM = \frac{mem_2 - mem_1}{t_2 - t_1} \tag{4.1}$$

- *taxa de variação do consumo de memória (TVCM)* que é utilizada para indicar o ritmo em que a variação de consumo de memória está aumentando ou diminuindo. Esta taxa é calculada através da razão entre a diferença da variação do consumo de memória e o intervalo de tempo decorrido, conforme a Equação 4.2.

$$TVCM = \frac{vcm_2 - vcm_1}{t_2 - t_1} \tag{4.2}$$

Os seguintes estados podem ser atribuídos para cada propriedade apresentada: Low (L), Medium (M) e High (H). Os estados L, M e H representam respectivamente um conjunto de números pequenos, médios e grandes em um intervalo de valores estabelecidos. A definição de L, M e H são usadas para representar o comportamento de uma determinada propriedade de uma forma simples e abstrata.

Assumindo que os valores para uma determinada propriedade está em um intervalo positivo $[limite_{minimo}, limite_{maximo}]$, então o mesmo pode ser dividido em 3 subintervalos que representam os estados *L*, *M* e *H*, como ilustrado na Figura 4.2. Observe que os valores de L, M e H são dependentes do limite mínimo e máximo de um intervalo estabelecido.

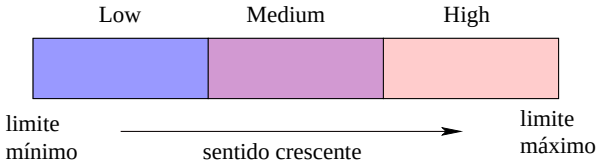

Figura 4.2: Subintervalos que representam os estados *Low (L)*, *Medium (M)* e *High (H)*.

A quantidade de memória consumida está localizada em um desses subintervalos, assumindo um dos três estados $\{Low, Medium, High\}$ apresentados. Sendo assim a quantidade de memória consumida é classificada como *baixo consumo* de memória quando o seu valor está no estado L, *médio consumo* de memória quando o seu valor está no estado M e *alto consumo* de memória quando localizado no estado H. A quantidade de memória consumida abrange somente valores inteiros positivos, visto que consumo de memória de valor negativo é logicamente inexistente no mundo real.

Analogamente, isso é aplicado também para os valores de VCM e TVCM. Levando em consideração que VCM e TVCM podem assumir valores negativos, a divisão do intervalo $[limite_{minimo}, limite_{maximo}]$ pode consistir em 3 subintervalos para valores negativos e 3 subintervalos para valores positivos, conforme ilustrado na Figura 4.3.

Portanto VCM é considerada como *variação baixa* quando o valor está em L, *variação média* quando o valor está em M e *variação alta* quando o valor está em H. VCM assume

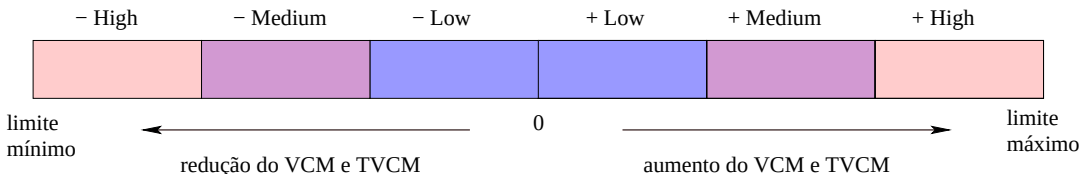

Figura 4.3: Subintervalos que representam os estados *Low (L)*, *Medium (M)* e *High (H)* para valores negativos e positivos.

valores negativos e positivos, pois páginas de memória que foram alocadas anteriormente podem ser liberadas posteriormente.

Por exemplo: a memória utilizada por uma aplicação é 1024 KB em um dado instante t_1 e a aplicação carrega um arquivo em um instante t_2 aumentando o consumo de memória para 2048 KB. Após isso, o arquivo que foi carregado é encerrado no instante t_3, levando o consumo de memória de volta para 1024 KB. No intervalo de tempo entre t_2 e t_1, a variação do consumo sofreu um crescimento, assumindo portanto um valor positivo. Por outro lado, no intervalo de tempo entre t_3 e t_2, a variação foi reduzida, assumindo um valor negativo. Considerando que o intervalo entre os tempos seja de 1s, os valores de VCM são respectivamente para cada intervalo mencionado: 1024 KB/s e -1024 KB/s.

Finalmente TVCM é classificada como *taxa de variação baixa* quando o seu valor está em L, *taxa de variação média* quando o valor está em M e *taxa de variação alta* quando o valor correspondente está em H. TVCM também assume valores negativos e positivos, visto que VCM sofre variações durante a execução de um programa.

Os dados utilizados para classificar os padrões de consumo de memória são representados pela tripla ⟨*memória, vcm, tvcm*⟩. Visto que os parâmetros da tripla ⟨memória, vcm, tvcm⟩ podem assumir respectivamente 3, 6 e 6 estados diferentes, a tripla é capaz de representar no total 108 classes diferentes de configuração de consumo de memória, através da combinação de todos os estados possíveis entre os parâmetros da tripla.

As classes que representam um consumo elevado de memória possuem o primeiro parâmetro no estado H, independentemente de quais são os valores instanciados nos outros parâmetros da tripla. Tais classes são consideradas como *críticas* e representadas pelas triplas da Tabela 4.1.

Classes Críticas		
$\langle H_{mem}, \pm L_{vcm}, \pm L_{tvcm} \rangle$	$\langle H_{mem}, \pm L_{vcm}, \pm M_{tvcm} \rangle$	$\langle H_{mem}, \pm L_{vcm}, \pm H_{tvcm} \rangle$
$\langle H_{mem}, \pm M_{vcm}, \pm L_{tvcm} \rangle$	$\langle H_{mem}, \pm M_{vcm}, \pm M_{tvcm} \rangle$	$\langle H_{mem}, \pm M_{vcm}, \pm H_{tvcm} \rangle$
$\langle H_{mem}, \pm H_{vcm}, \pm L_{tvcm} \rangle$	$\langle H_{mem}, \pm H_{vcm}, \pm M_{tvcm} \rangle$	$\langle H_{mem}, \pm H_{vcm}, \pm H_{tvcm} \rangle$

Tabela 4.1: Triplas consideradas como classes críticas de consumo de memória.

As classes consideradas candidatas potenciais para atingir um consumo elevado de memória apresentam o primeiro parâmetro no estado M, o segundo e/ou o terceiro parâmetro no estado H. Se VCM ou TVCM está elevada e a quantidade alocada de memória está prestes a sair do estado M e entrar no estado H, então a classe que apresenta este tipo de comportamento tem uma grande possibilidade de alcançar um consumo alto de memória. As classes pertencentes a este tipo de situação são consideradas como p*otencialmente críticas* e representadas pelas triplas da Tabela 4.2.

Classes Potencialmente Críticas	
$\langle M_{mem}, +H_{vcm}, +L_{tvcm} \rangle$	$\langle M_{mem}, +H_{vcm}, +M_{tvcm} \rangle$
$\langle M_{mem}, +L_{vcm}, +H_{tvcm} \rangle$	$\langle M_{mem}, +M_{vcm}, +H_{tvcm} \rangle$

Tabela 4.2: Triplas consideradas como classes potencialmente críticas de consumo de memória.

Considerando o modelo de rede neural auto-organizável descrito na Seção 3.3, o padrão de consumo ⟨memória, vcm, tvcm⟩ pode ser mapeado facilmente em estrutura de dados aceitos pelo SOM. Visto que os dados de entrada e os pesos sinápticos da rede SOM são representados por vetores, a tripla ⟨memória, vcm, tvcm⟩ é mapeada diretamente para um vetor constituído de 3 dimensões, onde cada dimensão representa um argumento da tripla, como apresentada na Figura 4.4.

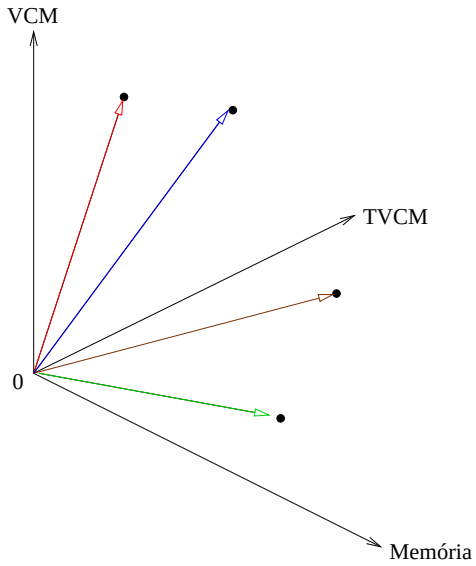

Figura 4.4: Tripla ⟨memória, vcm, tvcm⟩ representada em um espaço vetorial de 3 dimensões.

