ALAN FILIPE SANTANA

TREINAMENTO DE REDES NEURAIS ARTIFICIAIS UTILIZANDO ALGORITMOS GENÉTICOS EM PLATAFORMA DISTRIBUÍDA

LAVRAS – MG
2012

ALAN FILIPE SANTANA

TREINAMENTO DE REDES NEURAIS ARTIFICIAIS UTILIZANDO ALGORITMOS GENÉTICOS EM PLATAFORMA DISTRIBUÍDA

Monografia apresentada ao Colegiado do Curso de Sistemas de Informação, como uma das exigências para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador:
Dr. Wilian Soares Lacerda

LAVRAS – MG
2012

ALAN FILIPE SANTANA

TREINAMENTO DE REDES NEURAIS ARTIFICIAIS UTILIZANDO ALGORITMOS GENÉTICOS EM PLATAFORMA DISTRIBUÍDA

Monografia de graduação apresentada ao Colegiado do Curso de Sistemas de Informação, para obtenção do título de Bacharel em Sistemas de Informação.

APROVADA em 10 de outubro de 2012.

RAPHAEL WINCKLER DE BETTIO

BRUNO HENRIQUE GROENNER BARBOSA

WILIAN SOARES LACERDA (orientador/a)

LAVRAS-MG

2012

Aos meus pais Geraldo e Valéria por todo o amor e dedicação.

A meu irmão, Antônio Rafael pela amizade e companheirismo.

DEDICO

AGRADECIMENTOS

À Universidade Federal de Lavras (UFLA) e ao Departamento de Ciência da Computação, pela oportunidade concedida para realização da graduação.

Ao professor Dr. Wilian Soares Lacerda pela orientação, paciência e ensinamentos que foram essenciais para o desenvolvimento deste trabalho.

RESUMO

Redes Neurais Artificiais são algoritmos inspirados no funcionamento do sistema nervoso que apresentam várias aplicações como classificação de dados e previsão de valores. Seu funcionamento depende de um processo de treinamento que, em alguns problemas, pode consumir considerável tempo de processamento. Este trabalho teve como objetivo avaliar a utilização de um modelo paralelo de Algoritmo Genético para realizar o treinamento de redes do tipo Multilayer Perceptron. A análise foi dividida em duas etapas: avaliação de duas variações de algoritmos genéticos a partir de comparações dos resultados obtidos com o algoritmo de aprendizagem Backpropagation; e avaliação do sistema distribuído através de testes com a utilização de até cinco computadores. Para a realização dos testes foram utilizados seis bases de dados do repositório de aprendizagem de máquina UCI. Foi observado que o Algoritmo Genético possui desempenho inferior ao Backpropagation, entretanto é capaz de encontrar soluções com boa capacidade de generalização. Com a utilização do sistema distribuído a redução média total do tempo de treinamento foi de 31%. Apesar do baixo ganho em desempenho, o modelo paralelo aumenta a estabilidade do processo de aprendizagem evolutivo.

Palavras chaves: Redes Neurais Artificiais. Treinamento. Algoritmo Genético. Sistema distribuído.

ABSTRACT

Artificial Neural Networks are algorithms inspired in the functioning of the nervous system that have many applications such as data classification and time series prediction. Its use depends on a training process which, in some problems, can consume significant processing time. This study aimed to evaluate the use of a parallel model of Genetic Algorithm to execute the training of Multilayer Perceptron networks. The analysis was performed by two steps: evaluation of two types of genetic algorithms by comparison among results obtained by Backpropagation learning algorithm; and evaluation of the distributed system by tests running on up to five computers. For the tests, six databases from UCI machine learning repository were used. The Genetic Algorithm has underperformed Backpropagation; however it was able to find solutions with good generalization capacity. The total average training time was reduced to 31% when the distributed system was applied. Despite the low gain in performance, the parallel model increases the stability of the evolutionary learning process.

Keywords: Artificial Neural Networks. Training. Genetic Algorithm. Distributed System.

LISTA DE FIGURAS

Figura 1 Representação simplificada de um neurônio biológico 16

Figura 2 Modelo do neurônio artificial 17

Figura 3 Exemplos de funções de ativação: (a) função linear, (b) função threshold, (c) função sigmoidal 18

Figura 4 Rede feedforward de única camada... 19

Figura 5 Rede feedforward com uma camada escondida 20

Figura 6 Exemplo de rede recorrente 20

Figura 7 Diagrama de bloco do processo de aprendizagem supervisionado 21

Figura 8 Diagrama de bloco do processo de aprendizagem não supervisionado 22

Figura 9 Pseudocódigo de um algoritmo evolucionário padrão 30

Figura 10 Posicionamento dos AEs como técnica de busca. 31

Figura 11 Ilustração do operador *crossover* de dois pontos 34

Figura 12 Exemplo de operação do *crossover flat*... 35

Figura 13 Ilustração do operador de mutação em uma representação binária.... 36

Figura 14 Ciclo geral da evolução dos pesos de uma RNA.............................. 38

Figura 15 Exemplo de representação binária de uma RNA.............................. 39

Figura 16 Framework combinando BP com AG 40

Figura 17 Ciclo geral da evolução de arquiteturas de RNAs. 42

Figura 18 Modelo cliente-servidor.. 44

Figura 19 Modelo *peer-to-peer*... 45

Figura 20 Ilustração da abordagem mestre-escravo... 48

Figura 21 Topologia de migração em anel 49

Figura 22 Modelo de vizinhança utilizando uma estrutura em grade 50

Figura 23 Exemplo de modelos híbridos.. 51

Figura 24 Esquema simplificado da execução de um programa Java 52

Figura 25 Ilustração de uma conexão por meio de sockets TCP 53

Figura 26 Ilustração do modelo adotado na implementação do sistema distribuído 55

Figura 27 Diagrama de classe da implementação de uma MLP 56
Figura 28 Diagrama de classe da implementação dos algoritmos de aprendizagem 57
Figura 29 Diagrama de classe da implementação do modelo de ilhas 59
Figura 30 Estrutura das mensagens utilizadas no sistema distribuído................ 60
Figura 31 Ilustração do processo de formação de conexões.............................. 61

LISTA DE GRÁFICOS

Gráfico 1 Evolução dos EQMs para classificação da flor íris 71

Gráfico 2 Evolução do EQM para classificação do câncer de mama................. 73

Gráfico 3 Previsão dos valores de habitação fornecidos pelo melhor treinamento utilizando BP 75

Gráfico 4 Previsão dos valores de habitação fornecidos pelo melhor treinamento utilizando AGB... 76

Gráfico 5 Previsão dos valores de habitação fornecidos pelo melhor treinamento utilizando AGR... 76

Gráfico 6 Evolução do EQM para a previsão de valores de habitação 77

Gráfico 7 Evolução do tempo médio de treinamento em relação ao número de ilhas para resolução do problema abalone.. 80

Gráfico 8 Evolução do tempo médio de treinamento em relação ao número de ilhas para resolução do problema splicing 81

Gráfico 9 Evolução do tempo médio de treinamento em relação ao número de ilhas para resolução do problema landsat.. 83

LISTA DE QUADROS

Quadro 1 Exemplos de funções de ativação 18

Quadro 2 Aplicações comuns para algoritmos genéticos 37

Quadro 3 AGs utilizados no treinamento da MLP para classificação da flor íris 70

Quadro 4 AGs utilizados no treinamento da MLP para classificação do câncer de mama.. 72

Quadro 5 AGs utilizados no treinamento da MLP para classificação do câncer de mama.. 74

Quadro 6 Parâmetros do AGR utilizados no problema abalone 79

Quadro 7 Parâmetros do AGR utilizado no problema splicing 81

Quadro 8 Parâmetros do AGR utilizado no problema landsat........................... 82

LISTA DE TABELAS

Tabela 1 Média dos EQMs obtidos nos treinamentos para classificação da flor íris 71

Tabela 2 Média dos acertos obtidos nos treinamentos para classificação da flor íris 72

Tabela 3 Média dos EQMs obtidos nos treinamentos para classificação do câncer de mama 73

Tabela 4 Média dos acertos obtidos nos treinamentos para classificação do câncer de mama 74

Tabela 5 Média dos EQMs obtidos nos treinamentos para previsão de valores de habitação 75

Tabela 6 Resultados do modelo paralelo aplicado no problema abalone 79

Tabela 7 Resultados do modelo paralelo aplicado no problema splicing 81

Tabela 8 Resultados do modelo paralelo aplicado no problema landsat 82

SUMÁRIO

1 INTRODUÇÃO 12
1.1 Contextualização e motivação 12
1.2 Objetivos 13
1.3 Estrutura do trabalho... 13
2 REFERENCIAL TEÓRICO 15
2.1 Redes neurais artificiais 15
2.1.1 Inspiração biológica.. 15
2.1.2 Modelo do neurônio artificial 16
2.1.3 Arquiteturas de rede neurais.. 19
2.1.4 Processo de aprendizagem.. 21
2.1.5 Regra delta.. 22
2.1.6 Perceptron 23
2.1.7 Multilayer Perceptron 24
2.1.8 Backpropagation 24
2.1.9 Considerações sobre o treinamento de MLPs..................................... 26
2.1.10 Aplicações 28
2.2 Algoritmos genéticos... 29
2.2.1 Representação dos indivíduos 31
2.2.2 Função de avaliação.. 32
2.2.3 Seleção de pais 33
2.2.4 Operador de recombinação 34
2.2.5 Operador de mutação.. 35
2.2.6 Módulo de população 36
2.2.7 Aplicações 36
2.3 Redes neurais artificiais evolutivas 37
2.3.1 Evolução dos pesos sinápticos 38
2.3.2 Evolução da arquitetura de rede 41
2.3.3 Trabalhos relacionados 42
2.4 Sistemas distribuídos... 43
2.4.1 Modelos de arquiteturas.. 44
2.4.2 Comunicação entre processos na Internet 45
2.5 Paralelização de algoritmos genéticos 46
2.5.1 Panmictic 47
2.5.2 Modelo de ilhas 48
2.5.3 Modelo de vizinhança 49
2.5.4 Modelos híbridos 50
2.6 Java 51
2.7 Sockets 53
3 MATERIAL E MÉTODOS... 54

3.1 Modelo de paralelização 54
3.2 Estrutura de classes 55
3.2.1 Implementação da MLP 56
3.2.2 Módulo de treinamento 57
3.2.3 Implementação do modelo de ilhas 58
3.3 Protocolo de comunicação 59
3.4 Algoritmos de aprendizagem 61
3.4.1 AG com representação binária 62
3.4.2 AG com representação real 63
3.4.3 Backpropagation 66
3.5 Procedimentos metodológicos 66
3.6 Ambiente computacional 68
4 RESULTADOS E DISCUSSÃO 69
4.1 Testes dos algoritmos de aprendizagem 69
4.1.1 Classificação da flor íris 70
4.1.2 Classificação do câncer de mama 72
4.1.3 Previsão de valores de habitação 74
4.1.4 Análise dos resultados 77
4.2 Teste do modelo distribuído 78
4.2.1 Previsão da idade do abalone 78
4.2.2 Classificação da junção de splicing 80
4.2.3 Classificação de imagens multiespectrais 82
4.2.4 Análise dos resultados 83
5 CONCLUSÃO 84
REFERÊNCIAS 85
APÊNDICE A – Implementação da RNA 88
APÊNDICE B – Implementação do Backpropagation 96
APÊNDICE C – Implementação do AGR 103
APÊNDICE D – Implementação da classe Aplicação 110

1 INTRODUÇÃO

Este capítulo apresenta o assunto tratado neste trabalho mostrando os principais aspectos relativos ao seu contexto, motivação e objetivos. No final do capitulo é apresentada a organização do conteúdo utilizada neste documento.

1.1 Contextualização e motivação

As redes neurais artificiais (RNAs) constituem uma técnica computacional inspirada no funcionamento do sistema nervoso e das suas unidades. A técnica pertence a uma classe da inteligência artificial aplicada na resolução de problemas como classificação, regressão, aproximação de função e reconhecimento de padrões.

A utilização de RNAs envolve a definição de uma topologia de rede e um processo de aprendizagem, na qual seus parâmetros são ajustados de acordo com estímulos do ambiente. Dependendo do problema tratado, este processo pode demandar uma grande quantidade de tempo e recurso computacional o que desestimula a utilização da técnica.

Com o objetivo de aumentar a eficiência do treinamento das redes neurais foram desenvolvidas versões paralelas do algoritmo de aprendizagem. Segundo Foo, Saratchandran e Sundararajan (1997), existem principalmente dois paradigmas para o mapeamento de uma rede neural em uma plataforma paralela: o paralelismo baseado na rede e o paralelismo do conjunto de treinamento. O paralelismo baseado em rede é caracterizado pela distribuição de partes da RNA para os processadores e, como consequência, apresenta alto grau de comunicação entre os processos. Já o paralelismo baseado no conjunto de treinamento apresenta menos trocas de mensagens, entretanto utiliza o algoritmo

de aprendizado no modo *batch* o que pode acarretar em uma convergência global mais lenta.

Em vários trabalhos foram propostos sistemas híbridos para realizar o treinamento de redes neurais na tentativa de combinar as melhores características de diferentes algoritmos. Neste contexto, a utilização de Algoritmos Genéticos (AGs) juntamente com RNAs constitui parte da classe de redes neurais evolutivas (RNAE) (YAO, 1999). Esta combinação possibilita a adoção de novas alternativas para a paralelização do algoritmo de aprendizagem, motivando o desenvolvimento deste trabalho.

1.2 Objetivos

Este trabalho tem como principal objetivo desenvolver um sistema distribuído para realizar o treinamento de redes neurais artificiais do tipo Multilayer Perceptron (MLP) utilizando um algoritmo genético paralelo. Com a implementação espera-se reduzir o tempo do processo de treinamento e avaliar os benefícios da utilização de algoritmos genéticos para a evolução dos pesos de uma rede neural.

1.3 Estrutura do trabalho

O conteúdo deste trabalho está organizado em quatro capítulos além deste.

O capítulo 2 introduz temas importantes para o desenvolvimento e compreensão do trabalho. Os assuntos tratados são: redes neurais artificiais, algoritmos genéticos, redes neurais artificiais evolutivas, sistemas distribuídos, modelos de paralelização de algoritmos genéticos, linguagem de programação Java e utilização de sockets.

O capítulo 3 apresenta a metodologia de desenvolvimento adotada na implementação e nos testes do sistema distribuído. Os resultados são apresentados e discutidos no capítulo 4. O capítulo 5 concluiu o trabalho, considerando as análises realizadas e possibilidades para trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este capítulo apresenta os conceitos básicos de temas e assuntos relacionados com o desenvolvimento deste trabalho.

2.1 Redes neurais artificiais

A capacidade do cérebro humano de realizar tarefas complexas, como o reconhecimento de padrões e o processo de aprendizagem, inspirou o desenvolvimento de modelos matemáticos dos sistemas neurais biológicos. Estes modelos fundamentaram técnicas computacionais de inteligência artificial que constituem as redes neurais artificiais.

Segundo Haykin (2008) uma rede neural artificial pode ser definida como um processador maciçamente paralelo e distribuído formado de simples unidades capazes de armazenar conhecimento baseado em experiência. As redes neurais possuem uma importante capacidade de aprender a generalizar a solução de um problema a partir de um conjunto específico de exemplos (BISHOP, 1994). Essa capacidade permite a resolução de problemas que não podem ser expressos em uma simples série de passos.

2.1.1 Inspiração biológica

A unidade construtora de uma rede neural é um modelo simplificado do que é suposto ser o comportamento funcional de um neurônio orgânico (VEELENTURF, 1995, p. 2). De maneira simplificada, um neurônio pode ser dividido em quatro regiões: dendritos, corpo celular, axônio e terminações do axônio.