Cada vetor no espaço tridimensional é uma instância específica da tripla que é mapeado na grade de neurônios do SOM. Cada célula da grade de neurônios armazena então um

vetor tridimensional do tipo [memória, vcm, tvcm], conforme a Figura 4.5.

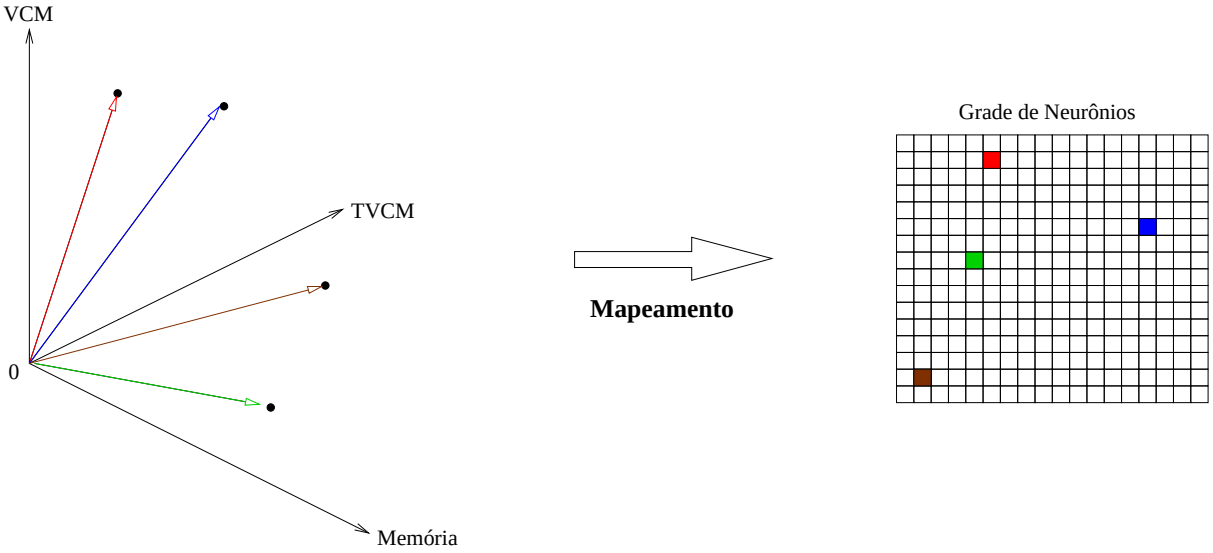

Figura 4.5: Os vetores são mapeados em um espaço bidimensional representado pela grade de neurônios.

A idéia de utilizar o mapa de neurônios auto-organizáveis é agrupar topologicamente as classes apresentadas de consumo de memória em áreas ou regiões distintas. Dessa forma, cada região representa uma configuração ou conjunto de configurações similares capaz de representar o estado do consumo de memória em um determinado instante.

Portanto o mapa de neurônios é constituído de regiões críticas e potencialmente críticas, assim como regiões estáveis de consumo de memória. As regiões estáveis abrangem as triplas que não são classificadas como críticas ou potencialmente críticas. A implementação dessa idéia é descrita nas Seções 3.2 e 5.3.

Sumário

Neste Capítulo foram descritos os padrões de consumo de memória e a possibilidade de tais padrões serem representados em uma rede neural auto-organizável em função da quantidade de memória física consumida, da variação do consumo de memória e da taxa de variação do consumo de memória.

No próximo Capítulo serão apresentados os resultados alcançados nesta dissertação que abrangem a definição dos cenários de casos de uso e algumas questões relacionadas ao treinamento e funcionamento da rede neural para classificação de padrões de consumo de memória.

Capítulo 5

Resultados Alcançados

Este Capítulo apresenta os experimentos realizados e resultados produzidos durante o decorrer deste trabalho. Os cenários dos casos de uso são descritos na Seção 5.1, o procedimento para treinar a rede neural utilizando mapas auto-organizáveis é apresentado na Seção 3.2 e a aplicação implementada para classificar os padrões de consumo de memória baseada em uma rede neural treinada é introduzida na Seção 5.3.

5.1 Descrição de Casos de Uso

Cenários de *casos de uso* foram projetados para prover os dados de treinamento do mapa auto-organizável e permitir a verificação do padrão de consumo de memória de cada cenário após o treinamento da rede.

O padrão de consumo de memória não está somente interligado com o tipo de aplicação, mas principalmente com a maneira em que a aplicação é utilizada em diferentes cenários. Uma determinada aplicação pode pertencer a uma classe A de padrão de consumo de memória em um cenário e uma classe B em um outro cenário.

Uma forma simples de exemplificar isso é utilizar um editor de texto para carregar um arquivo de 10 KB e o mesmo editor para carregar um arquivo de 10 MB. Certamente o consumo de memória do mesmo editor é diferente nos dois cenários descritos, visto que o arquivo carregado por cada cenário ocupa espaços diferentes na memória.

Todos os casos de uso são executados de forma automática através da implementação de scripts na linguagem Python. O projeto *Linux Desktop Testing Project (LDTP)* foi utilizado para manipular as aplicações baseadas em interfaces gráficas de maneira automatizada. O LDTP [30] oferece um conjunto de APIs em Python que permite acessar os diversos componentes gráficos de uma aplicação com interface baseada em *GTK*, acrônimo do *Gimp Toolkit* [31].

Visto que o LDTP proporciona a viabilidade de manipular os componentes gráficos do GTK, as aplicações selecionadas para o projeto dos casos de uso devem apresentar a implementação da interface gráfica baseada em GTK. Portanto as aplicações utilizadas como casos de uso são: o visualizador de documentos em formato PDF chamado de *Gpdf*, o navegador Web *Galeon*, o tocador de vídeo *Totem*, o editor de texto *Gedit* e o visualizador de imagem *Gthumb*, que são todos implementados em GTK.

Em termos práticos, o LDTP possibilita os scripts a disparar ações como clique de mouse em botões e menus, inserção de caracteres em campos de texto, movimentação das barras de rolagens e outros. Com esse tipo de recurso disponível, os cenários podem ser realizados várias vezes de modo sistemático sem a necessidade da intervenção de um usuário real.

O módulo *run_app_analyse.py* é o script principal utilizado para disparar os cenários de casos de uso. Conforme os parâmetros de entrada fornecidos para o módulo run_app_analyse.py, o módulo *main_[nome da aplicação].py* é chamado para iniciar a aplicação, o coletor de dados e o caso de uso correspondente. O coletor de dados é um programa implementado em C que efetua a leitura dos dados de consumo de memória da aplicação e armazena as informações coletadas no disco após a conclusão do caso de uso. O caso de uso é realizado pelo módulo *[aplicação]_[número do caso de uso].py* que depende da aplicação e do caso de uso fornecidos como parâmetros de entrada. Um exemplo de script utilizado para implementar o primeiro cenário de caso de uso do gpdf é apresentado no Algoritmo 5.1.

A descrição do script é apresentada resumidamente na lista abaixo.

1 Diretiva para permitir executar o script como arquivo executável.

2-3 Importa os módulos do LDTP.