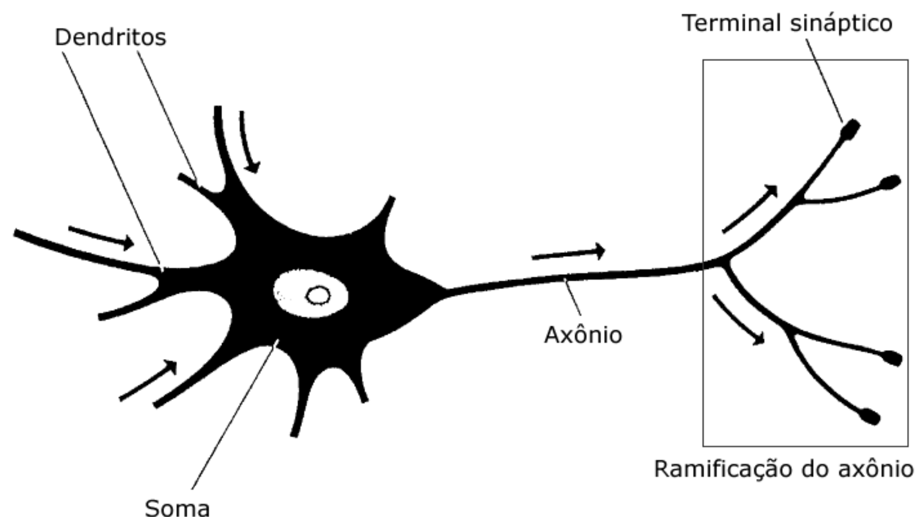

Figura 1 Representação simplificada de um neurônio biológico
Fonte: Adaptado de Arbib (2003)

A partir do corpo celular projetam-se fibras chamadas dendritos que juntos constituem a superfície de entrada do neurônio. Também, a partir do corpo celular existe uma ramificação, chamada axônio, que transmite um sinal do corpo celular para suas extremidades. As terminações do axônio são conectadas com os dendritos de outros neurônios pelas sinapses. Pulsos elétricos gerados pelos neurônios são transmitidos ao longo do axônio para as sinapses. Estes pulsos podem contribuir para a excitação ou inibição de outros neurônios. Segundo Veelenturf (1995), as sinapses desempenham um importante papel nas transmissões de pulsos elétricos uma vez que a eficiência destas transmissões pode ser alterada conforme a *rentabilidade* da mudança. Esta mudança, em função de sinais externos, representa o processo de aprendizagem.

2.1.2 Modelo do neurônio artificial

Em 1943, McCulloch e Pitts publicaram um artigo que introduziu o modelo do neurônio artificial. Segundo Arbib (2003), o trabalho combinou

neurofisiologia e lógica matemática, usando a propriedade *tudo-ou-nada* da transmissão de sinais elétricos do neurônio, para construir um modelo de um elemento binário. A partir da generalização deste modelo podem-se identificar três elementos básicos, conforme apresentado por Haykin (2008):

1. Um conjunto de sinapses caracterizadas por seu respectivo peso.
2. Um somatório dos sinais de entradas ponderados pelos pesos das respectivas sinapses.
3. Uma função de ativação para limitar a amplitude da saída do neurônio.

Cada neurônio, considerado como uma unidade de processamento, calcula a soma ponderada de suas entradas mais um polarizador (*bias*) e aplica o resultado em uma função de ativação que poderá produzir ou não um sinal de saída. A Figura 2 ilustra o modelo a partir de um diagrama de bloco.

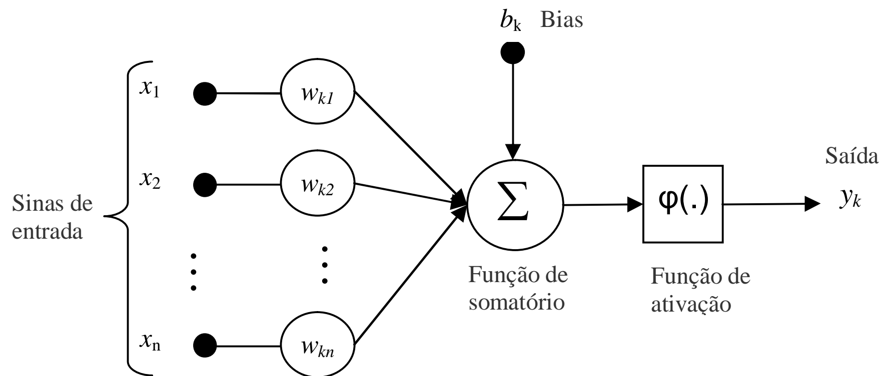

Figura 2 Modelo do neurônio artificial

Matematicamente, o neurônio pode ser descrito pela equação:

$$y_k = \varphi\left(\sum_{j=1}^{n} w_{kj} x_j + b_k\right) \tag{1}$$

onde, n é o número de entradas do neurônio k, x_j é a entrada j do neurônio, w_{kj} é o peso correspondente à entrada j do neurônio k, b_k corresponde ao parâmetro polarizador, φ é a função de ativação e y_k é a saída do neurônio k.

O parâmetro *bias* também pode ser representado pelo peso sináptico de uma entrada cujo valor é fixo em 1. Exemplos de funções de ativação comumente utilizadas são exibidos no Quadro 1 e na Figura 3.

O modelo original de McCulloch-Pitts utilizou a função *threshold* mostrada na Figura 3(b). Segundo Bishop (1994), a maioria das redes neurais de interesse prático utiliza a função sigmoidal mostrada na Figura 3(c).

Quadro 1 Exemplos de funções de ativação

Funções de ativação		
Nome	**Fórmula** $f(u)$	**Derivada** $\frac{\partial f(u)}{\partial u}$
Sigmóide	$f(u) = \frac{1}{1 + \ell^{-u/T}}$	$f(u)\left[1 - f(u)\right]/T$
Threshold	$f(u) = \begin{cases} 1 \rightarrow u > 0 \\ 0 \rightarrow u \leq 0 \end{cases}$	-
Linear	$f(u) = au + b$	a

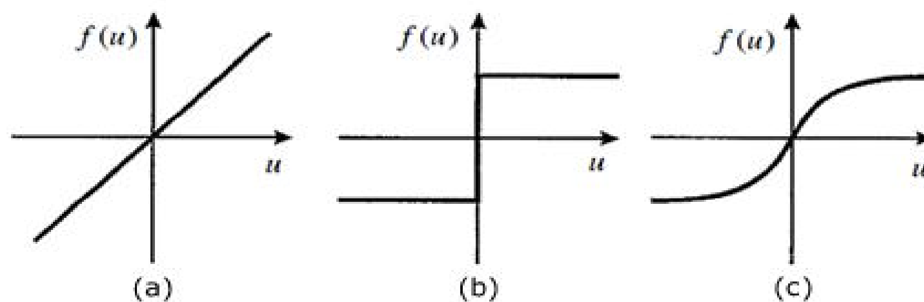

Figura 3 Exemplos de funções de ativação: (a) função linear, (b) função threshold, (c) função sigmoidal

2.1.3 Arquiteturas de rede neurais

Uma rede neural pode ser representada por um grafo direcionado, onde o nó representa um neurônio e as arestas representam as entradas ou a saída de um neurônio. A forma como os neurônios são conectados define diferentes topologias de rede. De maneira geral pode-se identificar três classes de arquiteturas de redes neurais (HAYKIN, 2009, p. 21):

1. Redes feedforward de única camada: nesta classe, os neurônios são organizados em uma única camada que produzem diretamente a saída da rede (Figura 4). O termo feedforward indica o sentido único de propagação dos sinais, não existindo ciclos na camada.

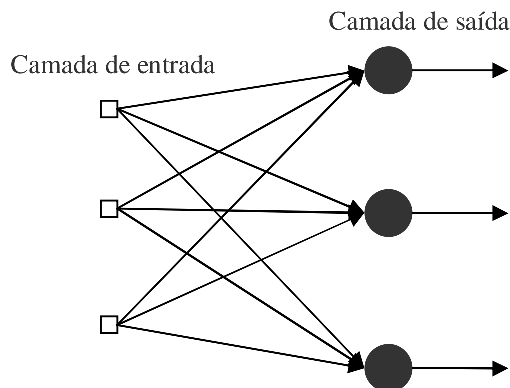

Figura 4 Rede feedforward de única camada

2. Redes feedforward de múltiplas camadas: existem uma ou mais camadas de neurônios escondidas (Figura 5). Conforme descreve Haykin (2009), o termo escondido refere-se ao fato de que esta parte da rede não é diretamente conectada a saída da rede. A camada escondida alimentada pelos sinais de entrada provê o vetor de entrada da camada seguinte, propagando sinais até a camada de saída da rede. A camada escondida permite a extração de estatísticas de ordem superior.

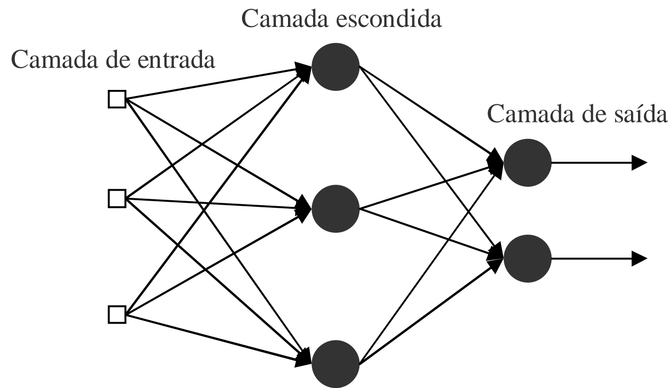

Figura 5 Rede feedforward com uma camada escondida

3. Redes recorrentes: esta classe de arquitetura é caracterizada pela presença de pelo menos um laço de realimentação. Segundo Haykin (2009), este laço causa um profundo impacto na capacidade de aprendizado da rede e em sua performance. Este laço envolve a utilização de um ramo particular que produz um atraso de tempo de uma unidade (Figura 6).

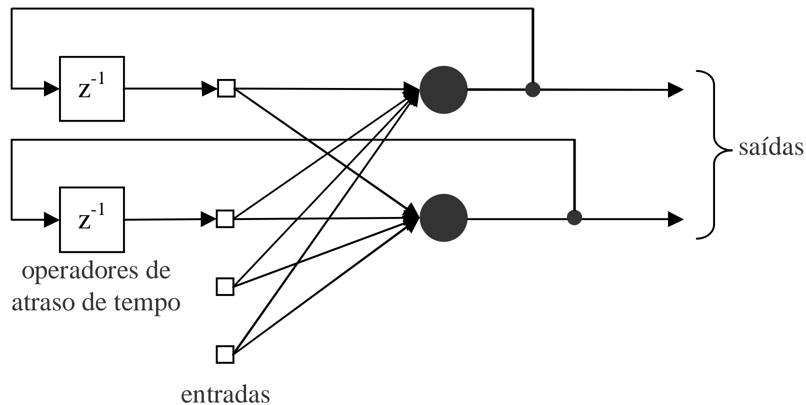

Figura 6 Exemplo de rede recorrente

2.1.4 Processo de aprendizagem

Basicamente, o processo de aprendizagem envolve a adaptação dos pesos da rede neural a partir de estímulos fornecidos por um conjunto de dados. Segundo Rezende (2005, p. 149): “o objetivo do processo de aprendizagem é encontrar o ajuste do vetor de pesos Δw para que o objetivo do treinamento que visa à convergência seja atingido”. Dependendo da maneira como o ajuste é obtido, o processo pode ser classificado como: aprendizado supervisionado, aprendizado não supervisionado e aprendizado por reforço.

O aprendizado supervisionado é caracterizado pela presença de um tutor que indica, a partir de mapeamentos de entradas e saídas desejadas o erro da rede. Os pares de entrada e saída constituem o conjunto de treinamento que, em termos conceituais, representa o conhecimento do tutor a respeito do ambiente (Haykin, 2009). Desta maneira, é possível utilizar o erro calculado e o vetor de entrada na obtenção de Δw. Esta forma de treinamento é ilustrada na Figura 7.

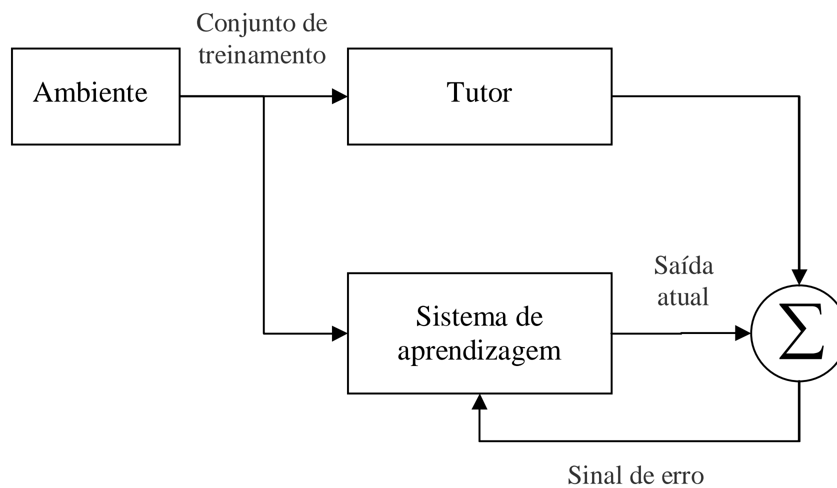

Figura 7 Diagrama de bloco do processo de aprendizagem supervisionado

No aprendizado não supervisionado não há o conjunto de exemplos formado por pares de entrada e saída desejada, consequentemente, o ajuste dos pesos é realizado com base nas regularidades estatísticas dos dados fornecidos à rede (Figura 8). Esta forma de treinamento permite a utilização de redes neurais para categorização de dados.

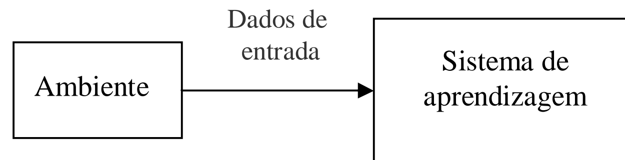

Figura 8 Diagrama de bloco do processo de aprendizagem não supervisionado

O aprendizado por reforço pode ser considerado um paradigma intermediário entre o aprendizado supervisionado e o não supervisionado (REZENDE, 2005). O treinamento é realizado com base nos sinais de reforço ou penalização fornecidos por um *crítico* externo a rede. Desta forma o algoritmo de aprendizado visa à maximização da probabilidade de reforços positivos para um dado conjunto de treinamento.

Este trabalho tem como foco o treinamento de redes feedforward de múltiplas camadas utilizando um processo de aprendizado supervisionado.

2.1.5 Regra delta

Widrow e Hoff desenvolveram em 1960 um processo de aprendizagem por correção de erro denominado *regra delta*. Neste processo, o erro é obtido através da equação:

$$e(n) = d(n) - y(n) \quad (2)$$

onde n representa uma amostra do conjunto de treinamento, $d(n)$ o valor desejado e $y(n)$ o valor fornecido por um neurônio. Os ajustes são aplicados de forma a minizar a função custo:

$$\xi(n) = \frac{1}{2} e^2(n) \tag{3}$$

que representa o valor da energia do erro no instante n. Dessa forma, o ajuste num peso w_j no passo n é definido pela equação:

$$\Delta w_j(n) = \eta e(n) x_j(n) \tag{4}$$

onde η é uma constante positiva que representa a taxa de aprendizado do algoritmo e $x_j(n)$ é o sinal de entrada j da amostra n.

2.1.6 Perceptron

O perceptron, proposto em 1958 por Rosenblatt, foi o primeiro modelo de aprendizagem supervisionada de redes neurais. Construído em torno do modelo de neurônio de McCulloch-Pits tinha como objetivo classificar um conjunto de estímulos em uma dentre duas classes.

O ajuste dos pesos do perceptron é dado pela equação:

$$w(n+1) = w(n) + \eta[d(n) - y(n)]x(n) \tag{5}$$

onde n representa um passo de iteração do processo de ajuste dos pesos sinápticos, $w(n)$ é o vetor de pesos corrente, $d(n)$ é o valor desejado como saída, $y(n)$ é o valor fornecido pela rede, $x(n)$ é o vetor de entrada, $w(n + 1)$ corresponde ao vetor de pesos atualizado e η é a taxa de aprendizado cujo valor deve estar entre 0 e 1.

Rosenblatt demonstrou que se os padrões utilizados para treinar o perceptron pertencerem a duas classes linearmente separáveis, o algoritmo de

aprendizado proposto converge e posiciona a superfície de decisão na forma de um hiperplano entre as duas classes.

2.1.7 Multilayer Perceptron

A utilização de perceptrons de camada única permite apenas a resolução de problemas linearmente separáveis. Esta limitação foi matematicamente provada em 1969 por Minsky e Papert que também sugeriram a utilização de mais de uma camada de neurônios. Entretanto, conforme comenta Bishop (1994), esta arquitetura não era amplamente considerada devido à dificuldade em se encontrar um algoritmo de treinamento adequado. A solução deste problema foi a utilização de funções de ativação diferenciáveis o que contribuiu com a descoberta de novos algoritmos de aprendizado e o desenvolvimento das redes feedfoward de múltiplas camadas, também chamadas de Multilayer Perceptrons. Haykin (2009) destaca três características básicas das redes multilayer perceptrons:

1. O modelo de cada neurônio da rede inclui uma função de ativação diferenciável.
2. A rede contém uma ou mais camadas escondidas.
3. A rede apresenta um alto grau de conectividade

O método comumente utilizado para o treinamento de uma MLP é o algoritmo backpropagation, apresentado na seção seguinte.

2.1.8 Backpropagation

O backpropagation (BP) é um algoritmo de aprendizagem supervisionada baseada na generalização da regra delta onde se busca a minimização da função do erro quadrático representada por:

$$E(n) = \frac{\sum_{j=1}^{k}(d_j(n) - y_j(n))^2}{2} \quad (6)$$

onde n é a época, k é o número de neurônios da camada de saída, d_j é o valor desejado do neurônio j e y_j é a saída fornecida pelo neurônio.

Segundo Haykin (2009), o processo de treinamento é constituído de duas fases:

1. Na fase *forward* (para frente) ocorre a propagação do sinal de entrada ao longo da rede.
2. Na fase *backward* (para trás) um sinal de erro é produzido através da comparação da saída obtida com a saída desejada. Este sinal é propagado *para trás* ao longo da rede. Nesta fase ocorre a correção dos pesos.

O algoritmo pode ser resumido nos seguintes passos:

1. Inicialização dos pesos sinápticos com valores aleatórios de distribuição uniforme.
2. Para cada padrão de entrada realiza-se a propagação do sinal por meio da equação:

$$y_k = \varphi\left(\sum_{j=0}^{n} w_{kj} x_j\right) \quad (7)$$

que é similar a Equação (1) considerando que w_{k0} e x_0 representam o valor do *bias* e o valor 1 respectivamente.

3. Calcula-se o erro de cada unidade de saída através da equação:

$$e_j(n) = d_j(n) - y_j(n) \quad (8)$$

4. Calculam-se os gradientes locais da rede, sendo que para a última camada, utiliza-se a equação:

$$\delta_j(n) = e_j(n)\varphi_j^{'}(v_j(n)) \quad (9)$$

onde $v_j(n)$ é dado pela equação:

$$v_j(n) = \sum_{j=1}^{m} w_{kj}(n) x_j(n) \tag{10}$$

para as camadas escondidas o gradiente é dado por:

$$\delta_j^{(l)}(n) = \varphi_j^{'}(v_j^{(l)}(n)) \sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) \tag{11}$$

onde l representa a camada e k o número de neurônios da camada seguinte. Os pesos são ajustados de acordo com a regra delta generalizada:

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha w_{ji}^{(l)}(n-1) + \eta \delta_j^{(l)}(n) y_j^{(l-1)}(n) \tag{12}$$

onde η é a taxa de aprendizado e α é a constante momento que é aplicada sobre a variação anterior obtida a fim de acelerar a convergência da rede além de provocar um efeito estabilizante no treinamento.

5. Enquanto o critério de parada adotado não for atendido, repete-se o processo a partir do passo 2, apresentando uma nova época de exemplos de treinamento.

Exemplos de critérios de parada incluem: um número máximo de épocas, um erro mínimo desejado e uma medida da capacidade de generalização da rede.

2.1.9 Considerações sobre o treinamento de MLPs

Basicamente, o treinamento de uma MLP busca minimizar uma função de erro e maximizar a capacidade de generalização da rede. Conforme define Haykin (2009), a capacidade de generalização da rede é verificada quando o mapeamento de entrada-saída calculado pela rede está correto (ou pelo menos próximo do resultado correto) para um conjunto de teste nunca utilizado no processo de aprendizado.

De forma geral, pode-se resumir o processo de treinamento de uma MLP nos seguintes passos:

- Processamento dos dados: os dados de saída devem estar dentro dos limites da função de ativação escolhida para a camada de saída. Além disso, conforme LeCun (apud HAYKIN, 2009, p. 208), a normalização dos dados de entrada pode aumentar a velocidade do processo de aprendizagem.
- Definição da rede: escolha do número de camadas, da quantidade de neurônios presente em cada camada e da função de ativação a ser utilizada.
- Configuração de parâmetros: dependendo do algoritmo de aprendizado escolhido haverá diferentes parâmetros a serem configurados. No caso de se utilizar o backpropagation, é necessário configurar a taxa de aprendizado e a taxa de momento. Estes parâmetros podem ser modificados durante o processo de aprendizagem.
- Escolha do critério de parada.
- Execução do algoritmo de aprendizagem: além do backpropagation existem outros algoritmos de aprendizagem. A sessão 2.3.1 apresenta a utilização de algoritmos genéticos para realizar o ajuste dos pesos da rede neural.
- Avaliação da rede: avalia-se capacidade de generalização da rede. Para isto, pode-se utilizar a técnica de validação cruzada onde os dados disponíveis são divididos em subconjuntos distintos, sendo que um é utilizado na estimação do modelo e o restante para realizar a validação e teste do modelo.

O ajuste dos pesos pode ser realizado de forma *on-line*, após a apresentação de cada padrão de entrada, ou do modo *batch*, na qual os ajustes são acumulados até que todo conjunto de treinamento tenha sido apresentado.

Segundo Haykin (2009) o modo *batch* permite a paralelização do processo de aprendizado, e a estimativa mais precisa do vetor de gradiente. Já o modo *on-line* é mais simples de implementar e provê soluções efetivas para problemas difíceis de classificação de padrões.

Pode ser necessária a repetição dos passos apresentados até a determinação de uma rede ideal. Desta forma, dependendo do problema tratado, o processo de treinamento pode consumir grande quantidade de tempo.

2.1.10 Aplicações

Existem problemas de engenharia (e outras áreas), para os quais encontrar a perfeita solução requer uma quantidade de recursos que em termos práticos não é aceitável. Para esta classe de problemas as RNAs podem fornecer boas soluções (MEIRELES; ALMEIDA; SIMÕES, 2003, p. 587).

Heaton (2008), lista quatro tipos de problemas frequentemente resolvidos pelas RNAs:

- Classificação: representa a tarefa de classificar um padrão de entrada em grupos.
- Previsão: a partir de uma série temporal, uma RNA pode ser utilizada para prever valores futuros. Redes neurais são comumente utilizadas nas previsões de valores do mercado financeiro.
- Reconhecimento de padrões: esta tarefa pode ser definida como o processo pelo qual um dado padrão de entrada é associado a uma classe dentre várias predefinidas.
- Otimização: na qual se procura maximizar ou minimizar uma determinada função relacionada com o problema.

As RNAs permitem a extração de padrões em conjuntos de dados, esta capacidade motiva o seu uso em aplicações industriais e de mineração de dados.

“Um dos aspectos fascinantes, da aplicação prática das RNAs em sistemas industriais, é a capacidade de gerenciar a interação entre os dados elétricos e o comportamento mecânico e, muitas vezes, outras disciplinas”. (MEIRELES; ALMEIDA; SIMÕES, 2003, p. 598, tradução nossa).

Outras utilizações de RNAs são apresentadas por Bishop (1994), como a resolução de problemas inversos, controle de aplicações e aproximação de funções.

2.2 Algoritmos genéticos

Algoritmos genéticos é um ramo dos algoritmos evolucionários (AE) que são métodos estocásticos de busca baseados nos processos naturais de evolução. A história dos algoritmos evolucionários se inicia na década de 40, quando pesquisadores começam a se inspirar na natureza para criar o ramo da inteligência artificial (LINDEN, 2006, p. 32). Eiben e Smith (2003) citam três trabalhos da década de 60 relacionados com o desenvolvimento da computação evolucionária: Fogel, Owens e Walsh que introduziram a programação genética, Holland que chamou o seu método de algoritmo genético, e Rechenberg e Schwefel que inventaram as estratégias evolucionárias. Em 1975 Holland publicou o livro *Adaptation in Natural and Artifical Systems*, no qual foi apresentada a teoria por trás dos algoritmos genéticos e suas aplicações práticas.

Conforme descreve Linden (2006, p. 37):

> Os algoritmos evolucionários funcionam mantendo uma população de estruturas denominadas indivíduos ou cromossomos, que se comportam de forma semelhante à evolução das espécies. A estas estruturas são aplicados os chamados operadores genéticos, como recombinação e mutação, entre outros. Cada indivíduo recebe uma avaliação que é uma quantificação da sua qualidade como solução do problema em questão. Com base nesta avaliação serão aplicados os operadores genéticos de forma a simular a sobrevivência do mais apto.

O esquema geral do algoritmo pode ser representado pelo pseudocódigo mostrado na Figura 9. Os diferentes ramos dos algoritmos evolucionários apresentam variações principalmente na representação de indivíduos e nos operadores genéticos. Tais algoritmos são considerados técnicas de *busca aleatória guiada*, uma vez que usam informações do estado corrente para guiar a pesquisa. A Figura 10 mostra um diagrama do posicionamento dos algoritmos evolucionários como técnica de busca.

```
INICIO
  T = 0
  INICIALIZA POPULAÇÃO P(0);
  AVALIA POPULAÇÃO P(0);
  ENQUANTO O CRITÉRIO DE PARADA NÃO FOR SATISFEITO
    P′ = SELECIONA PAIS P(T);
    P′ = RECOMBINAÇÃO E MUTAÇÃO (P′);
    AVALIA POPULAÇÃO P′;
    P(T + 1) = SELECIONA SOBREVIVENTES (P′)
    T = T + 1;
  FIM ENQUANTO
FIM
```
Figura 9 Pseudocódigo de um algoritmo evolucionário padrão

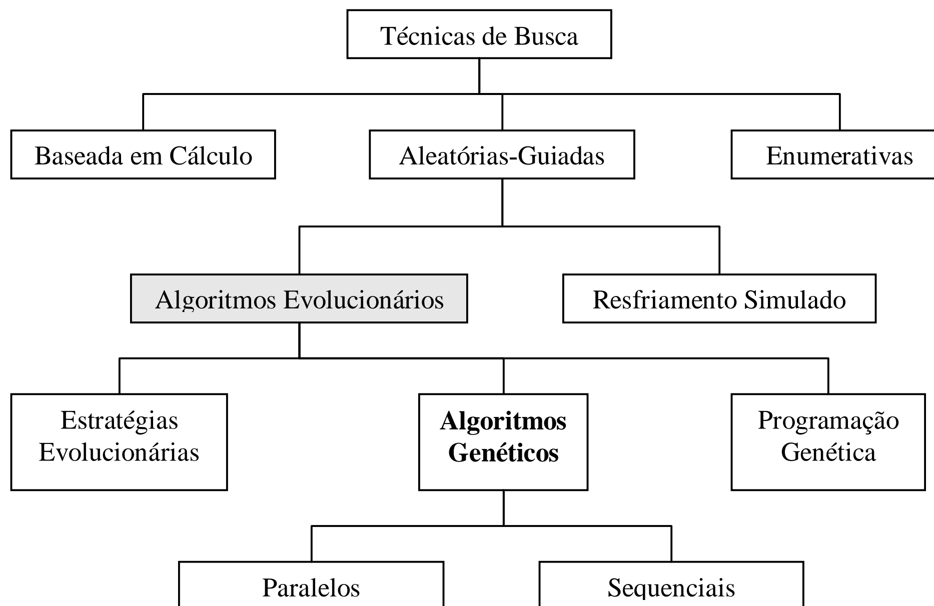

Figura 10 Posicionamento dos AEs como técnica de busca.
Fonte: Adaptado de Linden (2006)

O AG clássico tem como principal característica a predominância do operador de recombinação, geneticamente inspirado, na geração de novos indivíduos. Nas seções seguintes serão apresentados os componentes básicos do algoritmo.

2.2.1 Representação dos indivíduos

A representação dos indivíduos, ou representação cromossomial, corresponde à maneira de representar uma solução do problema de forma a ser tratada pelo AG. Em outros termos, significa a definição do genótipo e fenótipo do indivíduo. Conforme destacam Eiben e Smith (2003), é importante escolher a representação correta do problema a ser tratado para se conseguir bons resultados. Normalmente isto requer um bom conhecimento do domínio da

aplicação. Bonissone et al. (2006) consideram esta a principal etapa para se embutir o conhecimento do domínio do problema no algoritmo.

Dentre as representações mais comumente utilizadas, pode-se citar:

- Representação binária: o genótipo consiste numa sequência de dígitos binários. Cada gene é representado por um *bit*, o que torna simples a aplicação dos operadores genéticos. Esta representação, entretanto, exige a conversão dos dados do problema para a forma binária o que pode acarretar numa sequência muito grande de *bits*, prejudicando a eficiência do algoritmo.
- Representação numérica: em muitos problemas, utilizar cromossomos que representam diretamente os parâmetros sendo otimizados, como números reais, pode proporcionar maior eficiência para o AG (LINDEN, 2006). Dessa forma, o genótipo com k genes é representado por um vetor (x_1,..., x_k) onde x_k representa um número real, normalmente com valores limitados.

Linden (2006) ressalta que a representação cromossomial é completamente arbitrária, devendo ser considerado as dificuldades de cada representação.

2.2.2 Função de avaliação

A função de avaliação tem como objetivo fornecer uma medida da qualidade da solução de um indivíduo. Esta medida serve de base para a seleção dos indivíduos e dessa forma possibilita que o AG guie a busca da solução. Esta função também é comumente chamada de função *fitness* (aptidão).

2.2.3 Seleção de pais

Segundo Linden (2006), o processo de seleção deve simular o mecanismo de seleção natural de forma que indivíduos mais aptos (com maior *fitness*) tenham maior chance de reproduzir. É importante, entretanto, que os indivíduos menos aptos também possam gerar descendentes de forma a evitar a convergência genética (quando a população é composta de indivíduos com pouca ou nenhuma variação genética).

A seguir são apresentados três métodos de seleção comumente utilizados:

- Roleta viciada: neste método cada cromossomo possui a probabilidade de escolha proporcional à sua avaliação (a seleção ocorre de maneira similar à utilização de uma roleta, na qual cada porção representa o *fitness* de um indivíduo). Este método apresenta alguns problemas conforme comentam Eiben e Smith (2003): indivíduos que apresentam uma avaliação muito superior que o resto, tendem a dominar toda a população causando a convergência prematura do algoritmo; quando os valores de fitness estão muito próximos, não há pressão seletiva, consequentemente a seleção se torna quase uniformemente aleatória.
- Seleção por *ranking*: a seleção é realizada com base no *ranking* dos indivíduos. Este método evita o problema da convergência prematura, mas requer a ordenação dos indivíduos no cálculo da nova avaliação.
- Seleção por torneio: consiste em selecionar aleatoriamente *k* indivíduos e então escolher o de melhor *fitness*. Este método tem a vantagem de não necessitar do conhecimento global da população (EIBEN; SMITH, 2003). Um problema deste método é que o pior indivíduo apenas será escolhido se ele for sorteado todas *k* vezes (LINDEN, 2006).

Várias outras técnicas podem ser encontradas na literatura e a escolha deve ser realizada de forma a equilibrar a capacidade de exploração e convergência do AG.

2.2.4 Operador de recombinação

Recombinação é o processo pelo qual um novo indivíduo (solução) é criado a partir da informação contida dentro de duas ou mais soluções pais (EIBEN; SMITH, 2003, p. 46). Este processo, considerado uma das principais características dos AGs, também é chamado de *crossover* como uma analogia do processo biológico que ocorre na meiose.

O funcionamento deste operador depende da representação cromossomial adotada. Para a representação binária, um operador comum é o *crossover* de dois pontos ilustrado na Figura 11. Este operador atua do seguinte modo:

- Escolhem-se aleatoriamente dois pontos de corte utilizado para separar o cromossomo dos pais em três partes;
- Geram-se dois filhos resultantes da combinação de duas partes de um pai e uma parte do segundo pai.

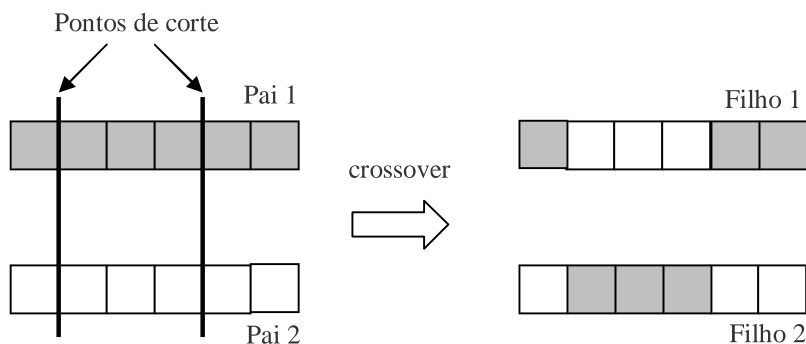

Figura 11 Ilustração do operador *crossover* de dois pontos

Um exemplo de operador de recombinação utilizado para representações numéricas é o *crossover flat* (Figura 12) que consiste em estabelecer um intervalo fechado para cada par de valores no cromossomo e então obter um valor aleatório pertencente a este intervalo.

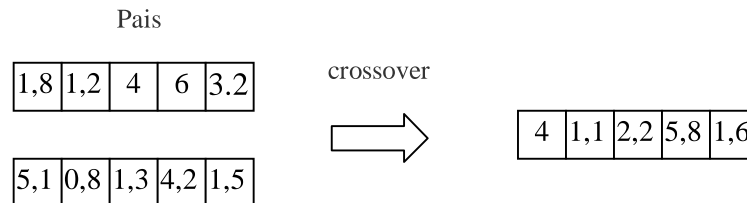

Figura 12 Exemplo de operação do *crossover flat*

2.2.5 Operador de mutação

O operador de mutação realiza uma alteração, com uma probabilidade pequena, em um ou mais genes do indivíduo. Segundo Linden (2006), ele é fundamental para a continuidade da existência da diversidade genética na população. A probabilidade de mutação é um parâmetro dos AGs e pode ser alterado durante a execução do algoritmo de forma a manter a capacidade exploratória da busca.