4 Importa alguns módulos padrões.

6 Inicializa o mapa *gpdf_1.map* que descreve a hierarquia dos componentes gráficos da aplicação.

7 Inicializa o leitor de PDF.

8 Adquire a lista de nomes dos arquivos PDFs que serão carregados.

10 Carrega cada arquivo da lista.

11 Acessa o menu para abrir a janela ou caixa de diálogo para a escolha de arquivos.

Algoritmo 5.1 Script que realiza o cenário de caso de uso da aplicação Gpdf.

```
1    #!/usr/bin/env python
2    from ldtp import *
3    from ldtputils import *
4    import sys, string, os
5
6    initappmap('%s/gpdf_1.map' % os.getcwd())
7    launchapp('gpdf')
8    files = os.listdir('pdf_files')
9    title_name = 'PDF Viewer'
10   for i in range(0, len(files)):
11       selectmenuitem('frmPDFViewer', 'mnuFile;mnuOpen')
12       if waittillguiexist('dlgLoadfile'):
13           wait(1)
14           selectrow('dlgLoadfile', 'tblFiles', 'pdf_files')
15           wait(1)
16           click('dlgLoadfile', 'btnOpen')
17           wait(1)
18           selectrowindex('dlgLoadfile', 'tblFiles', i)
19           click('dlgLoadfile', 'btnOpen')
20           setcontext(title_name, files[0])
21           wait(10)
```

12 Espera a janela de diálogo ser exibida.

14 Seleciona a pasta pdf_files no diretório atual.

16 Abre a pasta pdf_files.

18 Seleciona o arquivo PDF a ser carregado.

19 Abre o arquivo PDF selecionado.

21 Espera por 10 segundos para carregar o próximo arquivo PDF.

Figura 5.1 mostra o conjunto de chamadas dos módulos a partir do módulo principal run_app_analyse.py.

Os cenários definidos neste trabalho procuram simular casos reais de uso de um sistema computacional como por exemplo visualizar documentos PDFs, navegar por diversas páginas da Internet, tocar arquivos de multimídia, editar arquivos textos e visualizar as fotos tiradas em uma viagem feita à Europa.

Os cenários para o visualizador Gpdf são apresentados a seguir.

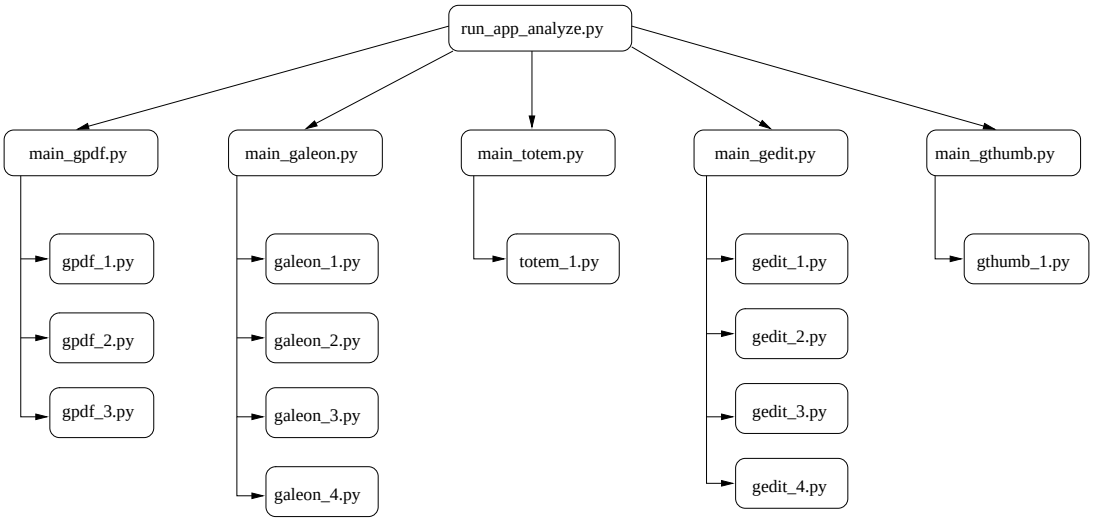

Figura 5.1: Módulos em Python que automatiza a coleta de dados e a execução dos cenários de casos de uso.

Cenário 1: Carrega arquivos PDFs.

1. Inicializar o programa Gpdf.
2. Carregar arquivos PDFs sequencialmente utilizando o menu 'File → Open'.

Cenário 2: Percorre um arquivo PDF página por página.

1. Inicializar o programa Gpdf juntamente com o arquivo PDF a ser carregado.
2. Visualizar a página movendo a barra de rolagem para baixo.
3. Seguir para a página seguinte clicando no botão 'Next'.
4. Voltar para o passo 2 ou finalizar o caso de uso após várias iterações.

Cenário 3: Percorre dois arquivos PDFs de modo alternado, realizando diferentes visualizações de zoom.

1. Inicializar o programa Gpdf juntamente com o arquivo PDF a ser carregado.
2. Carregar um segundo arquivo PDF clicando no botão 'Open'.
3. Ampliar o zoom no segundo e depois no primeiro arquivo utilizando o menu 'View → Zoom In'.
4. Reduzir o zoom no segundo e depois no primeiro arquivo utilizando o menu 'View → Zoom Out'.

5. Ajustar a visualização da página no segundo e depois no primeiro arquivo utilizando o menu 'View → Best Fit'.
6. Ajustar a visualização da página conforme a largura da mesma no segundo e depois no primeiro arquivo utilizando o menu 'View → Fit page width'.
7. Seguir para a página seguinte clicando no botão 'Next' da janela do segundo arquivo.
8. Visualizar a página movendo a barra de rolagem para baixo do segundo arquivo.
9. Repetir o passo 6 e 7 para o primeiro arquivo.
10. Voltar para o passo 3 ou finalizar o caso de uso após várias iterações.

Os cenários para o navegador Galeon são basicamente relacionados com o acesso a páginas Web, como mostrado a seguir.

Cenário 1: Carrega várias páginas Web na mesma aba.

1. Inicializar o navegador Galeon.
2. Carregar várias páginas Web sequencialmente na mesma aba.

Cenário 2: Carrega várias páginas Web em abas diferentes.

1. Inicializar o navegador Galeon.
2. Carregar página Web.
3. Abrir uma nova aba utilizando o menu 'File → New Tab'.
4. Voltar para o passo 2 ou finalizar o caso de uso após todas as páginas Web forem carregadas.

Cenário 3: Carrega e descarrega páginas Web aleatoriamente.

1. Inicializar o navegador Galeon.
2. Carregar página Web.
3. Abrir uma nova aba utilizando o menu 'File → New Tab'.
4. Carregar página Web na nova aba.
5. Fechar aleatoriamente uma aba com a respectiva página Web anteriormente carregada.
6. Voltar para o passo 3 ou finalizar o caso de uso após todas as páginas Web forem carregadas.

Cenário 4: Carrega e depois descarrega as páginas Web de cada aba.

1. Inicializar o navegador Galeon.
2. Carregar página Web.
3. Abrir uma nova aba utilizando o menu 'File → New Tab'.
4. Voltar para o passo 2 ou prosseguir para o passo seguinte após todas as páginas Web forem carregadas.
5. Fechar todas as abas com as respectivas páginas Web carregadas.

Para o tocador de videos Totem foi definido um único cenário que consiste em carregar alguns arquivos de vídeo.

Cenário: Toca arquivos de vídeo.

1. Inicializar o tocador de vídeo Totem.
2. Carregar um arquivo de formato MPEG utilizando o menu 'Movie → Open Location'.
3. Executar o vídeo.
4. Voltar para o passo 2 ou finalizar o caso de uso após todos os arquivos MPEG forem carregados.

Os cenários para o editor de texto Gedit são descritos a seguir.

Cenário 1: Edita um novo arquivo texto e depois salva o arquivo editado.

1. Inicializar o editor Gedit.
2. Digitar uma seqüência de caracteres dezenas de milhares de vezes no editor.
3. Salvar o arquivo utilizando o menu 'File → Save As'.

Cenário 2: Edita um novo arquivo texto e salva o arquivo ao longo da edição.

1. Inicializar o editor Gedit.
2. Digitar uma seqüência de caracteres milhares de vezes no editor.
3. Salvar o arquivo utilizando o menu 'File → Save As'.
4. Digitar novamente uma seqüência de caracteres milhares de vezes no editor.
5. Salvar o arquivo utilizando o menu 'File → Save'.

6. Voltar para o passo 4 ou finalizar o caso de uso após algumas iterações.

Cenário 3: Carrega arquivos textos.

1. Inicializar o editor Gedit.
2. Carregar um arquivo texto em uma aba.
3. Voltar para o passo 2 ou finalizar o caso de uso após todos os arquivos texto forem carregados

Cenário 4: Carrega arquivos textos e depois descarrega os arquivos abertos.

1. Inicializar o editor Gedit.
2. Carregar um arquivo texto em uma aba.
3. Voltar para o passo 2 ou prosseguir para o passo seguinte após todos os arquivos texto forem carregados.
4. Fechar todas as abas com os respectivos arquivos texto carregados.