O operador comumente utilizado em representações binárias consiste na verificação de cada *bit* (gene) a probabilidade de realizar a sua inversão. Este operador é ilustrado na Figura 13. No caso de indivíduos com representações numéricas, o operador pode atuar através do sorteio de um novo valor para um determinado gene dentro de um intervalo pré-definido.

Figura 13 Ilustração do operador de mutação em uma representação binária.

2.2.6 Módulo de população

A forma como é definida a população das gerações é controlada pelo módulo de população (LINDEN, 2006). A estratégia mais simples é substituir todos os indivíduos pelos filhos gerados. Já no modelo conhecido como *steady-state,* apenas uma parcela da população é substituída. Neste caso, os filhos normalmente substituem os piores pais o que pode acelerar a convergência do algoritmo.

Uma pequena alteração na seleção de sobreviventes pode ser realizada a fim de manter os melhores indivíduos de cada geração. Esta estratégia, conhecida como elitismo, pode melhorar de forma significante o desempenho do AG.

2.2.7 Aplicações

Linden (2006) destaca algumas características que tornam o AG uma boa técnica de busca para problemas considerados intratáveis (devido ao tamanho do espaço de busca):

- Permite processamento paralelo.
- Não utilizam informação local, o que torna os AGs menos sensíveis aos máximos locais.

- Não são afetadas por descontinuidades na função a ser otimizada.
- São capazes de lidar com funções discretas e contínuas.

Heaton (2008), apresenta algumas utilizações comuns de AGs, resumidas no Quadro 2.

Quadro 2 Aplicações comuns para algoritmos genéticos

Aplicações de Algoritmos Genéticos	
Propósito	**Aplicação**
Otimização	Produção de horários, roteamento de transporte, determinação de layout de circuitos elétricos.
Design	Aprendizado de máquina: design de redes neurais, design e controle de robôs.
Aplicações de negócio	Utilizado em negociações financeiras, avaliação de crédito, distribuição de orçamentos, detecção de fraudes

Fonte: Adaptado de Heaton (2008).

2.3 Redes neurais artificiais evolutivas

Redes Neurais Artificiais Evolutivas constituem a classe de RNAs que utilizam estratégias evolutivas no processo de treinamento ou definição da arquitetura de rede. Esta combinação é motivada pelas características positivas apresentadas pelos algoritmos evolutivos para a resolução de problemas de busca.

YAO (1999) apresenta uma revisão das abordagens utilizadas na combinação de RNAs com AEs. Esta combinação pode ter como objetivo:

- Evolução dos pesos sinápticos.
- Otimização de arquitetura.
- Otimização de parâmetros.
- Otimização do conjunto de treinamento.

É importante observar que a estratégia evolutiva pode conter mais de um objetivo. Um exemplo é o trabalho de Leung et al. (2003) onde foi utilizado um AG modificado para a otimização da estrutura e dos parâmetros da rede neural.

Nas seções seguintes são apresentados alguns aspectos das abordagens utilizadas para a otimização da arquitetura e dos pesos de uma RNA.

2.3.1 Evolução dos pesos sinápticos

Métodos de treinamento baseado em gradientes como o *backpropagation* são sensíveis aos mínimos locais da função erro (YAO, 1999). A fim de superar este problema, foram propostos algoritmos evolutivos de aprendizagem supervisionada, uma vez que tais técnicas são alternativas interessantes para encontrar soluções próximas do mínimo global. Outra característica interessante desta abordagem é a possibilidade de se utilizar funções de ativação não diferenciáveis.

Segundo Yao (1999), esta abordagem consiste em duas principais etapas: a escolha da representação do indivíduo (conjunto de pesos sinápticos) e a definição dos operadores genéticos. O ciclo geral da evolução dos pesos é mostrado na Figura 14. O critério de parada pode ser, por exemplo, o erro mínimo desejado.

1. Decodificação de cada indivíduo da geração atual em um conjunto de pesos e a construção da correspondente RNA.
2. Avaliação de cada indivíduo com base no erro médio quadrático da RNA correspondente (outras funções de erro podem ser utilizadas).
3. Seleção dos pais para reprodução.
4. Aplicação dos operadores de recombinação e mutação para formar a nova geração.

Figura 14 Ciclo geral da evolução dos pesos de uma RNA
Fonte: Adaptado de Yao (1999).

A representação binária do indivíduo consiste na conversão de cada peso em um número binário de certo tamanho (Figura 15). As principais desvantagens desta representação é a limitação da precisão dos pesos e o tamanho excessivo do cromossomo para redes com muitas conexões. Esta representação foi utilizada por James e Frenzel (1993) para o treinamento de uma rede neural de alta ordem. Neste trabalho, ao se utilizar uma função de penalização no cálculo do *fitness*, os resultados obtidos foram superiores se comparados com o treinamento utilizando o BP.

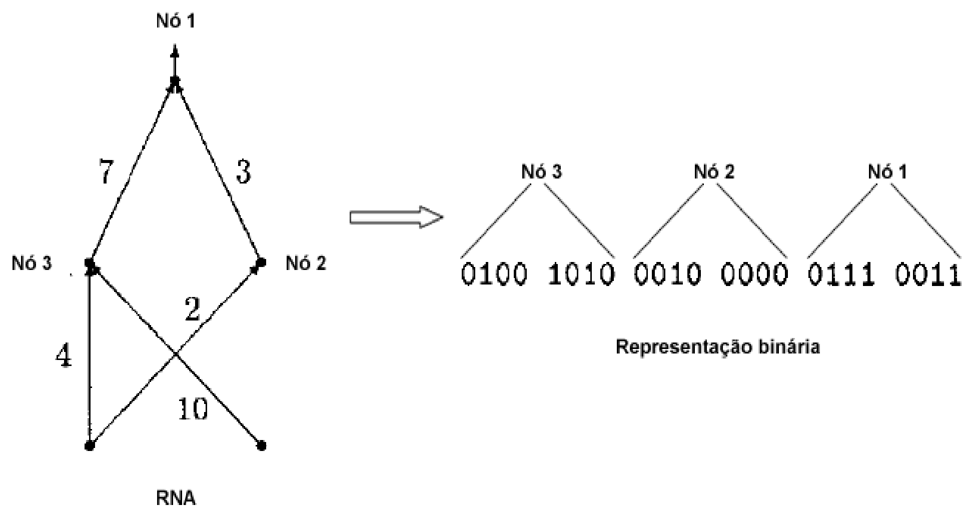

Figura 15 Exemplo de representação binária de uma RNA
Fonte: Adaptado de Yao (1999)

O cromossomo também pode ser representado diretamente pelo conjunto de pesos da RNA. Esta forma de representação apresenta maior escalabilidade se comparada com a representação binária, mas requer a utilização de operadores de combinação e mutação específicos. Montana e Davis (1989) desenvolveram e testaram vários operadores. Os que obtiveram melhores resultados foram os operadores de mutação e crossover baseados em nós (neurônios) representados por partes dos cromossomos. Em alguns trabalhos foi utilizada uma versão

modificada do AG para realizar o treinamento da RNA. Segundo Pan; Wu e Lai (2007) o ponto chave da AG modificado é a utilização de um crossover que distribui os cromossomos gerados de forma a situarem próximos do centro e do limite de uma região do espaço de busca.

A otimização dos pesos pode ser realizada de forma híbrida, combinando um AE, para realizar a busca global, com uma técnica de busca local. Li, Chen e Li (2009) utilizaram a abordagem híbrida de BP e AG para otimizar a medição de perturbações no sistema de energia elétrica por uma rede neural MLP. Irani e Nasimi (2011) também utilizaram uma combinação de AG e BP para melhorar a confiabilidade de uma RNA aplicada na previsão da permeabilidade de um reservatório de água. O esquema da implementação utilizada é mostrado na Figura 16.

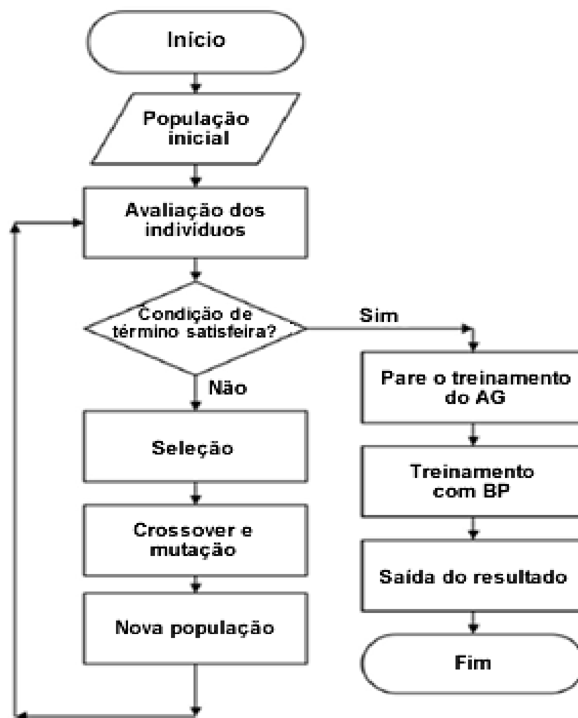

Figura 16 Framework combinando BP com AG
Fonte: Adaptado de Irani e Nasimi (2011)

2.3.2 Evolução da arquitetura de rede

A definição da arquitetura de uma RNA muitas vezes é realizada a partir de um processo de tentativa e erro ou com base em conhecimentos empíricos, consequentemente, a escolha de uma boa estrutura pode consumir bastante tempo. No projeto evolucionário de RNAs, cada indivíduo pode ser visto como um ponto ou estado no espaço de possíveis redes (BRAGA; CARVALHO; LUDERMIR, 2007, p. 126). Utilizando um AG é possível encontrar indivíduos que representem boas arquiteturas de redes neurais.

Segundo Yao (1999), um dos aspectos chaves na evolução das arquiteturas, é a forma de como são codificadas no cromossomo. Braga, Carvalho e Ludermir (2007) apresentam quatro fatores a serem considerados na representação:

- Se a representação permite que soluções ótimas ou aproximadamente ótimas sejam representadas.
- Como estruturas inválidas podem ser excluídas.
- Como devem atuar os operadores de reprodução, de forma que a nova geração possua somente redes válidas.
- Como a representação suporta o crescimento em tamanho das redes.

A representação pode ser direta, se todas as conexões da rede estiverem representadas no cromossomo, ou indireta, se apenas alguns parâmetros são codificados (por exemplo, o número de camadas e o número de neurônios em cada camada escondida). A Figura 17 mostra o ciclo geral da evolução de arquiteturas apresentado por Yao (1999).

1. Decodificação de cada indivíduo da geração atual em uma arquitetura.
2. Treinamento de cada RNA com a arquitetura decodificada por uma regra de aprendizado pré-definida.
3. Cálculo da avaliação de cada indivíduo com base nos resultados do treinamento e em outros critérios como a complexidade da arquitetura.
4. Seleção dos pais para reprodução.
5. Aplicação dos operadores de recombinação e mutação para formar a nova geração.

Figura 17 Ciclo geral da evolução de arquiteturas de RNAs.

Ao utilizar a representação indireta é possível utilizar parte do cromossomo para codificar os parâmetros de treinamento da rede neural. Um exemplo é o trabalho de Majdi e Beiki (2010) que utilizaram um cromossomo para evoluir, ao mesmo tempo, os parâmetros de aprendizado do BP e o número de neurônios na camada escondida.

2.3.3 Trabalhos relacionados

Na revisão apresentada por Yao (1999) pode-se observar que as estratégias evolutivas aplicadas nas RNAs contribuem para otimização do processo de treinamento mas, geralmente, são computacionalmente caras. Estas características têm motivado a adoção de implementações paralelas de RNAEs.

Riessen; Williams e Yao (1999) implementaram uma estrutura paralela para o algoritmo evolucionário EPNet. O EPNet é uma estratégia que utiliza a programação evolucionária (PE) para definir a arquitetura de uma MLP. A PE não utiliza o operador de *crossover* no processo de evolução. Cinco operadores de mutação foram utilizados no EPNet: treinamento híbrido utilizando um algoritmo composto por um BP modificado e pela técnica de busca resfriamento

simulado, exclusão de nós, exclusão de conexões, inclusão de nós e inclusão de conexões. A implementação utilizou o paralelismo da população e o paralelismo individual. Os resultados mostraram melhorias na precisão e performance do treinamento.

Wu et al (2010) otimizou uma RNA para o diagnóstico de estudantes com dificuldades de aprendizagem utilizando metodologias com algoritmos genéticos em plataforma distribuída. No trabalho um procedimento baseado em algoritmos genéticos foi utilizado para seleção das características a serem utilizas no treinamento. Posteriormente, foi realizada a busca pelos parâmetros (número de neurônios da camada escondida, taxa de aprendizado e taxa de momento) utilizando uma abordagem paralela de AG. Como resultado obteve-se uma maior taxa de identificação correta se comparada com trabalhos anteriores.

2.4 Sistemas distribuídos

Coulouris; Dollimore e Kindberg (2007, p. 16) definem um sistema distribuído como sendo "aquele no qual os componentes de hardware ou software, localizados em computadores interligados em rede, se comunicam e coordenam suas ações apenas enviando mensagens entre si". O desenvolvimento destes sistemas tem como principal motivação o compartilhamento de recursos. Seu projeto envolve alguns desafios relacionados com os aspectos de segurança, escalabilidade, tratamento de falhas, concorrência e transparência.

Nas seções seguintes, serão apresentados alguns conceitos básicos que envolvem o desenvolvimento deste trabalho.

2.4.1 Modelos de arquiteturas

Um modelo de arquitetura define a maneira como ocorre a interação entre os componentes dos sistemas distribuídos e como eles estão localizados (COULOURIS; DOLLIMORE; KINDBERG, 2007). Os dois principais tipos de modelos são o cliente-servidor e o *peer-to-peer*.

O modelo cliente-servidor é estruturado por processos clientes que solicitam e recebem informações de um processo servidor (Figura 18). O processo servidor é responsável por gerenciar os recursos solicitados pelos clientes. Este é o modelo predominante na Internet.

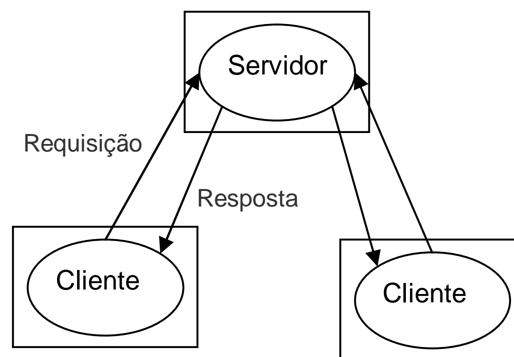

Figura 18 Modelo cliente-servidor

No modelo *peer-to-peer* cada processo (*peer*) desempenha funções semelhantes apresentando, ao mesmo tempo, características de um processo servidor e cliente (Figura 19). Esta arquitetura oferece maior capacidade de compartilhamento de recursos e escalabilidade.

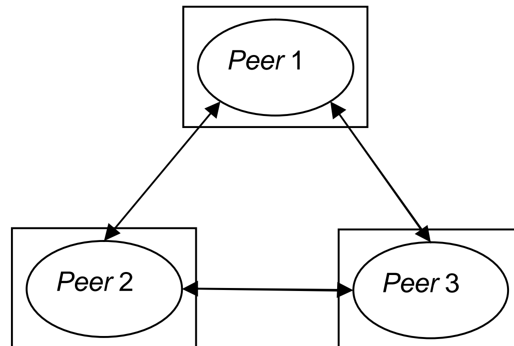

Figura 19 Modelo *peer-to-peer*

2.4.2 Comunicação entre processos na Internet

A comunicação entre processo pode ser representada pela passagem de mensagens através das operações *send* e *receive*, definidas em termos de destinos e mensagens (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 127). A comunicação pode ocorrer de forma síncrona, quando o remetente espera uma resposta ao realizar um envio (*send* bloqueante), ou assíncrona, quando o remetente pode prosseguir o envio de mensagens assim que ela tenha sido copiada para um *buffer* local (*send* não bloqueante).

Na Internet o destino de uma mensagem é identificado pelo par endereço IP e porta local. A porta representa o destino da mensagem dentro de um sistema final. Um processo pode utilizar várias portas a fim de estabelecer uma ou mais conexões com um ou mais processos.

Uma aplicação distribuída pode exigir confiabilidade e ordenamento. A confiabilidade representa a garantia de entrega e a integridade das mensagens. O ordenamento refere-se à garantia da ordem de entrega das mensagens.

A arquitetura de protocolos da Internet, TCP/IP, provê duas formas de comunicação entre aplicações: comunicação por datagrama UDP ou comunicação por fluxo TCP.

A comunicação por datagrama UDP (User Datagram Protocol) é mais rápida, entretanto não oferece nenhuma garantia de confiabilidade ou da ordem de entrega das mensagens. Neste tipo de comunicação não é estabelecida uma conexão entre os processos, assim, para enviar uma mensagem de resposta, é necessário recuperar em cada datagrama o endereço IP e a porta do remetente. Esta forma de comunicação é adequada para aplicações em tempo real que possuem certa tolerância a falhas na transmissão.

A comunicação por fluxo TCP (Transmission Control Protocol) envolve o estabelecimento de uma conexão entre os processos e oferece garantia de confiabilidade e da ordem de entrega das mensagens. Esta comunicação envolve um processo servidor que utiliza uma porta para a *escuta* de pedidos de conexão de processos clientes. Uma vez estabelecida a conexão, é criado um fluxo de entrada e saída de dados utilizado pela aplicação no envio e recebimento de mensagens.

2.5 Paralelização de algoritmos genéticos

Para a resolução de problemas não triviais, o ciclo reprodutivo de um AE com uma grande população e/ou com indivíduos complexos requer alto recurso computacional (ALBA; TOMASSINI, 2002). A fim de melhorar a eficiência destas técnicas de busca, vários modelos paralelos foram desenvolvidos. De modo geral, estes modelos envolvem a paralelização de operadores genéticos, da função de avaliação e o uso de populações estruturadas (o que corresponde à forma como os indivíduos estão distribuídos no espaço).

Alba e Tomassini (2002) resumem as vantagens esperadas a partir da paralelização de AEs:

- Possibilidade de encontrar várias soluções para um mesmo problema.
- Busca paralela a partir de múltiplos pontos no espaço.

- Busca mais eficiente, mesmo quando não é utilizada uma plataforma distribuída.
- Fácil cooperação com outros procedimentos de busca.
- Aumento da velocidade devido ao uso de múltiplos processadores.

Nas seções a seguir, serão apresentados os principais modelos de paralelização de AEs. Estes modelos podem ser aplicados em qualquer ramo dos AEs, inclusive para a implementação de AGs paralelos.

2.5.1 Panmictic

Este modelo tem como principal característica a manutenção de uma única população global. Dessa forma, a paralelização pode ser realizada no processo de avaliação de indivíduos e/ou na utilização dos operadores genéticos. Esta implementação acelera consideravelmente o algoritmo, mas exige uma forma de gerenciar os processos.

Segundo Linden (2006, p. 278), a abordagem mestre-escravo (Figura 20) é comumente utilizada neste modelo. Nesta abordagem, o processo mestre é responsável por controlar a seleção dos processadores para realizar a avaliação dos indivíduos, ou aplicar os operadores sobre os cromossomos. Dois problemas podem ser observados nessa abordagem:

- A dificuldade de se realizar o balanceamento do uso dos processadores.
- A velocidade máxima do AE é limitada pela capacidade do processador mestre de gerenciar os processadores escravos.

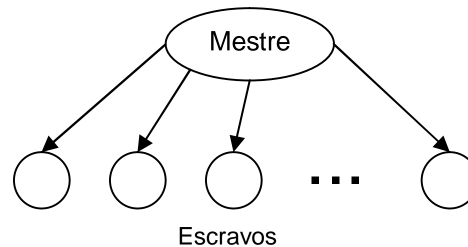

Figura 20 Ilustração da abordagem mestre-escravo

2.5.2 Modelo de ilhas

O modelo de ilhas, também chamado de modelo *coarse-grained* ou modelo distribuído, foi inspirado na teoria do equilíbrio pontual de Eldredge e Gould. Esta teoria considera que o efeito da pressão seletiva em pequenas populações é mais intenso do que em grandes populações, consequentemente, surgem espécies melhores adaptadas às peculiaridades do seu ambiente.

No modelo de ilhas, a população de cromossomos é particionada em certo número de subpopulações isoladas que evoluem de forma independente e, periodicamente, trocam indivíduos através de migrações (LINDEN, 2006). Este modelo exige a criação de uma política de migração que defina:

- a topologia de ilhas;
- o período entre migrações;
- quantos indivíduos serão trocados;
- quais indivíduos serão trocados;
- como ocorrerá a sincronização entre os processos.

A topologia de ilhas determina o caminho pelo qual ocorrerão as migrações. O modelo de transporte mais simples é o modelo em anel ilustrado na Figura 21.

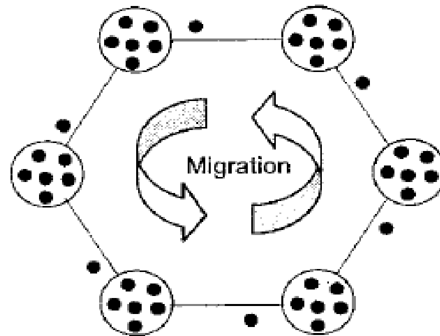

Figura 21 Topologia de migração em anel
Fonte: Alba e Tomassini (2002).

As ilhas podem evoluir sua população de forma distinta, ou seja, cada ilha pode utilizar diferentes parâmetros do algoritmo ou, até mesmo, diferentes versões de AEs. O critério de parada deve considerar a população como um todo, pois mesmo se uma ilha alcançar a convergência genética, a migração poderá introduzir novos materiais genéticos na população.

Segundo Linden (2006), este tipo de implementação é utilizado em sistemas distribuídos que possuem memória local privada, uma vez que cada processador trabalha de forma independente, sincronizando durante o processo de migração. Segundo Alba e Tomassini (2002), o modelo distribuído é normalmente mais rápido do que o modelo panmictic devido ao número reduzido de avaliações realizadas de forma separada no espaço de busca.

2.5.3 Modelo de vizinhança

No modelo de vizinhança, também chamado de modelo *finely-grained*, cada processador aloca apenas um indivíduo em uma estrutura de vizinhança representada por uma grade ou uma hipergrade (Figura 22). Desta forma, cada indivíduo interage apenas com os indivíduos da sua vizinhança.

Figura 22 Modelo de vizinhança utilizando uma estrutura em grade
Fonte: Alba e Tomassini (2002).

A principal questão a se resolver ao utilizar este modelo é como representar a estrutura de vizinhança. Linden (2006) comenta que o tamanho da vizinhança não deve ser nem pequeno demais, prejudicando a diversidade genética, nem demasiado grande, degradando o desempenho do algoritmo.

A implementação deste modelo requer uma intensa comunicação entre os processadores, sendo adequada para estruturas maciçamente paralelas, conforme destaca Linden (2006).

2.5.4 Modelos híbridos

Conforme apresentam Alba e Tomassini (2002), vários modelos híbridos combinando dois ou mais níveis de abordagens paralelas foram empreendidas. Um exemplo é a combinação do modelo de ilhas com a paralelização global (implementação mestre-escravo do modelo panmictic), onde cada ilha paraleliza a avaliação de seus indivíduos. Alguns modelos híbridos, utilizando o modelo de ilhas no primeiro nível, são ilustrados na Figura 23.

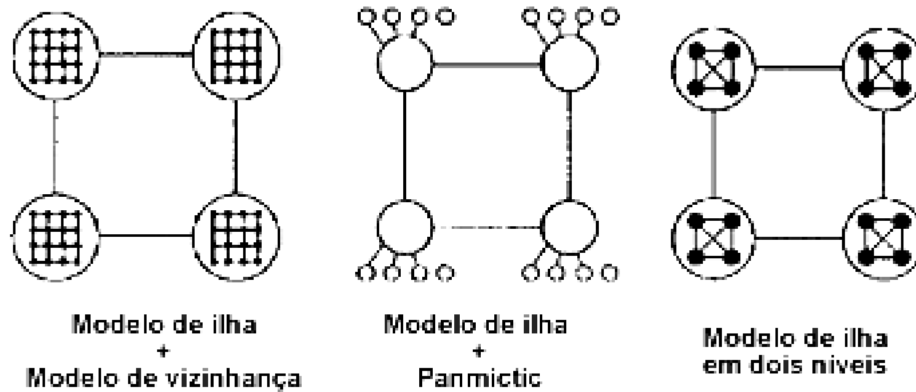

Figura 23 Exemplo de modelos híbridos
Fonte: Adaptado de Alba e Tomassini (2002).

2.6 Java

A linguagem de programação Java surgiu a partir de um projeto de pesquisa financiado pela Sun Microsystems em 1991. O projeto, liderado por James Gosling, tinha como motivação a união entre dispositivos digitais e computadores. Com mercado desfavorável para os dispositivos eletrônicos inteligentes, a linguagem passou a utilizar a infraestrutura da Internet e, em 1995, foi formalmente anunciada (ORACLE, 2011b).

Um programa Java, no geral, passa por cinco fases até a sua execução: edição, compilação, carga, verificação e execução (DEITEL; DEITEL, 2005, p. 8). O compilador Java (javac) converte o código fonte Java em *bytecodes* que são carregados para a memória principal e passam por um processo de validação a fim de evitar violações de segurança. Os *bytecodes* são executados pela Java Virtual Machine (JVM). A utilização da máquina virtual torna a linguagem independente de plataforma uma vez que os *bytecodes* podem ser executados em qualquer hardware que possua a JVM instalada. O processo de execução de um programa Java é ilustrado na Figura 24.

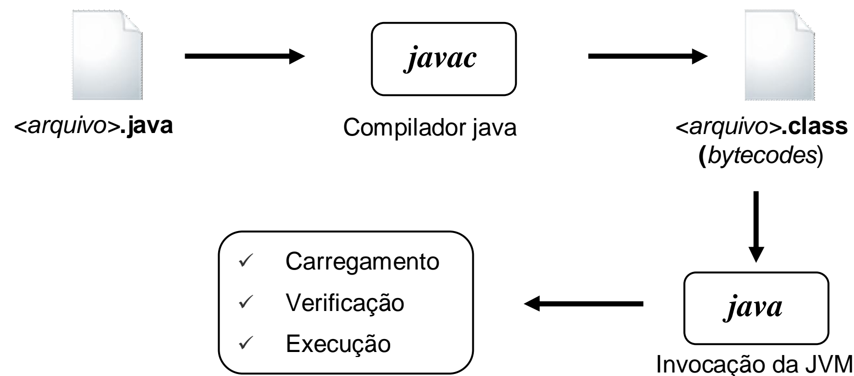

Figura 24 Esquema simplificado da execução de um programa Java

Java é uma linguagem orientada a objeto (LOO). Conforme descreve Linden (2006, p.13), o conceito básico das LOO é um objeto, o qual constitui um pacote de software que contém uma coleção de procedimentos e dados relacionados, representando um item, unidade ou entidade. No Java, a unidade de programação é a classe, a partir do qual os objetos são eventualmente instanciados (DEITEL; DEITEL, 2005, p. 15). Os conceitos básicos da programação orientada a objeto com Java envolvem (ORACLE, 2011a):

- Objeto: pacote que software que relaciona comportamentos e estados. A organização do software por meio de objetos trás alguns benefícios como modularidade, encapsulamento, abstração e facilidade de reaproveitamento de códigos.
- Classe: modelo ou protótipo a partir do qual os objetos são criados.
- Herança: mecanismo que permite as classes *herdarem* comportamentos e estados de outras classes. Este mecanismo possibilita estruturar as classes do um software de maneira hierárquica.
- Interface: representa um contrato entre uma classe o mundo exterior. A interface força uma classe a apresentar determinados comportamentos.

- Pacote: representa um conjunto lógico de nomes para a organização de classes e interfaces.

2.7 Sockets

Socket é uma interface de comunicação entre processos que abstrai a utilização dos protocolos TCP e UDP. Os sockets são originários do UNIX BSD, mas também estão presentes na maioria das versões do UNIX, incluindo LINUX, assim como no Windows e Macintosh OS (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 128).

A interface socket envolve a utilização de um processo servidor na qual uma porta é utilizada para a escuta de conexões (com a utilização do TCP) ou para recebimento de datagramas (com a utilização do protocolo UDP) e um processo cliente que inicia uma conexão ou envia um datagrama para um par endereço IP e porta específico. A Figura 25 ilustra a comunicação entre processos utilizando socket TCP. A classe *java.net.Socket* permite que programas Java se comuniquem através da rede de uma forma independente de plataforma (ORACLE, 2011c).

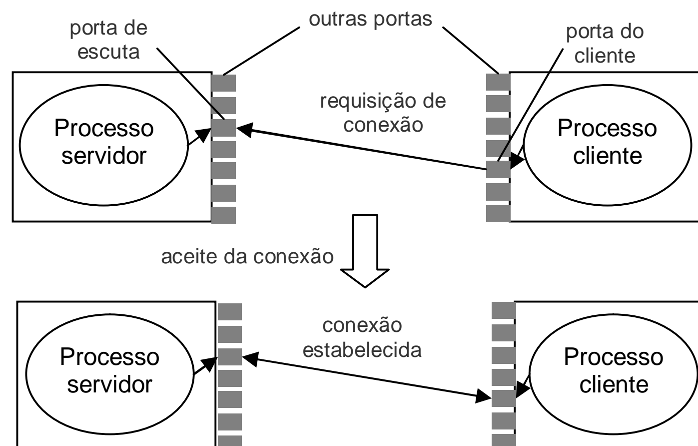

Figura 25 Ilustração de uma conexão por meio de sockets TCP

3 MATERIAL E MÉTODOS

A aplicação foi desenvolvida utilizando a linguagem de programação Java. A comunicação entre os processos que compõe o sistema distribuído foi realizada por meio de sockets. Nas seções seguintes é apresentada a estratégia de implementação e a metodologia de desenvolvimento adotada.

3.1 Modelo de paralelização

O modelo utilizado na paralelização do algoritmo genético foi o modelo de ilhas devido principalmente as seguintes características:

- É o modelo que apresenta menor grau de interação entre as subpopulações, consequentemente, o número de mensagens transmitidas entre os processadores é menor. Dependendo da infraestrutura de rede o número de mensagens transmitidas limita significativamente o desempenho do sistema distribuído, assim é essencial considerar o grau de interação necessário entre os processos.
- Cada processador pode executar de forma independente, deste modo, não é necessário um mecanismo de balanceamento de carga. A execução pode ocorrer de forma assíncrona e é possível utilizar diferentes configurações do algoritmo em cada nó da rede.

O sistema distribuído utilizou o modelo de arquitetura *peer-to-peer* onde cada nó pode iniciar um processo de treinamento e manter conexões com outros nós de forma arbitrária. Assim, qualquer topologia de migração poderá ser adotada dependendo apenas da quantidade de nós disponíveis na rede. A Figura 26 ilustra o modelo adotado na implementação.

O melhor indivíduo de uma ilha (nó da rede) compõe o grupo de migração. Se este indivíduo apresentar a solução desejada ele é enviado para os

nós adjacentes da rede até alcançar o nó que iniciou o treinamento, finalizando a execução do algoritmo.

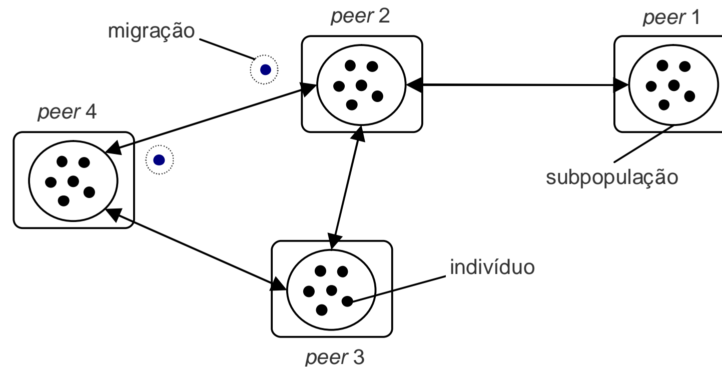

Figura 26 Ilustração do modelo adotado na implementação do sistema distribuído

3.2 Estrutura de classes

Na modelagem do sistema distribuído foram utilizados os conceitos de herança e polimorfismo do paradigma de programação orientado a objetos a fim de facilitar a utilização de diferentes algoritmos de aprendizagem no treinamento de uma rede MLP. Segundo DEITEL e DEITEL (2005), herança corresponde à forma de reutilização de software na qual uma nova classe é criada absorvendo os membros de uma classe existente e modificando ou adicionado novas capacidades. A implementação de uma hierarquia de heranças permite que diferentes objetos compartilhem uma mesma *superclasse* e possam assim ser tratados de forma homogênea. Esta característica representa o conceito de polimorfismo.

Nas seções a seguir são apresentados os diagramas de classes dos principais componentes do sistema desenvolvido.