Para o visualizador de imagens Gthumb foi definido um único cenário que consiste em carregar vários arquivos de imagens:

Cenário: Visualiza diversos arquivos de imagens.

1. Inicializar o editor Gthumb com o caminho do diretório de imagens.
2. Carregar um arquivo de imagem.
3. Voltar para o passo 2 ou finalizar o caso de uso após todos os arquivos JPEG forem carregados.

Os casos de usos descritos anteriormente são executados com o propósito de coletar os dados relacionados ao consumo de memória. Durante a execução de cada cenário, as informações relativas à quantidade de memória física alocada são coletadas continuamente a cada 10 milisegundos e armazenadas no arquivo quando o cenário é concluído. O arquivo pode ser então processado para obter os valores VCM e TCM do consumo de memória do caso de uso correspondente.

O resultado deste processo gera um histórico que contém uma seqüência de triplas do tipo $\langle mem_t, vcm_t, tvcm_t \rangle$, onde as variáveis *mem*, *vcm* e *tvcm* são respectivamente a quantidade de memória consumida, a variação e a taxa de variação do consumo de memória de cada caso de uso em um instante t. O procedimento é ilustrado na Figura 5.2. O conjunto de histórico de todos os casos de usos constitui os dados de entrada

da rede neural durante a sua fase de treinamento. O treinamento da rede neural para classificação de padrão de consumo de memória é descrito na Seção 5.2.

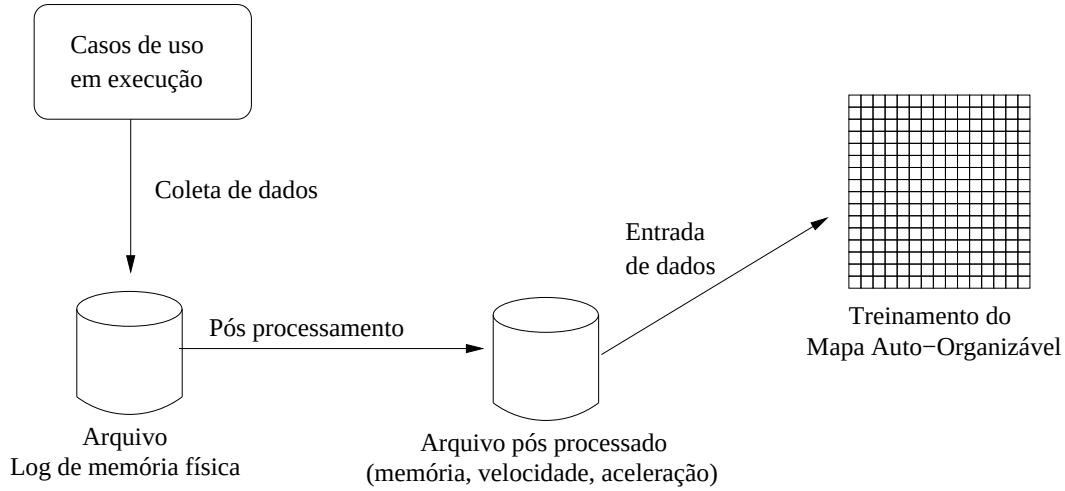

Figura 5.2: Procedimento efetuado para a coleta de dados do treinamento da rede neural.

5.2 Treinamento da Rede Neural

O treinamento da rede para classificação de padrão de consumo de memória é baseado na descrição de SOM apresentada na Seção 3.3. A rede neural é implementada na linguagem C e baseada na biblioteca GNU libc. Inclusive os algoritmos apresentados na Seção 3.3 para mostrar as etapas do treinamento da rede foram obtidos da implementação desta dissertação.

O conjunto de vetores de entrada utilizado para treinar a rede é baseado em informações de consumo de memória coletadas durante a execução de cada caso de uso, como ilustrada na Figura 5.2. Como a tripla determinada para representar o padrão de consumo de memória é do tipo $\langle mem, vcm, tvcm \rangle$, logo o vetor de entrada x da rede é composto por 3 dimensões $x = [x_1, x_2, x_3]$, onde cada dimensão representa um parâmetro da tripla. Isso implica também que o vetor de pesos sinápticos dos neurônios da grade seja da mesma dimensão, como descrito na Seção 3.3. Portanto os pesos sinápticos de cada neurônio j é representado como $w_j = [w_{j1}, w_{j2}, w_{j3}]$.

Uma interface gráfica baseada em GTK foi desenvolvida para permitir a visualização da configuração da rede antes e depois do treinamento. Além disso, para poder verificar a configuração dos pesos sinápticos dos neurônios antes e depois do treinamento, cores são usadas para exibir os ajustes ou adaptações sinápticas de todos os neurônios da grade.

Um mapeamento é necessário para poder estabelecer uma relação de cores com os pesos sinápticos de cada neurônio, visto que cada componente do vetor apresenta valores no intervalo de $[0, 1]$ na rede (ver Seção 3.3). Basicamente uma transformação matemática é aplicada em cada dimensão do vetor para obter os 3 componentes de cores RGB. Como os atributos *red*, *green* e *blue* da estrutura *GdkColor*, que representam os 3 componentes RGB, varia no intervalo de $[0, 65535]$, basta multiplicar então 65535 por cada elemento do vetor, obtendo assim o valor desejável. Um exemplo de código deste procedimento é apresentado no Algoritmo 5.2.

Algoritmo 5.2 Mapeamento do vetor de pesos sinápticos do neurônio localizado na posição (i, j) da grade em cores RGB.

```
color.red = (int)(grids[i][j]->weights[0] * 65535);
color.blue = (int)(grids[i][j]->weights[1] * 65535);
color.green = (int)(grids[i][j]->weights[2] * 65535);
```

Como a rede neural inicializa os pesos sinápticos de cada neurônio de modo aleatório, a grade de neurônios apresenta inicialmente várias cores diferentes distribuídas de maneira não uniforme ao longo do mapa neural. Após o treinamento da rede, as cores apresentadas estarão organizadas de uma forma ordenada e agrupadas por pesos sinápticos similares, como ilustra a Figura 5.3.

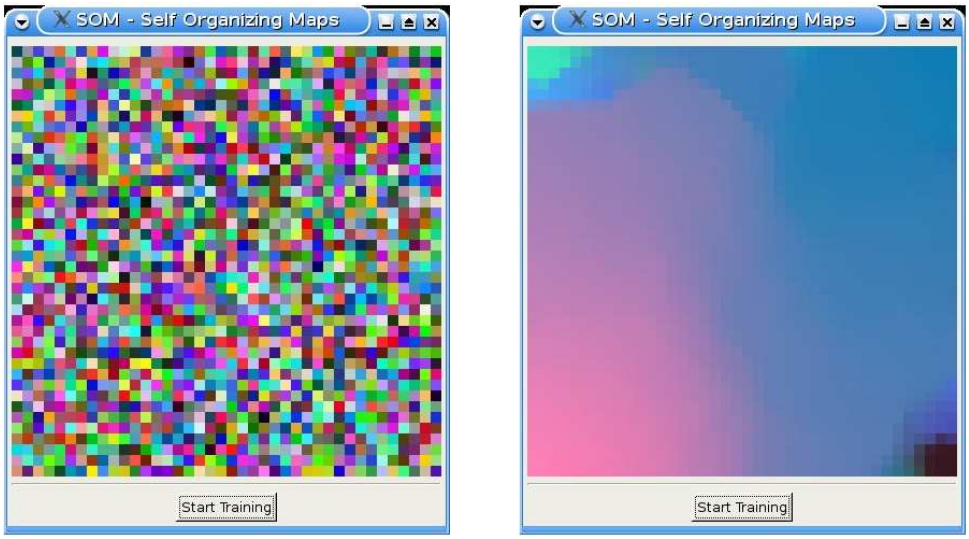

Figura 5.3: Visualização da grade de neurônios antes, posicionada no lado esquerdo, e depois, posicionada no lado direito, do treinamento da rede.

Além disso, os valores das triplas $\langle mem, vcm, tvcm \rangle$ precisam ser também convertidos apropriadamente, pois os componentes do vetor de entrada utilizado para treinar a rede

estão também no intervalo de $[0, 1]$. Basicamente isso pode ser realizado aplicando um método de normalização [32], como formulado pela Equação 5.1:

$$\delta(d) = \frac{d - d_{min}}{d_{max} - d_{min}} \tag{5.1}$$

onde d_{min} e d_{max} representam respectivamente o limite inferior e superior do valor d a ser convertido. Os limites inferior e superior são definidos através dos dados coletados. Por exemplo, os valores coletados para a taxa de alocação de memória física durante a execução dos cenários de casos de uso mostram que o menor e o maior valor coletado são respectivamente -5570 e 2305. Logo tais valores correspondem ao limite inferior e superior da taxa de alocação. O código da Equação 5.1 é apresentado no Algoritmo 5.3.

Algoritmo 5.3 Código que normaliza os valores para o intervalo $[0, 1]$ conforme a Equação 5.1.

```
double normalize(double value, double max, double min) {
    return (value-min)/(max-min);
}
```

O treinamento da rede SOM é realizado pela função `train()` que recebe como parâmetros uma lista de vetores de entrada e o tamanho da lista. No término da execução da função `train()`, a grade de neurônios terá os seus pesos sinápticos ajustados e adaptados para classificar os padrões de consumo de memória dos cenários de casos de uso definidos na Seção 5.1. A implementação da função `train()` é mostrada no Algoritmo 5.4 e a descrição do código é apresentada a seguir.

2-3: Tamanho da grade de neurônios.

6: Tamanho inicial do raio de vizinhança topológica.

7: Constante de tempo é dependente do número de iterações e do raio.

14: Taxa de aprendizagem inicial.

16: Treina a rede *NUM_ITERATIONS* vezes. *NUM_ITERATIONS* é um macro que define a quantidade de vezes que o treinamento da rede é efetuado.

17-18: Calcula o raio da vizinhança topológica para o treinamento atual.

19: Percorre o vetor de entrada para treinar a rede.

20: Acessa os pesos sinápticos do vetor de entrada.

Algoritmo 5.4 Função `train()` implementada para efetuar o treinamento da rede SOM.

```
void train(struct input_pattern * inputs[], int input_len) {
    int lw = GRIDS_XSIZE;
    int lh = GRIDS_YSIZE;
    int xstart, ystart, xend, yend;
    double dist, influence;
    double grid_radius = max(lw, lh)/2;
    double time_constant = NUM_ITERATIONS / log(grid_radius);
    int iteration = 0;
    double nbh_radius;
    struct som_node * bmu = NULL;
    struct som_node * temp = NULL;
    double * new_input = NULL;
    double learning_rate = START_LEARNING_RATE;
    int i, x, y;
    while (iteration < NUM_ITERATIONS) {
        nbh_radius = neighborhood_radius(grid_radius, iteration,
                                         time_constant);
        for (i=0; i<input_len; i++) {
            new_input = inputs[i]->weights;
            bmu = get_bmu(new_input, WEIGHTS_SIZE, grids);
            xstart = (int)(bmu->xp - nbh_radius - 1);
            ystart = (int)(bmu->yp - nbh_radius - 1);
            xend = (int)(bmu->xp + nbh_radius + 1);
            yend = (int)(bmu->yp + nbh_radius + 1);
            if (xend > lw) xend = lw;
            if (xstart < 0) xstart = 0;
            if (yend > lh) yend = lh;
            if (ystart < 0) ystart = 0;
            for (x=xstart; x<xend; x++) {
                for (y=ystart; y<yend; y++) {
                    temp = grids[x][y];
                    dist = distance_to(bmu, temp);
                    if (dist <= (nbh_radius * nbh_radius)) {
                        influence = get_influence(dist, nbh_radius);
                        adjust_weights(temp, inputs[i], WEIGHTS_SIZE,
                                       learning_rate, influence);
                    }
                }
            }
        }
        iteration++;
        learning_rate = START_LEARNING_RATE *
                exp(-(double)iteration/NUM_ITERATIONS);
    }
}
```

21: Encontra o neurônio vencedor ou BMU conforme o vetor de entrada fornecido.

22-29: Otimização aplicada para percorrer a grade neurônios. Somente os valores (x, y) que está localizado dentro do raio é acessado.

30-31: Percorre os neurônios da grade.

32-33: Calcula a distância entre o BMU e o neurônio na posição (x, y) da grade (ver Algoritmo 3.2).

34: Somente altera o neurônio localizado dentro na vizinhança topológica (ver Figura 3.3).

35: Calcula o fator de influência para o ajuste dos pesos sinápticos (ver Equação 3.6 e Algoritmo 3.7).

36-37: Ajusta os pesos sinápticos do neurônio (ver Equação 3.4 e Algoritmo 3.5).

43-44: Calcula a taxa de aprendizagem para a próxima iteração de treinamento (ver Equação 3.5).

Quando o treinamento da rede é concluído, a nova configuração do mapa de neurônios é armazenada no disco, permitindo que a rede neural treinada possa ser depois carregada. É importante ressaltar que o treinamento da rede é um processo custoso, visto que durante a experimentação deste trabalho, a duração do treinamento levava em torno de 7 horas para ser concluído. Para evitar que o mesmo treinamento seja feito novamente, é viável que a configuração da rede treinada possa ser armazenada e recuperada posteriormente por uma aplicação que utilize tais informações para classificar os padrões de dados.

5.3 Funcionamento da Rede Neural Treinada

Uma ferramenta gráfica, que leva o mesmo nome do modelo de rede neural implementado neste trabalho, chamada de SOM foi desenvolvida para exibir o funcionamento da rede neural treinada. O objetivo dessa ferramenta é mostrar a evolução do consumo de memória de uma determinada aplicação fundamentada nas possíveis classes representadas pela rede neural auto-organizável.

As classes descritas no Seção 4.2 são mapeadas nas regiões de cores agrupadas e ordenadas de modo topológico após o treinamento da rede. A ferramenta SOM possui duas maneiras para visualizar a evolução do consumo de memória de uma aplicação. A primeira

maneira, chamada de *online*, consiste em mostrar o progresso do consumo de memória enquanto a aplicação monitorada está sendo executada, ou seja, todo procedimento é realizado durante a execução da aplicação. A segunda maneira, chamada de *offline*, é exibir o progresso do consumo de memória baseado em arquivos de log. Neste caso a aplicação é executada e as suas informações de consumo de memória são monitoradas e salvas em arquivos de log. A ferramenta SOM é iniciada posteriormente para exibir o comportamento da aplicação monitorada através da leitura do arquivo de log.

A informação do consumo de memória é obtida através da leitura da entrada *statm* disponível no sistema de arquivos /proc. Como o valor de *statm* é relacionado a cada processo do sistema, o seu formato de acesso é do tipo */proc/[PID]/statm*.

O motivo de estabelecer o mecanismo de visualização baseada em log é porque existem casos de uso que consomem recursos de hardware demasiadamente, impossibilitando assim o funcionamento da ferramenta no online de ser concluído com sucesso. O primeiro caso de uso da aplicação Gedit, descrito na Seção 5.1, é um exemplo que pode ser mencionado. Durante a execução deste caso de uso, foi certificado que a ferramenta SOM não consegue exibir a evolução do consumo de memória, pois a interface gráfica da ferramenta fica congelada, sem a capacidade de mostrar as mudanças do mapa neural.

Figura 5.4 mostra a interface da ferramenta SOM desenvolvida. A opção *Runtime* que é habilitada como a escolha padrão, permite visualizar a evolução do consumo de memória no modo online, enquanto que a opção *Logfile* quando selecionada, permite visualizar a evolução do consumo de memória no modo offline.

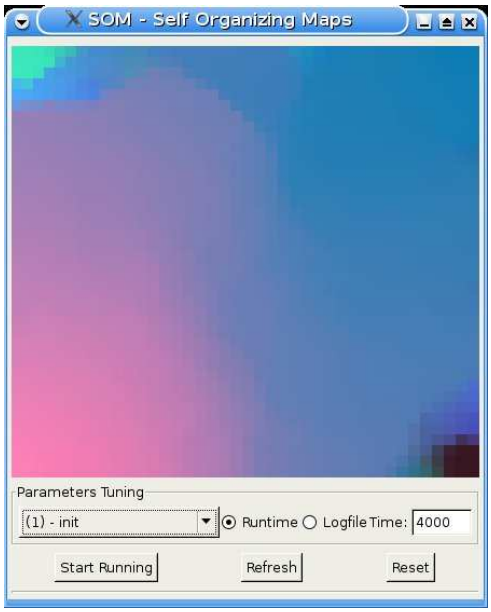

Figura 5.4: Interface da ferramenta SOM com a opção Runtime habilitada.