3.2.1 Implementação da MLP

Para a implementação da RNA foi criada a classe *MLP* que contém uma ou mais camadas de neurônios representadas pela classe *CamadaMLP*. Cada *CamadaMLP* possui um número definido de neurônios, um vetor de entrada, uma matriz de pesos e um vetor de saída. A ligação entre as camadas é realizada utilizando a referência do vetor de saída de uma *CamadaMLP* como sendo o vetor de entrada de outra *CamadaMLP*.

No processamento de um padrão de entrada é utilizado em cada *CamadaMLP* um objeto que implementa a interface *FuncaoAtivacao*. Esta interface define o comportamento esperado de uma função de ativação utilizada no processamento de sinais em um neurônio facilitando a utilização de diferentes funções na configuração de uma rede neural. A Figura 27 mostra o diagrama de classe deste componente com alguns métodos das classes utilizadas, a implementação pode ser consultada no Apêndice A.

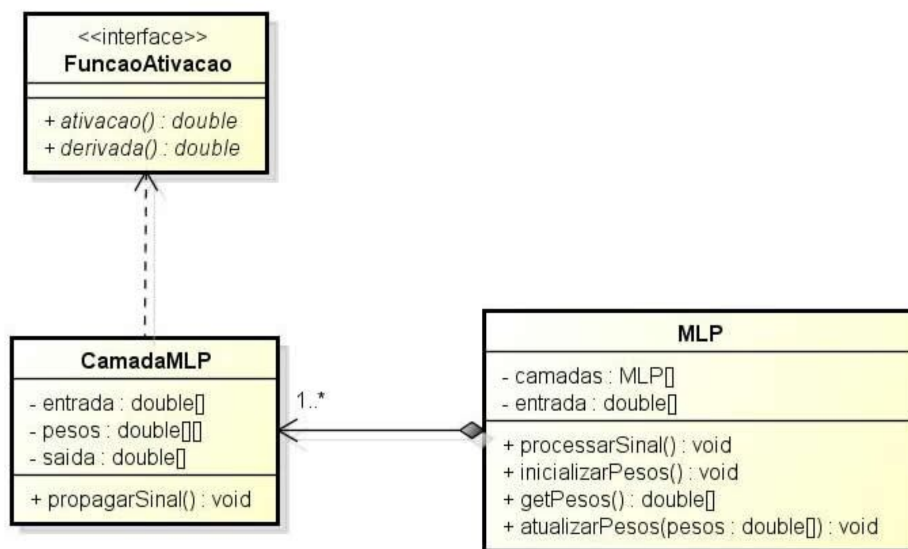

Figura 27 Diagrama de classe da implementação de uma MLP

3.2.2 Módulo de treinamento

Os algoritmos de aprendizagem utilizados no treinamento da MLP são um tipo de aprendizagem supervisionada, assim, foi criada a classe abstrata *AprendizagemSupervisionada* que define o comportamento padrão deste processo de aprendizagem. Esta classe utiliza um objeto *MLP* e uma implementação da interface *CondicaoParada*. Também foi implementado um *ouvinte* das iterações do algoritmo, utilizado para informar o usuário a respeito do progresso do treinamento.

Para que o algoritmo de aprendizagem possa ser utilizado no ambiente distribuído, ele deve implementar a interface *Ilha* que representa uma população de indivíduos de um algoritmo evolucionário. A Figura 28 mostra o diagrama de classe deste módulo com os principais métodos da classe *AprendizagemSupervisionada.*

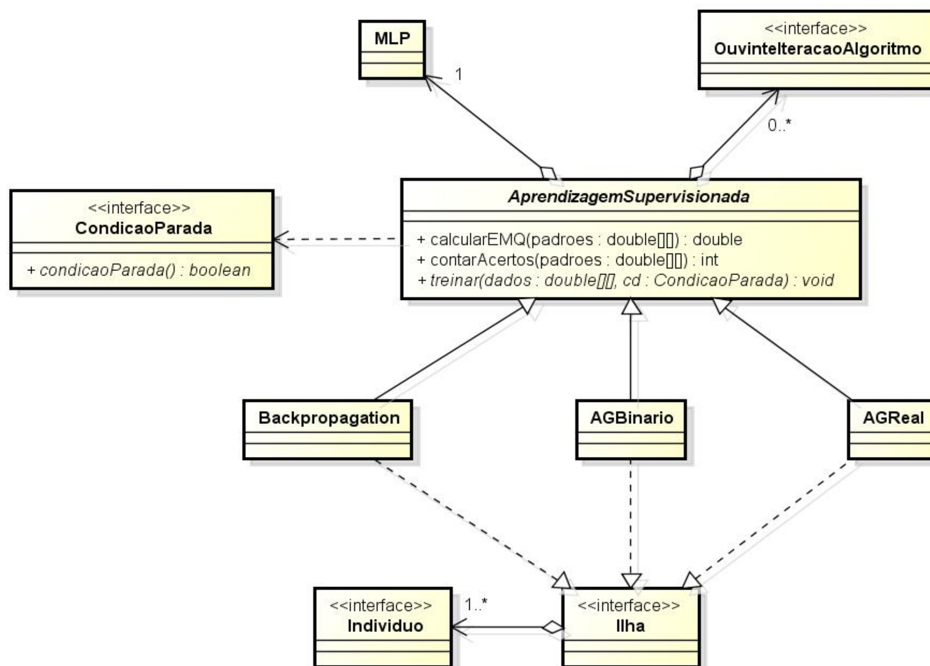

Figura 28 Diagrama de classe da implementação dos algoritmos de aprendizagem

3.2.3 Implementação do modelo de ilhas

A comunicação entre os nós da rede é gerenciada pela classe *Aplicação* (Apêndice D). Esta classe possui uma instância da classe *Servidor*, uma *thread* que utiliza um *ServerSocket* para receber conexões de outros nós. Para cada conexão iniciada, é criada uma classe *Conexao* que também estende a classe *Thread* e mantém um socket TCP aguardando o recebimento de mensagens.

Para implementar a paralelização do algoritmo genético foi criada a classe *ModeloIlhas* que gerencia um conjunto próprio de conexões uma vez nem todas as conexões estabelecidas pelo sistema distribuído precisam compor uma topologia de migração. O método migração é responsável por enviar o melhor indivíduo de uma ilha para as ilhas adjacentes além de incluir os imigrantes na população da ilha. Para isto o método recebe como parâmetro um objeto que implementa a interface *Ilha* mostrada na seção anterior. Quando um algoritmo (que implementa a interface *Ilha*) finaliza a sua execução, ele chama o método *fimExecusao* a fim de sinalizar o término do processo de aprendizagem. A classe *OuvinteModeloIlhas* é utilizada para notificar o término do treinamento considerando todos os nós da rede. A Figura 29 mostra o diagrama de classe simplificado da implementação deste componente.

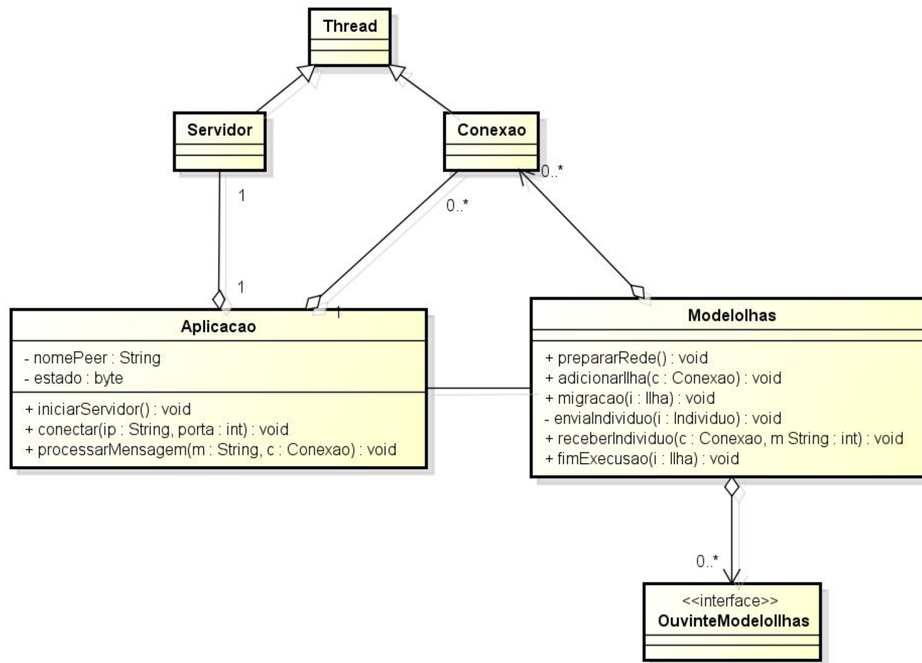

Figura 29 Diagrama de classe da implementação do modelo de ilhas

3.3 Protocolo de comunicação

Conforme descreve Coulouris, Dollimore e Kindberg (2007, p. 79): “o termo protocolo é utilizado para designar um conjunto bem conhecido de regras e formatos a serem utilizados na comunicação entre processos a fim de realizar uma determinada tarefa”. Para a implementação do modelo de ilhas foi necessário o desenvolvimento de um protocolo a fim de coordenar o envio de mensagens de controle e de indivíduos na rede. A estrutura das mensagens utilizadas segue o padrão demonstrado na Figura 30. Nesta estrutura, a primeira linha é utilizada para definir o tipo de mensagem e qual ação deve ser realizada. A partir da segunda linha, dependendo do comando enviado, será preenchida com valores de variáveis ou objetos.

```
<comando>\n
<conteúdo>
```

Figura 30 Estrutura das mensagens utilizadas no sistema distribuído

As conexões entre os *peers* (nós da rede) são estabelecidas utilizando o comando *RNAED* seguido da versão do sistema utilizado. Cada *peer* possui um estado que pode ser: *disponível*, *aguardando* ou *ocupado*. Quando um *peer* inicia a configuração de uma MLP ele passa para o estado *aguardando*. No momento que o treinamento é iniciado o estado passa a ser *ocupado*.

A configuração de um treinamento envolve os seguintes passos:

1. Definição da estrutura da MLP;
2. Importação do conjunto de treinamento;
3. Configuração algoritmo de aprendizagem;
4. Configuração do período de migração;
5. Seleção das conexões que irão compor a topologia de migração;
6. Início do treinamento.

No passo 5 ao se adicionar uma ilha na topologia de migração, um objeto contendo todos os dados da RNA e do treinamento é enviado ao novo nó da rede. Nesta etapa, se o nó adicionado estiver com o estado *disponível*, a nova ilha é marcada como sendo uma ilha *subordinada* àquela que iniciou a conexão. Esta *flag* é importante para criar um caminho nos quais mensagens de controle são enviadas na rede indicando o fim do treinamento de uma ilha ou de toda a rede. A Figura 31 ilustra este processo de formação das conexões onde cada círculo representa uma ilha, o circulo preenchido indica o nó que iniciou a configuração do treinamento e as setas representam as conexões estabelecidas. As setas de cor cinza indicam que no momento em que a conexão foi estabelecida a ilha (nó) já estava com o treinamento configurado, resultado de

uma conexão anterior e dessa forma não é marcada como sendo uma ilha subordinada.

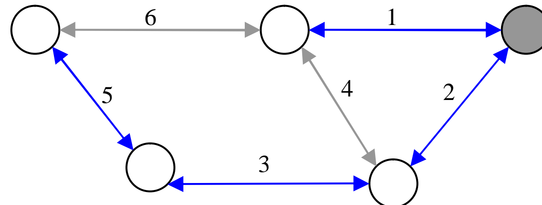

Figura 31 Ilustração do processo de formação de conexões

Durante a execução do treinamento, o procedimento de migração verifica a geração atual e, com base no período de migração configurado, envia o melhor indivíduo da população local aos nós adjacentes. Ao receber um indivíduo, o nó o armazena em um *buffer* até que o procedimento de *imigração* seja executado. Neste procedimento os indivíduos recebidos são inseridos na população, preservando o melhor indivíduo corrente. Esta política de migração evita que uma ilha fique bloqueada a espera de novos indivíduos.

No final do treinamento de um nó, ele verifica o estado das ilhas subordinadas e, se todas finalizaram a sua execução, envia uma mensagem notificando a ilha mestre (nó que iniciou a conexão). Este procedimento é repetido até que todas as ilhas conectadas ao nó que iniciou o treinamento o notifiquem do término da execução. Neste momento o nó retorna uma mensagem indicando o fim do treinamento da rede e envia a melhor solução encontrada para as ilhas subordinadas.

3.4 Algoritmos de aprendizagem

Foram implementados três algoritmos de aprendizagem: um AG utilizando representação binária (AGB), um AG utilizando representação real (AGR) e o Backpropagation com *momento*. O BP foi utilizado para comparar os

resultados obtidos com as versões sequencias dos AGs. A seguir é apresentada uma descrição geral dos algoritmos desenvolvidos.

3.4.1 AG com representação binária

Os pesos de uma rede neural podem ser convertidos para a representação binária utilizando um parâmetro de precisão e um intervalo definido de valores. Quanto maior o intervalo e a precisão, mais bits são necessários para representar um peso de uma RNA e, portanto, maior será o genótipo de um indivíduo. O algoritmo segue os seguintes passos:

1. Inicialização da população: após definido o tamanho do cromossomo (que depende dos parâmetros de codificação utilizados), cada indivíduo é gerado aleatoriamente;
2. Avaliação da população: cada indivíduo é decodificado em um vetor de valores reais que é então utilizado como os pesos da RNA aplicada no conjunto de treinamento. Neste processo o melhor indivíduo é armazenado em uma variável global e o seu erro quadrático médio (EQM) é registrado. Neste passo também é realizada uma verificação do melhor EQM corrente: Se após dez gerações não houver alterações no valor do erro, um novo indivíduo é inserido na população.
3. Verificação da condição de parada: se a condição de parada for verdadeira o treinamento é finalizado, caso o contrário o procedimento de geração ou migração é executado.
4. Geração: São aplicados os operadores de recombinação e mutação nos indivíduos selecionados por um determinado método de seleção. Os operadores são utilizados de acordo com uma probabilidade definida pelo usuário do sistema. Se o procedimento de migração for

executado, para cada indivíduo imigrante, um cromossomo filho é gerado utilizando o operador de crossover que combina o cromossomo do imigrante com um indivíduo selecionado da população corrente.

5. Avaliação de indivíduos: mesmo procedimento do passo 2.
6. Seleção de sobreviventes: todos os indivíduos da população corrente são substituídos pelos filhos, exceto o melhor indivíduo (elitismo).
7. Iteração: registra o resultado da geração e volta ao passo 3.

Os operadores de recombinação implementados foram:

- Crossover de um ponto;
- Crossover de dois pontos;
- Crossover de três pontos;
- Crossover de um ponto por peso.

O último operador funciona aplicando o *crossover* de um ponto em cada peso codificado no cromossomo. Os anteriores seguem a lógica demonstrada na seção 2.2.4. Os operadores de mutação implementados foram:

- Mutação por peso: para cada peso da RNA um bit da sua codificação é alterado de acordo com uma probabilidade definida pelo usuário;
- Mutação dois pontos: uma porção do cromossomo é selecionada de forma similar a um crossover de dois pontos e seus bits são alterados com uma probabilidade definida pelo usuário.

3.4.2 AG com representação real

Neste algoritmo o vetor de pesos de uma rede neural é utilizado como genótipo de um indivíduo, desta forma não é necessário o procedimento de decodificação do genótipo no passo de avaliação de indivíduos. A classe AGR (Apêndice C) utiliza operadores genéticos específicos para a codificação real

para implementar este processo de aprendizagem. O algoritmo segue os seguintes passos:

1. Inicialização da população: cada peso é gerado aleatoriamente dentro de um intervalo definido pelo usuário.
2. Avaliação da população: o cromossomo de cada indivíduo é utilizado como sendo o vetor de pesos da RNA aplicada no conjunto de treinamento. O melhor indivíduo é armazenado em uma variável global e o seu EQM é registrado. Da mesma forma que no algoritmo anterior, é realizada uma verificação do melhor EQM corrente: Se após cinco gerações não houver alterações no valor do erro, um novo indivíduo obtido a partir de alterações em um cromossomo da população.
3. Verificação da condição de parada: se a condição de parada for verdadeira o treinamento é finalizado, caso o contrário o procedimento de geração ou migração é executado.
4. Geração: São aplicados os operadores de recombinação e mutação nos indivíduos selecionados por um determinado método de seleção. Os operadores são utilizados de acordo com uma probabilidade definida pelo usuário do sistema. Se o procedimento de migração for executado, para cada indivíduo imigrante, um cromossomo filho é gerado utilizando o operador de crossover que combina o cromossomo do imigrante com um indivíduo selecionado da população corrente.
5. Avaliação de indivíduos: mesmo procedimento do passo 2.
6. Seleção de sobreviventes: todos os indivíduos da população corrente são substituídos pelos filhos, exceto o melhor indivíduo (elitismo).
7. Iteração: registra o resultado da geração e volta para o passo 3.