No modo online, os parâmetros de entrada proporcionada pela interface são: o número do processo ou simplesmente *PID*, do inglês *Process ID*, usada para identificar uma determinada aplicação; e o tempo estabelecido para a ferramenta SOM monitorar a aplicação. No modo offline, o parâmetro de entrada é apenas o nome do arquivo de log, como ilustrado na Figura 5.5.

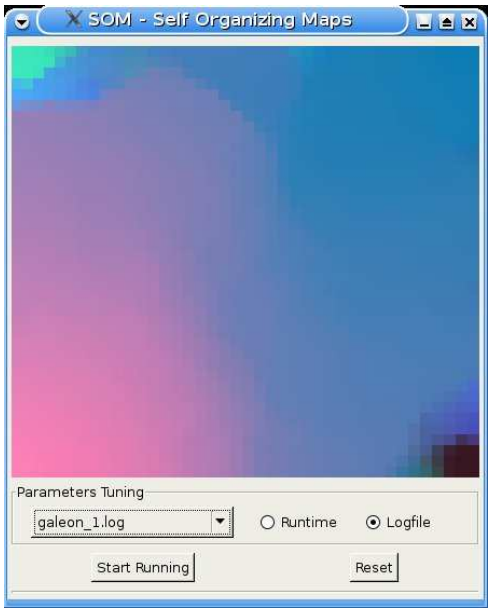

Figura 5.5: Interface da ferramenta SOM com a opção Logfile habilitada.

A região crítica é representada pela cor rosa que inclui as instâncias da classe $\langle H_{mem}, \pm L_{vcm}, \pm L_{tvcm} \rangle$, enquanto que a região estável é composta pelas cores restantes azul, verde, preta e magenta. As regiões azul, verde, preta e magenta abrangem respectivamente as instâncias da classe $\langle L_{mem}, \pm L_{vcm}, \pm L_{tvcm} \rangle$, $\langle L_{mem}, +L_{vcm}, +H_{tvcm} \rangle$, $\langle L_{mem}, -H_{vcm}, -H_{tvcm} \rangle$ e $\langle M_{mem}, \pm L_{vcm}, \pm L_{tvcm} \rangle$.

Apesar de várias classes terem sido descritas na Seção 4.2, a rede neural treinada durante a experimentação é capaz de prover no mapa topográfico somente algumas classes, pois a capacidade do SOM de representar um maior número de classes depende da diversidade de dados fornecidos para a rede durante a fase de treinamento.

A informação de consumo de memória coletada, representada pela tripla ⟨memória, vcm, tvcm⟩, em um dado instante corresponde a uma célula (x, y) da grade de neurônios. Essa correspondência é realizada verificando qual é o neurônio da grade com propriedades mais semelhantes ou idênticas à informação de consumo de memória coletada. A função implementada para encontrar o BMU é utilizada para realizar esse procedimento (ver Algoritmo 3.2). Neste caso o BMU que está na célula (x, y) representa o estado de

consumo de memória correspondente de uma determinada instância da tripla ⟨memória, vcm, tvcm⟩.

As células da grade que correspondem às informações de consumo de memória de uma dada aplicação são marcadas como visitadas no mapa neural. As células visitadas são representadas pela cor branca. Figura 5.6 mostra o mapa de consumo de memória do primeiro caso de uso do navegador Galeon. Observe que a maioria das células visitadas pertencem a região azul do mapa neural e portanto o padrão de consumo de memória é classificada como estável do tipo $\langle L_{mem}, \pm L_{vcm}, \pm L_{tvcm} \rangle$.

Figura 5.6: Mapa de consumo de memória do primeiro cenário de caso de uso do navegador Galeon.

Figura 5.7 ilustra o mapa de consumo de memória do primeiro caso de uso definido para o editor de texto Gedit. As células visitadas abrangem uma área de maior alcance, atingindo a região estável (azul) e a região crítica (rosa). Neste caso o padrão de consumo de memória é considerado como crítico do tipo $\langle H_{mem}, \pm L_{vcm}, \pm L_{tvcm} \rangle$. Apesar da área representada pela cor rosa não ter sido ocupada completamente, ainda é válido afirmar que o padrão de consumo de memória é crítico, pois uma pequena porção da região representada pela cor rosa foi ocupada.

Um programa que consome memória continuamente, onde cada alocação de memória pode variar entre 524 KB e 2 MB, é considerado também pertencente a classe $\langle H_{mem}, \pm L_{vcm}, \pm L_{tvcm} \rangle$, pois o consumo de memória chega a atingir a região crítica (rosa), como ilustra a Figura 5.8.

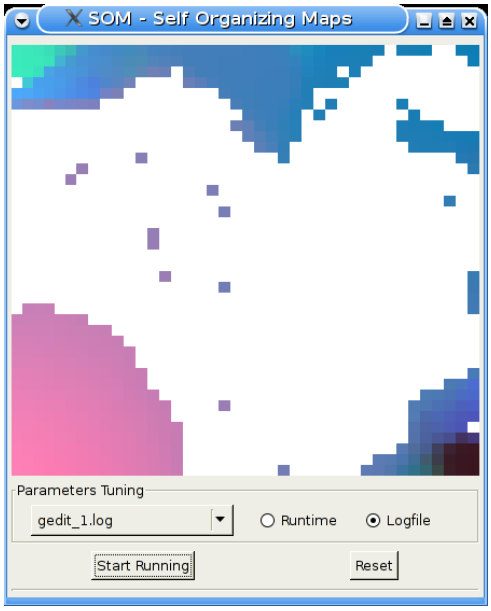

Figura 5.7: Mapa de consumo de memória do primeiro cenário de caso de uso do editor de texto Gedit.

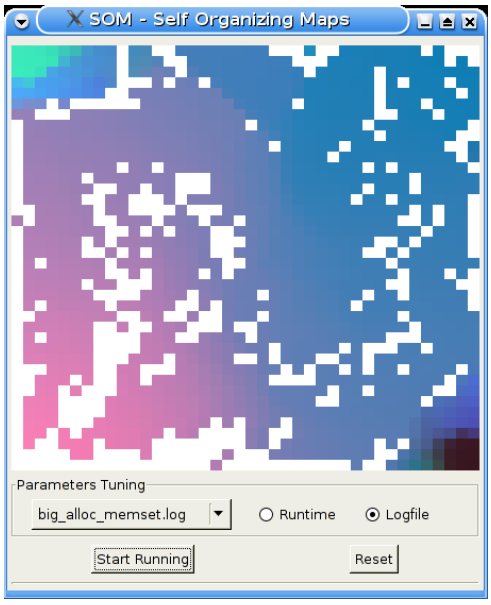

Figura 5.8: Mapa de consumo de memória do um programa que consome memória continuamente.

Algoritmo 5.5 mostra um exemplo de programa que segue este tipo de comportamento de consumo de memória. A utilização da função `malloc()` garante que a alocação foi efetuada somente na memória virtual do Linux. A alocação é efetivamente realizada na memória física quando a área criada pelo `malloc()` é acessada de alguma forma. Sendo assim, a função `memset()`, que escreve em uma determinada área da memória, é utilizada para que o tamanho da memória alocada seja refletido imediatamente na memória física.

Algoritmo 5.5 Programa que consome memória continuamente.

```
int main (void) {
    int i, size;
    char *p;

    srand(time(0));
    for (i=0; i<20000; i++) {
        sleep(1);
        /* Retorna um valor entre 2^19 e 2^21 bytes */
        size = get_rand(1 << 19, 1 << 21);
        p = malloc(size);
        memset(p, 0, size);
    }
    return 0;
}
```

De acordo com as Figuras 5.6 e 5.7 apresentadas, as regiões do mapa neural visitadas fornecem um mecanismo de identificar o comportamento de consumo de memória e classificar os padrões de consumo de memória de um determinado caso de uso. A ferramenta SOM pode ser utilizada para efetuar análises relacionadas ao desempenho de sistemas, com o objetivo de detectar as aplicações que são propensas em consumir memória de forma excessiva.

Sumário

Neste Capítulo foram descritos os cenários de casos de uso utilizados para fornecer os dados de treinamento da rede. A rotina de treinamento da rede e o modo como os dados de consumo de memória são mapeados apropriadamente como entradas da rede neural foram também explanadas. Além disso, o funcionamento da rede para classificar os padrões de consumo de memória foi apresentado através da ferramenta SOM implementada nesta dissertação.

As considerações finais e algumas sugestões de trabalhos futuros desta dissertação serão discutidas no Capítulo seguinte.

Capítulo 6

Conclusões

Este Capítulo apresenta as conclusões que podemos extrair dos experimentos que realizamos, bem como os trabalhos que poderão ser desenvolvidos a partir dos resultados obtidos e algumas considerações sobre a abordagem de redes neurais utilizada nesta dissertação.

6.1 Considerações Finais

O trabalho desenvolvido nesta dissertação apresentou a possibilidade de utilizar a abordagem de redes neurais auto-organizáveis como um modo de classificar os padrões de consumo de memória das aplicações inseridas em cenários de casos de uso.

A ferramenta SOM desenvolvida pode ser utilizada inicialmente em servidores ou computadores de mesa para analisar o comportamento do consumo de memória das aplicações do sistema. O administrador do sistema pode executar scripts em segundo plano para coletar as informações de consumo de memória por um determinado período e armazená-las em arquivos de log. Após isso a ferramenta SOM pode ser utilizada para visualizar a evolução do consumo de memória da aplicação monitorada.

A qualidade de resposta proporcionada pela rede neural é dependente do treinamento efetuado, logo a classificação realizada pela rede é mais precisa quando os dados oferecidos durante a fase de treinamento refletem as possibilidades de situações reais. O treinamento da rede precisa ser realizado novamente, dependendo do hardware que está sendo utilizado e dos tipos de aplicações e cenários de casos de uso que serão analisados. Portanto existe um custo de utilizar a abordagem de redes neurais para classificar padrões de consumo de memória que consiste em treinar a rede quando os requisitos do ambiente são modificados.

Os resultados apresentados no Capítulo 5 mostram que os objetivos descritos na Seção 1.3 foram concluídos com sucesso e que a abordagem implementada nesta dissertação pode

ser usada para motivar outros trabalhos a desenvolver soluções de cunho mais científico para o algoritmo de seleção do OOM Killer.

No entanto, os resultados desta dissertação não garantem que a metodologia de classificação de padrões de consumo de memória funciona satisfatoriamente no OOM Killer, pois o modelo de rede neural implementado não foi experimentado no algoritmo de seleção do OOM Killer. Sendo assim, é importante que a idéia descrita na Seção 6.2.1 seja implementada futuramente para analisar a viabilidade de aplicar o modelo de rede neural auto-organizável para a classificação de padrões de consumo de memória no algoritmo de seleção de processos do OOM Killer.

É importante averiguar também a possibilidade de utilizar a abordagem dessa dissertação em sistemas embarcados conforme a Seção 6.2.2, pois os experimentos realizados nesta dissertação foram baseados em computadores de mesa com memória RAM de 1GB, área de swap de 2GB e processador Intel Xeon de 3.06 GHz com suporte a tecnologia Hyper-Threading habilitada. Percebe-se então que o hardware utilizado neste trabalho apresenta características bem distintas de um dispositivo embarcado com restrição de memória.

6.2 Trabalhos Futuros

A explanação descrita nesta Seção serve como uma recomendação para o desenvolvimento de trabalhos futuros, apresentando a possibilidade de utilizar o mecanismo de classificação de padrão de consumo de memória baseado no SOM para o algoritmo de seleção do OOM Killer, bem como as dificuldades que poderão ser encontradas caso esta idéia seja aplicada. Além disso, a utilização da abordagem do SOM em sistemas embarcados é também sugerida como uma extensão deste trabalho.

6.2.1 Algoritmo de Seleção do OOM Killer baseado no SOM

É importante destacar que a descrição desenvolvida aqui é uma concepção teórica, onde implementações e experimentos precisam ser realizados para averiguar o funcionamento real da idéia apresentada.

O uso de SOM como uma abordagem de classificação de processos no OOM Killer necessita de uma forma de coletar os dados de entrada para a rede neural. A coleta de dados de certo processo pode ser realizada através de um *programa coletor de dados (PCD)* que monitora e armazena continuamente os dados de consumo de memória de cada

aplicação em execução. No caso do SOM, os dados relevantes são: tamanho, variação do consumo de memória e taxa de variação do consumo de memória.

O monitoramento e armazenamento dos dados de consumo de memória devem ser efetuados de um modo otimizado, visto que o acesso e a coleta contínua de dados podem causar uma sobrecarga indesejável no desempenho do sistema, principalmente quando o PCD for executado em um ambiente com pouco recurso de processamento. O PCD pode então ser iniciado durante o processo de boot do sistema operacional, pois a idéia é evitar qualquer overhead desnecessário de coleta de dados.

A maneira ideal é definir um limite similar ao ST (Signal Threshold) usado para disparar ações ou eventos, através de sinais emitidos pelo kernel, quando o consumo de memória atinge um determinado estado, conforme descrito na Seção 2.2. Neste caso o propósito deste limite, chamado de *Limite de Coleta* ou *Collect Threshold (CT)*, é justamente estabelecer o momento apropriado em que o PCD deve ser invocado para realizar a coleta e armazenamento de dados, como ilustra a Figura 6.1.

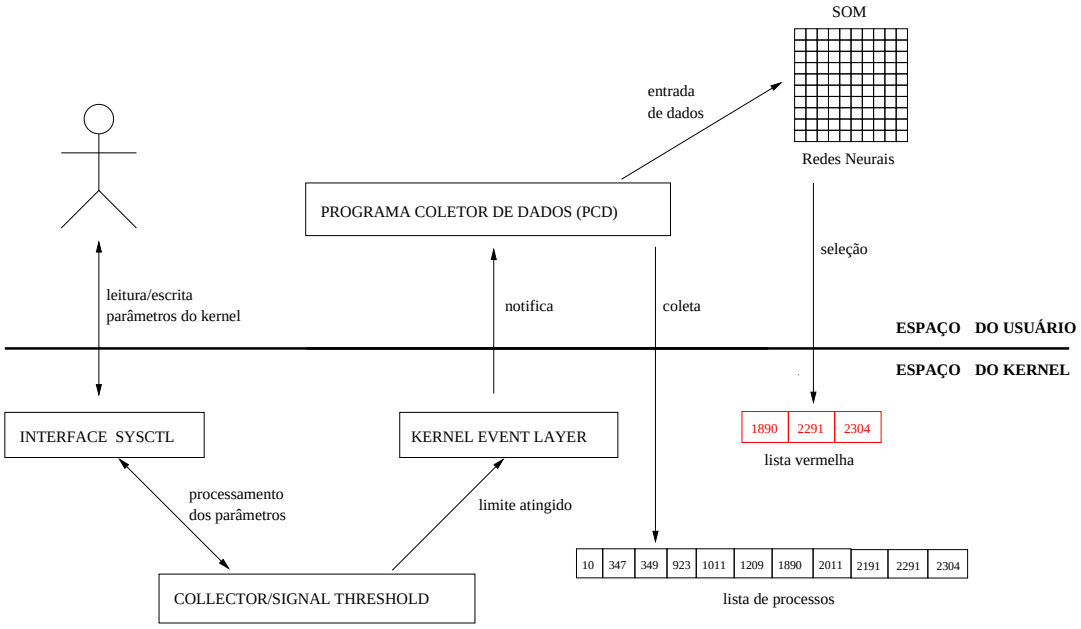

Figura 6.1: Arquitetura do OOM Killer baseado no SOM.

Os dados coletados do processo são então fornecidos para a rede neural treinada a fim verificar a classe de padrões de consumo de memória que o processo pertence. Caso o processo pertence a uma classe de padrões considerados como termináveis (killable), então este é adicionado em uma lista chamada de *lista vermelha*, que armazena os processos considerados como termináveis.