Os operadores de recombinação implementados foram:

- Crossover flat: apresentado na seção 2.2.4;
- Crossover dois pontos flat: opera de forma similar a um crossover de dois pontos sendo que numa parte do cromossomo, pertencente a um dos pais, é realizada a combinação dos pesos a partir de uma média ponderada;
- Crossover estruturado: uma estrutura é criada para mapear as regiões do cromossomo conforme o grau de ativação das conexões da RNA. Nesta estrutura os pesos são ordenados de acordo com a multiplicação de seu valor com o somatório dos valores de entradas da conexão correspondente. Os neurônios, representados no cromossomo por grupos de pesos, são ordenados conforme o somatório dos produtos dos pesos aplicados sobre sua saída. Utilizando esta estrutura, os pesos de uma porção entre 5% a 25% dos neurônios são alterados com uma probabilidade de 92% a 67%. As porcentagens são definidas conforme a média do número de gerações necessárias para obter um melhor indivíduo durante a execução do algoritmo. A atualização dos pesos é realizada a partir da combinação dos genes de dois indivíduos segundo a fórmula:

  $$w_j^{(f)} = w_j^{(p1)}(\tau + 0{,}33) + 0{,}65 w_j^{(p2)} \quad (13)$$

  onde $w_j^{(f)}$ corresponde ao peso j do cromossomo filho, $w_j^{(p1)}$ e $w_j^{(p2)}$ aos pesos equivalentes dos pais e τ um valor obtido aleatoriamente dentro do intervalo de 0 a 0,05.

O crossover estruturado evita o problema de permutação e permite que diferentes redes sejam combinadas de forma mais eficiente. Segundo Hancock (1992), o problema de permutação deriva da possibilidade de que as unidades equivalentes em dois cromossomos pais são definidas em diferentes locais.

Desta forma, genótipos distintos podem codificar redes semelhantes quanto aos resultados obtidos e, portanto, representam o mesmo ponto no espaço de busca.

Os operadores de mutação implementados foram:

- Mutação por neurônio: com a probabilidade de $1/n$, onde n corresponde ao total de neurônios da rede, os pesos de um neurônio são multiplicados por uma taxa entre 0,8 a 1,2.
- Mutação dois pontos: dois pontos do cromossomo são sorteados e os pesos entre eles são multiplicados por uma taxa entre 0,8 a 1,2 ou entre -0,8 a -1,2 com uma probabilidade de 60%.

3.4.3 Backpropagation

O BP implementado (Apêndice B) inclui, a cada de dez épocas, o procedimento de migração descrito na seção 3.3. O algoritmo implementa a interface *Ilha* através de uma estrutura com um único indivíduo cujo genótipo é o vetor de pesos da MLP. No procedimento de inicialização de pesos, os valores são obtidos dentro do intervalo de -1 a 1.

3.5 Procedimentos metodológicos

O desenvolvimento do projeto ocorreu em duas fases:

1. Avaliação do desempenho dos algoritmos genéticos a partir de comparações com resultados obtidos em treinamentos utilizando o BP.
2. Avaliação do sistema distribuído a partir da análise dos resultados obtidos nos treinamentos com diferentes números de ilhas na rede.

Para a realização dos testes, foi utilizado o repositório de aprendizagem de máquina UCI, mantido pela Universidade da Califórnia em Irvine. O repositório apresenta uma coleção de 211 bases de dados utilizadas pela

comunidade da área de aprendizagem de máquina para a análise empírica de algoritmos (FRANK; ASUNCION, 2011). Os problemas utilizados foram:

1. Íris: consiste num problema de classificação que envolve três espécies de flores. Utiliza quatro atributos e conta com 150 exemplos.
2. Câncer de mama: diagnóstico de câncer de mama. Tenta classificar se um tumor é benigno ou maligno baseando em 9 atributos referentes a exames microscópicos. A base de dados conta com 699 exemplos. Os dados foram originalmente obtidos a partir da Universidade de Wisconsin–Madison pelo Dr. William H. Wolberg.
3. Habitação: tenta prever os valores de moradias nos subúrbios de Boston com base em 13 atributos, dentre eles: taxa de criminalidade per capita da região, número médio de quartos e índice de acessibilidade para rodovias radiais. A base de dados conta com 506 exemplos.
4. Abalone: previsão da idade do molusco haliote a partir de medidas físicas como diâmetro, altura e peso. A base de dados conta com 8 atributos e 4177 exemplos.
5. Splicing: junções de splicing são pontos no DNA que cercam o limite entre um éxon (sequência de gene que é transcrito em RNA e mantido na molécula funcional) e um íntron (sequência não codificada dentro de um gene). O problema consiste no reconhecimento dos limites éxon/íntron (EI) ou íntron/éxon (IE) numa sequência de DNA com 60 posições de nucleotídeos. A base de dados conta com 3190 exemplos.
6. Landsat: classificação de imagens multiespectrais 3x3 com quatro faixas de espectro em uma dentre 6 classes de regiões como solo vermelho, lavoura de algodão e solo cinza úmido. A base de dados conta com 6435 exemplos.

3.6 Ambiente computacional

Para a realização dos testes, foram utilizados cinco computadores *desktop* com a seguinte configuração: processador AMD Phenom II X4 B93 de 2.8GHz, 4GB de memória DDR3 SDRAM, HD SATA de 250GB, placa de rede Gigabit Ethernet PCIe, sistema operacional Microsoft Windows XP Professional Service Pack 3. Os computadores foram conectados em uma rede Ethernet local, operando na velocidade de 1 Gbps utilizando cabos UTP (*Unshielded Twisted Pair*) categoria 5e.

4 RESULTADOS E DISCUSSÃO

Neste capítulo são apresentados os resultados e análises dos testes realizados. As seções estão divididas em duas partes, sendo a primeira referente aos testes dos algoritmos de aprendizagem e a segunda referente aos testes do modelo distribuído.

4.1 Testes dos algoritmos de aprendizagem

Para avaliar a capacidade dos algoritmos genéticos de realizar o treinamento de uma rede MLP, foram utilizadas as bases de dados: íris, câncer de mama e habitação; descritas na seção 3.5.

As RNAs utilizadas foram configuradas com a função de ativação sigmoidal. Os dados obtidos foram normalizados a fim de que o intervalo dos valores numéricos fique dentro de 0 a 1. Cada conjunto de dados foi embaralhado e dividido em dois subconjuntos: um conjunto de treinamento e um conjunto de validação, sendo este constituído de 20% da base de dados. O conjunto de validação foi utilizado após cada treinamento a fim de avaliar a capacidade de generalização da rede.

Para possibilitar a comparação entre os algoritmos, as condições de paradas utilizadas foram um tempo máximo de execução e um erro mínimo desejado. Como o AG é um algoritmo estocástico, os resultados podem variar consideravelmente entre uma execução e outra. Foram, portanto, realizados vários treinamentos para cada problema, analisando os valores médios dos resultados obtidos.

4.1.1 Classificação da flor íris

Para o problema de classificação da flor íris foram executados 20 treinamentos para cada algoritmo de aprendizagem com o tempo máximo de execução de 5 segundos e o EQM desejado de 0,005. A estrutura da RNA utilizada foi: 4 neurônios na camada de entrada, 2 neurônios na camada escondida e 3 neurônios na camada de saída.

O BP foi configurado com a taxa de aprendizado de 0,25 e a taxa de momento de 0.3. Os parâmetros dos AGs utilizados são exibidos no Quadro 3.

Quadro 3 AGs utilizados no treinamento da MLP para classificação da flor íris

Parâmetro	AG Binário	AG Real
Codificação do indivíduo	-25 a 25 com precisão de 0.05	-50 a 50
Número de indivíduos	22	14
Seleção de pais	Torneio	Torneio
Operador de recombinação	Crossover de dois pontos	Crossover estruturado
Operador de mutação	Mutação dois pontos	Mutação por neurônio
Probabilidade de mutação	12%	10%

A média e o desvio padrão dos EQMs obtidos nos treinamentos são mostrados na Tabela 1. Nas execuções o menor EQM foi obtido com a utilização do AGR, entretanto a menor média dos erros dos treinamentos foi observada com o BP. Quanto ao EQM da validação, os melhores resultados foram obtidos com a utilização do AGR.

Como é um problema de classificação, o valor da saída da rede é arredondado para 0 ou 1 e então comparado com a saída desejada. A média e o desvio padrão de acertos obtidos na classificação da flor íris são mostrados na Tabela 2.

Tabela 1 Média dos EQMs obtidos nos treinamentos para classificação da flor íris

Algoritmo	**EQM do treinamento**	**EQM da validação**
BP	0,008481 (±0,0000)	0,031684 (±0,0004)
AG Real	0,011033 (±0,0032)	0,019482 (±0,0151)
AG Binário	0,054743 (±0,0323)	0.042972 (±0,0318)

O desvio padrão é mostrado entre parênteses

A maior variância nos resultados foi observada nas execuções com o AGB onde a aplicação dos operadores genéticos podem provocar grandes mudanças no fenótipo e consequentemente gerar instabilidade no processo de treinamento. O Gráfico 1 mostra a evolução do EQM dos melhores treinamentos obtidos por cada algoritmo de aprendizagem.

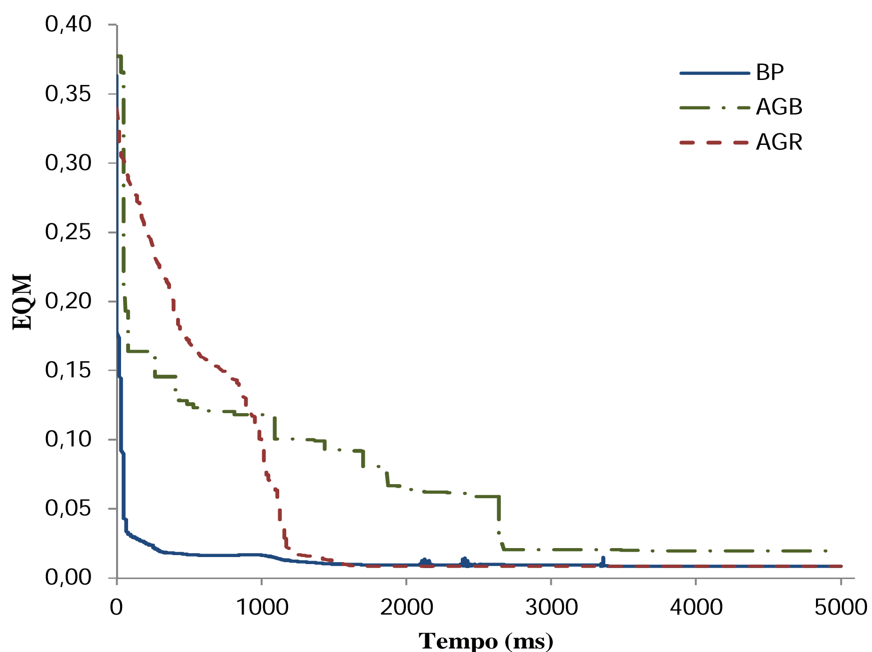

Gráfico 1 Evolução dos EQMs para classificação da flor íris

Tabela 2 Média dos acertos obtidos nos treinamentos para classificação da flor íris

Algoritmo	Acertos no treinamento (%)	Acertos na validação (%)
BP	99,16 (±0,00)	96,66 (±0,00)
AG Real	98,75 (±0,61)	97,83 (±1,58)
AG Binário	87,12 (±14,76)	88,83 (±15,53)

O desvio padrão é mostrado entre parênteses

4.1.2 Classificação do câncer de mama

Para o problema de classificação do câncer de mama, foram executados vinte treinamentos para cada algoritmo de aprendizagem com o tempo máximo de execução de 25 segundos e o EQM desejado de 0,001. A estrutura da RNA utilizada foi: 9 neurônios na camada de entrada, 5 neurônios na camada escondida e 1 neurônio na camada de saída.

O BP foi configurado com a taxa de aprendizado de 0,16 e a taxa de momento de 0,35. Os parâmetros dos AGs utilizados são exibidos no Quadro 4. A média e o desvio padrão dos EQMs obtidos nos treinamentos são mostrados na Tabela 3. A rede com o menor EQM foi obtida com o BP. O AGB foi inferior ao BP em todas as execuções, mas apresentou maior taxa de acerto médio sobre o conjunto de validação.

Quadro 4 AGs utilizados no treinamento da MLP para classificação do câncer de mama

Parâmetro	AG Binário	AG Real
Codificação do indivíduo	-25 a 25 com precisão de 0.01	-50 a 50
Número de indivíduos	20	14
Seleção de pais	Torneio	Torneio
Operador de recombinação	Crossover de dois pontos	Crossover estruturado
Operador de mutação	Mutação dois pontos	Mutação por neurônio
Probabilidade de mutação	12%	14%

Tabela 3 Média dos EQMs obtidos nos treinamentos para classificação do câncer de mama

Algoritmo	EQM do treinamento	EQM da validação
BP	0,003732 (±0,0012)	0,011544 (±0,0057)
AG Real	0,005761 (±0,0013)	0,010313 (±0,0038)
AG Binário	0,011108 (±0,0015)	0,006517 (±0,0038)

O desvio padrão é mostrado entre parênteses

A média e o desvio padrão de acertos obtidos na classificação do câncer de mama são mostrados na Tabela 4. Observou-se que as redes com altas taxas de acerto no conjunto de treinamento possuíram menores taxas de acertos no conjunto de validação, indicando um possível problema de *overfitting*. O Gráfico 2 mostra a evolução do EQM nos melhores treinamentos realizados com os algoritmos de aprendizagem.

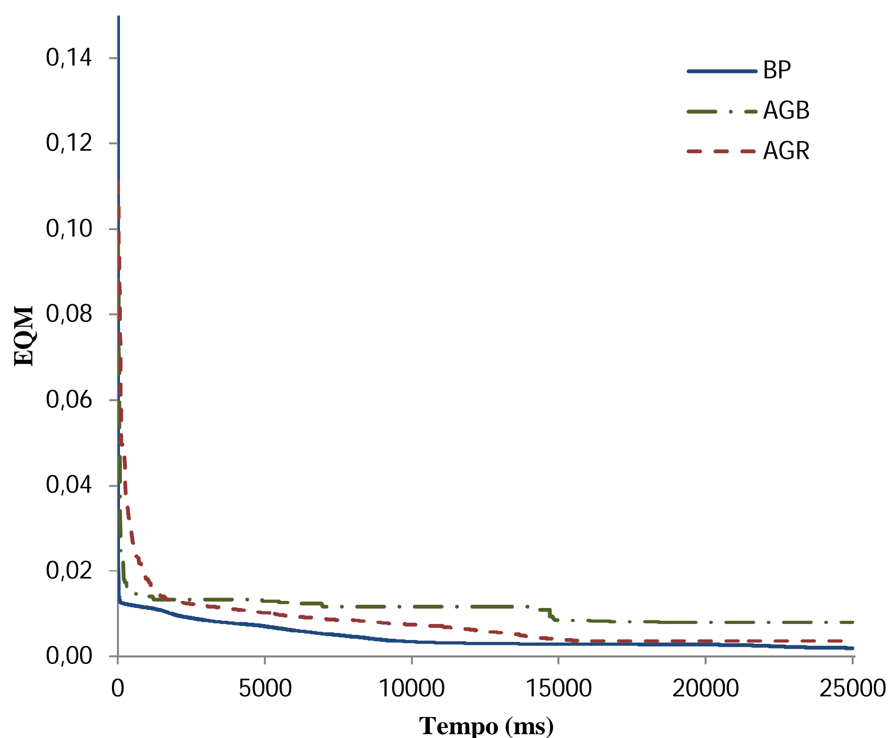

Gráfico 2 Evolução do EQM para classificação do câncer de mama

Tabela 4 Média dos acertos obtidos nos treinamentos para classificação do câncer de mama

Algoritmo	**Acertos no treinamento (%)**	**Acertos na validação (%)**
BP	99,25 (±0,25)	97,32 (±1,31)
AG Real	98,82 (±0,30)	97.60 (±0,82)
AG Binário	97.63 (±0,38)	98.50 (±0,84)

O desvio padrão é mostrado entre parênteses

4.1.3 Previsão de valores de habitação

Para o problema de previsão dos valores de habitação foram executados 20 treinamentos para cada algoritmo de aprendizagem com o tempo máximo de execução de 1 minuto e o EQM desejado de 0,0001. A estrutura da RNA utilizada foi: 13 neurônios na camada de entrada, 5 neurônios na camada escondida e 1 neurônio na camada de saída.

O BP foi configurado com a taxa de aprendizado de 0,15 e a taxa de momento de 0,26. Os parâmetros dos AGs utilizados são exibidos no Quadro 5.