Os processos da lista vermelha são efetivamente encerrados quando um segundo limite, novamente similar ao ST, é usado para acionar o procedimento de finalização dos processos registrados na lista vermelha. Na verdade este limite pode ser o próprio ST, visto que o mesmo é usado para liberar memória do sistema, como apresentada na Seção 2.2.1. Vale ressaltar que uma vez o PCD é iniciado, o mesmo pode ser encerrado posteriormente desde que o consumo de memória retorne a um nível abaixo do limite estabelecido para o disparo do PCD. Os limites CT e ST podem ser também configuráveis pelo usuário através da interface sysctl de forma análoga a implementação da LMW apresentada na Seção 2.2.

Um aspecto importante que ocorre com essa abordagem está relacionado com o número de programas em execução que o PCD vai monitorar para coletar os dados. Para poder classificar os processos como termináveis ou não, o PCD coleta os dados de consumo de memória de todos os processos do sistema. Tal procedimento pode representar um problema de desempenho significativo para o sistema, logo um estudo de otimização é fundamental para possibilitar um monitoramento adequado dos processos.

Uma análise comparativa entre o atual funcionamento do OOM Killer e a abordagem sugerida de desenvolver uma nova forma de seleção do OOM Killer baseado no mecanismo de classificação do SOM é apresentada na Tabela 6.1.

Análise Comparativa	*OOM Killer Convencional*	*OOM Killer baseado em Redes Neurais*
Modo de Execução	No espaço do kernel	No espaço do usuário
Critério de Seleção	Tamanho da memória virtual alocada, número de processos filhos, tempo de CPU, tempo de execução, prioridade do processo e acesso à hardware.	Classes de processos localizados em regiões do mapa auto-organizável. A formação das regiões são baseadas na tupla ⟨memória, vcm, tvcm⟩.
Disparo do OOM Killer	Ocorre quando há falta de memória.	Ocorre quando o limite é atingido.
Programa Auxiliar	Nenhum	PCD é usado para coletar os dados de entrada da rede SOM.

Tabela 6.1: Análise comparativa das abordagens adotadas para o algoritmo de seleção de processos do OOM Killer.

A quantidade de memória consumida pela OOM Killer baseado no SOM pode contribuir para a condição de falta de memória, pois a sua execução é efetuada no espaço de endereçamento do usuário. Sendo assim é importante analisar o comportamento do con-

sumo de memória do próprio algoritmo do SOM para verificar a viabilidade de utilizá-lo como um mecanismo de seleção de processos do OOM Killer.

Portanto o procedimento de aplicar o SOM como mecanismo de seleção de processos pode oferecer certos problemas de desempenho, pois em situações iminentes de falta de memória, a decisão de finalizar processos precisa ser eficiente para evitar qualquer tipo de latência acentuada do sistema.

6.2.2 Aplicação do SOM em Sistemas Embarcados

A ferramenta SOM implementada nesta dissertação não foi experimentada em sistemas embarcados com restrição de memória, logo é importante proporcionar resultados dessa abordagem em tais ambientes com a finalidade de analisar o comportamento da ferramenta desenvolvida.

A realização deste trabalho em sistemas embarcados implica que a rede neural deve ser treinada novamente, pois os padrões de consumo de memória apresentados nestes sistemas podem ser diferentes. Sendo assim, os casos de uso precisam também ser aplicados em tais ambientes para coletar os dados de consumo de memória que serão utilizados no treinamento. Além disso, é recomendável usar algum mecanismo que automatiza a execução dos cenários de casos de uso para facilitar o procedimento da coleta de dados, bem como o mapeamento apropriado dos dados coletados para a entrada da rede neural.

Referências Bibliográficas

[1] Dr. Inder M. Singh. Embedded linux: The 2.6 kernel is ideal for specialized devices of all sizes. http://www.linux-mag.com/2004-08/embedded_01.html, 2004.

[2] Brandon White. Linux 2.6: A breakthrough for embedded systems. http://www.linuxdevices.com/articles/AT7751365763.html, 2003.

[3] Linux kernel mailing list. http://lkml.org/lkml/2005/1/28/99, 2005.

[4] Mauricio Lin, Ville Medeiros, Raoni Novellino, Ilias Biris, and Edjard Mota. Memory management approach for swapless embedded systems. *Linux Journal*, pages 36–43, December 2005.

[5] Andrew Tanenbaum. *Modern Operating Systems (2nd Edition)*. Prentice Hall, 2nd edition, 2001.

[6] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley, 7th edition, 2004.

[7] Robert Love. *Linux Kernel Development*. Sams Publishing, 1st edition, 2004.

[8] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly Associates, 2nd edition, 2001.

[9] CodeSourcery LLC, Mark L. Mitchell, Alex Samuel, Jeffrey Oldham, and Jeffery Oldham. *Advanced Linux Programming*. Sams, 1st edition, 2001.

[10] Andries Brouwer. The linux kernel. http://www.win.tue.nl/~aeb/linux/lk/lk-9.html, 2003.

[11] Mel Gorman. Understanding the linux virtual memory manager. http://www.csn.ul.ie/~mel/projects/vm/guide/html/understand/node84.html, 2004.

[12] Kernel Community. Kernel source tree: Documentation/vm/overcommit-accounting. ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.10.tar.bz2, 2004.

[13] Linux-kernel archive: [patch] oom killer. http://www.ussg.iu.edu/hypermail/linux/kernel/9808.2/0053.html, 1998.

[14] Linux-kernel archive: [patch] oom killer (core). http://www.ussg.iu.edu/hypermail/linux/kernel/0412.0/0039.html, 2004.

[15] Christos Stergiou and Dimitrios Siganos. Neural networks. Technical report, Department of Computing - Imperial College London. http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html.

[16] Leslie Smith. *An Introduction to Neural Networks.* Department of Computing and Mathematics - University of Stirling. http://www.cs.stir.ac.uk/~lss/NNIntro/InvSlides.html.

[17] Simon Haykin. *Redes Neurais - Princípios e prática.* Bookman, 2nd edition, 1999.

[18] Cassia Yuri Tatibana and Deisi Yuki Kaetsu. *Uma Introdução às Redes Neurais.* Departmento de Informática - Universidade Estadual de Maringá. http://www.din.uem.br/ia/neurais/.

[19] Emerson Alecrim. Redes neurais artificiais. http://www.infowester.com/redesneurais.php, 2004.

[20] Dave Anderson and George McNeil. Artificial neural networks technology. Technical report, Data & Analysis Center for Software, 1992. http://www.dacs.dtic.mil/techs/neural/neural3.html.

[21] Tom Germano. Self organizing maps. http://davis.wpi.edu/~matt/courses/soms/, 1999.

[22] Ai-Junkie. Kohonen's self organizing feature maps. http://www.ai-junkie.com/ann/som/som1.html.

[23] Timo Honkela. *Self-Organizing Maps in Natural Language Processing.* PhD thesis, Helsinki University of Technology - Neural Networks Research Centre, P.O. Box 2200 FIN-02015 HUT, FINLAND, 1997. http://www.mlab.uiah.fi/~timo/som/thesis-som.html.

[24] Christian Borgelt. Self-organizing map training visualization. http://fuzzy.cs.uni-magdeburg.de/~borgelt/doc/somd/, 2000. School of Computer Science - Otto-von-Guericke-University of Magdeburg.

[25] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to The Theory of Neural Computation*. Addison-Wesley, 1991.

[26] P. Poinçot, S. Lesteven, and F. Murtagh. A spatial user interface to the astronomical literature. 1997.

[27] Jorma Laaksonen, Ville Viitaniemi, and Markus Koskela. Application of self-organizing maps and automatic image segmentation to 101 object categories database. In *Proceedings of Fourth International Workshop on Content-Based Multimedia Indexing*, 2005.

[28] Zhirong Yang and Jorma Laaksonen. Interactive retrieval in facial image database using self-organizing maps. In *Proceedings of IAPR Conference on Machine Vision Applications*, 2005.

[29] Edjard Mota and Mauricio Lin. Classes of memory allocation. Technical report, Instituto Nokia de Tecnologia - INdT, 2005.

[30] GNU Free Documentation License 1.2. Gnu / linux desktop testing project. http://gnomebangalore.org/ldtp/index.php/Main_Page, 2005.

[31] Tony Gale, Ian Main, and the GTK team. Gtk+ 2.0 tutorial. http://www.gtk.org/tutorial/.

[32] Kardi Teknomo. Similarity measurement. http://people.revoledu.com/kardi/tutorial/similarity/, 2005.