Quadro 5 AGs utilizados no treinamento da MLP para previsão de valores de habitação

Parâmetro	**AG Binário**	**AG Real**
Codificação do indivíduo	-15 a 15 com precisão de 0.0065	-50 a 50
Número de indivíduos	15	20
Seleção de pais	Torneio	Torneio
Operador de recombinação	Crossover de dois pontos	Crossover estruturado
Operador de mutação	Mutação por peso	Mutação por neurônio
Probabilidade de mutação	5%	12%

A média e o desvio padrão dos EQMs obtidos nos treinamentos são mostrados na Tabela 5. Para este problema de regressão, o BP apresentou resultados melhores tanto no treinamento quanto no processo de validação.

Tabela 5 Média dos EQMs obtidos nos treinamentos para previsão de valores de habitação

Algoritmo	EQM do treinamento	EQM da validação
BP	0,001096 (±0,0000)	0,002186 (±0,0002)
AG Real	0,002104 (±0,0004)	0,003140 (±0,0008)
AG Binário	0,007267 (±0,0008)	0,006897 (±0,0011)

O desvio padrão é mostrado entre parênteses

A partir dos melhores resultados obtidos por cada algoritmo, foram gerados gráficos de dispersão com os valores desejados e os valores fornecidos pelas RNAs. Os Gráficos 3, 4 e 5 mostram, respectivamente, os melhores resultados obtidos com os algoritmos BP, AGB e AGR; aplicados sobre o conjunto de validação. É possível verificar que a rede treinada com o BP realizou previsões mais precisas do que as que utilizaram as versões de AGs.

O Gráfico 5 mostra a evolução do EQM nos melhores treinamentos realizados com os algoritmos de aprendizagem.

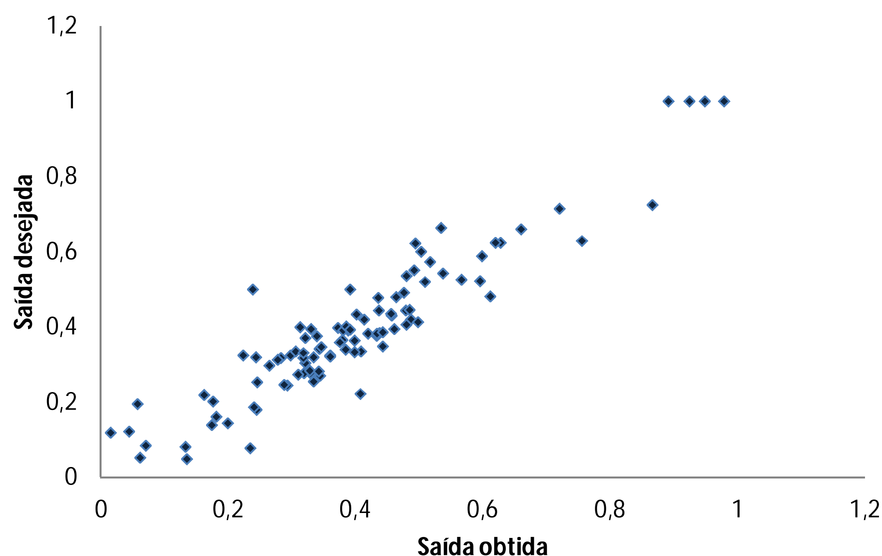

Gráfico 3 Previsão dos valores de habitação fornecidos pelo melhor treinamento utilizando BP

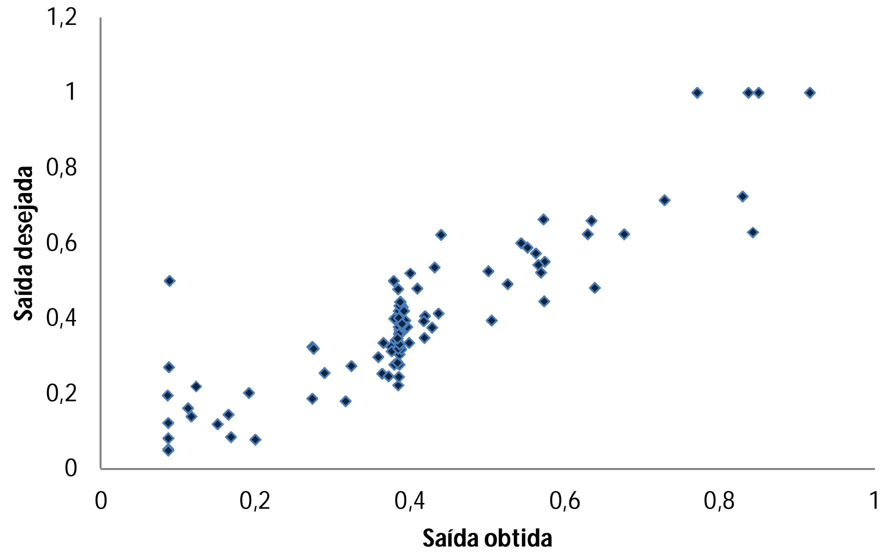

Gráfico 4 Previsão dos valores de habitação fornecidos pelo melhor treinamento utilizando AGB

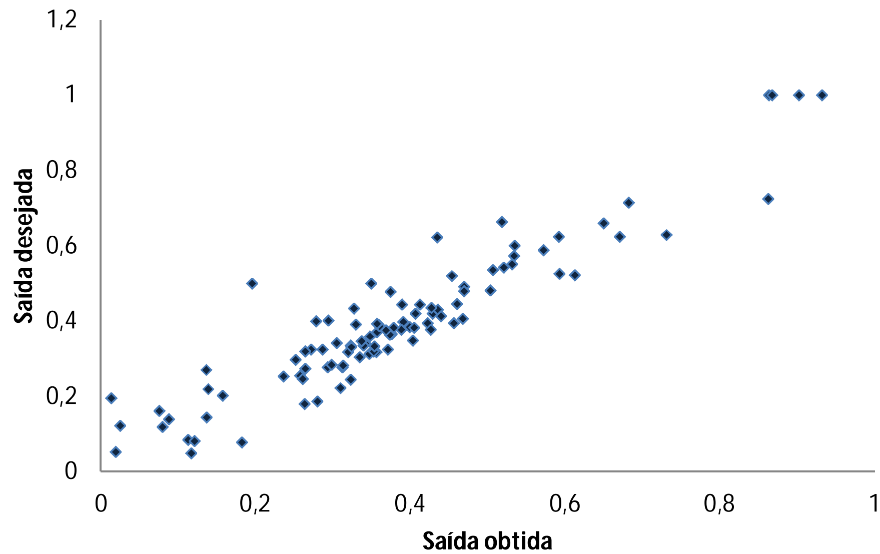

Gráfico 5 Previsão dos valores de habitação fornecidos pelo melhor treinamento utilizando AGR

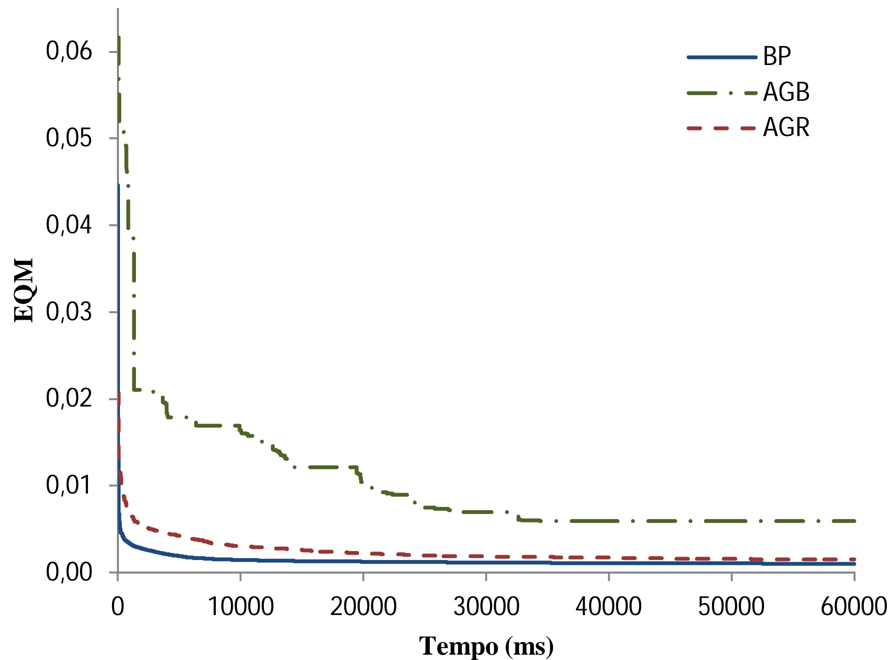

Gráfico 6 Evolução do EQM para a previsão de valores de habitação

4.1.4 Análise dos resultados

Nos três problemas o BP apresentou melhores taxas médias de acertos sobre os conjuntos de treinamento. Quanto à validação, os AGs forneceram melhores taxas médias de acertos nos problemas de classificação. O AGB obteve o pior desempenho médio nos testes além de exibir alta instabilidade no processo de aprendizagem. No AG com codificação real, o operador crossover estruturado foi o que apresentou os melhores resultados, entretanto o custo computacional é maior devido à etapa necessária para ordenar as regiões de ativação da rede neural. Observou-se que o algoritmo de aprendizagem baseado no gradiente de erro é mais preciso e fornece boas soluções em menor tempo de treinamento, entretanto, o algoritmo genético é capaz de encontrar soluções com

boa capacidade de generalização e pode fornecer uma alternativa ao treinamento de redes MLPs.

4.2 Teste do modelo distribuído

Para avaliar o modelo paralelo do algoritmo genético foram utilizadas as bases de dados: abalone, splicing e landsat; descritas na seção 3.5. A configuração das redes e a preparação das bases de dados foram realizadas da forma descrita na seção 4.1. O algoritmo de aprendizagem utilizado foi o AG com codificação real uma vez que este apresentou melhores resultados que o AG com codificação binária nos testes sequenciais realizados.

A análise do desempenho do modelo distribuído foi realizada a partir da comparação do tempo médio necessário para se alcançar um determinado EQM nas execuções com diferentes números de computadores. Foi adotada a topologia de migração em anel semelhante ao que é ilustrado na Figura 21.

4.2.1 Previsão da idade do abalone

Para o problema de previsão da idade do abalone as saídas foram agrupadas em cinco classes definidas com base na distribuição das frequências observada na base de dados. Foram executados cinco treinamentos para cada teste de desempenho com o tempo máximo de execução de 10 minutos e o EQM desejado de 0,265. A estrutura da RNA utilizada foi: 8 neurônios na camada de entrada, 6 neurônios na primeira camada escondida, 5 neurônios na segunda camada escondida e 5 neurônios na camada de saída. O período de migração utilizado foi de 25 gerações. Os parâmetros do AGR são exibidos no Quadro 6.

Quadro 6 Parâmetros do AGR utilizados no problema abalone

Parâmetro	AG Real
Codificação do indivíduo	-50 a 50
Número de indivíduos	15
Seleção de pais	Torneio
Operador de recombinação	Crossover estruturado
Operador de mutação	Mutação por neurônio
Probabilidade de mutação	12%

A Tabela 6 mostra os resultados dos testes realizados com diferentes números de computadores. O maior ganho em desempenho foi observado com a utilização de três máquinas. A evolução do tempo médio dos treinamentos é exibida no Gráfico 7.

Tabela 6 Resultados do modelo paralelo aplicado no problema abalone

Nº de ilhas	Tempo de treinamento (ms)				
	Média	Desvio padrão	Maior	Menor	Ganho
1	319737,60	±137656,48	567000	141360	-
2	293015,40	±127934,38	541437	193797	8,36%
3	227103,00	±45538,07	293438	173031	22,49%
4	216597,00	±47613,40	273235	141344	4,63%
5	211400,00	±9004,77	226688	200000	2,40%

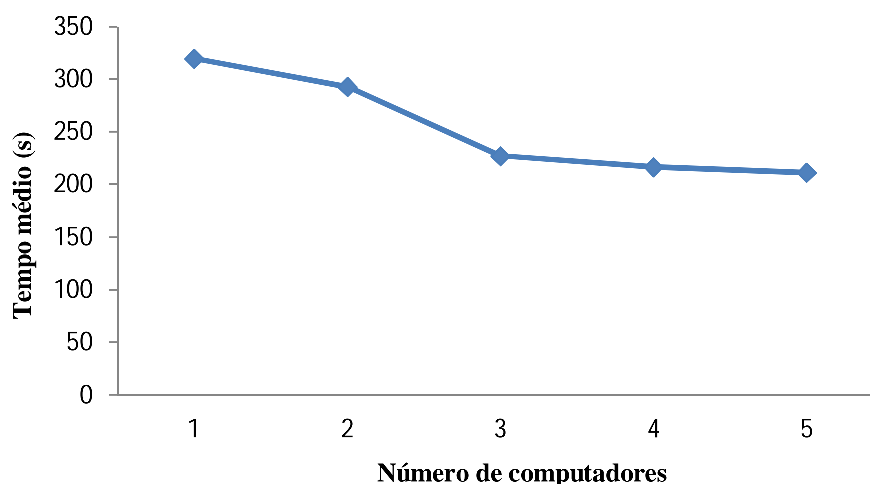

Gráfico 7 Evolução do tempo médio de treinamento em relação ao número de ilhas para resolução do problema abalone

4.2.2 Classificação da junção de splicing

Para a resolução do problema de reconhecimento de junção de *splicing*, foi utilizada uma RNA com 60 neurônios na camada de entrada, 10 neurônios na camada escondida e 3 neurônios na camada de saída. O tempo máximo de execução foi de 15 minutos, o EQM desejado foi 0,18 e o período de migração utilizado foi de 20 gerações. Os parâmetros do AGR são exibidos no Quadro 7.

Foram executados cinco treinamentos para cada número de ilhas. Os resultados são mostrados na Tabela 7 e a evolução do tempo médio de treinamento é exibida no Gráfico 8. Neste problema o maior ganho de desempenho observado foi de 13,92% com a utilização de três ilhas.

Quadro 7 Parâmetros do AGR utilizado no problema splicing

Parâmetro	AG Real
Codificação do indivíduo	-15 a 15
Número de indivíduos	12
Seleção de pais	Torneio
Operador de recombinação	Crossover estruturado
Operador de mutação	Mutação por neurônio
Probabilidade de mutação	10%

Tabela 7 Resultados do modelo paralelo aplicado no problema splicing

Nº de ilhas	Tempo de treinamento (ms)				
	Média	**Desvio padrão**	**Maior**	**Menor**	**Ganho**
1	541247,20	±99873,17	698157	415391	-
2	508712,40	±175339,21	852140	368344	6,01%
3	437918,80	±85602,08	516719	278234	13,92%
4	425306,20	±74069,23	548954	343109	2,88%
5	415159,40	±62544,53	515390	347344	2,39%

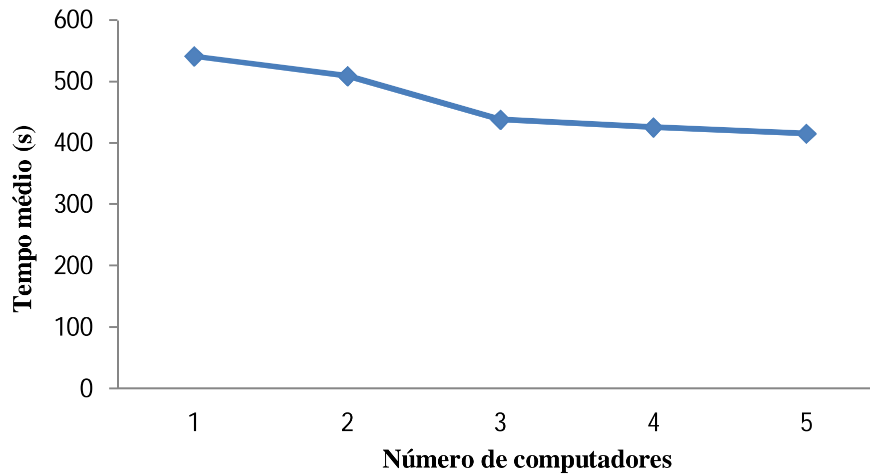

Gráfico 8 Evolução do tempo médio de treinamento em relação ao número de ilhas para resolução do problema splicing

4.2.3 Classificação de imagens multiespectrais

Para o problema landsat, foi utilizada uma RNA com 36 neurônios na camada de entrada, 9 neurônios na camada escondida e 6 neurônios na camada de saída. O tempo máximo de execução foi de 17 minutos, o EQM desejado foi 0,12 e o período de migração utilizado foi de 20 gerações. Os parâmetros do AGR são exibidos no Quadro 8.

Quadro 8 Parâmetros do AGR utilizado no problema landsat

Parâmetro	AG Real
Codificação do indivíduo	-20 a 20
Número de indivíduos	13
Seleção de pais	Torneio
Operador de recombinação	Crossover estruturado
Operador de mutação	Mutação por neurônio
Probabilidade de mutação	6%

Foram executados cinco treinamentos para cada número de ilhas. Os resultados são mostrados na Tabela 8, a evolução do tempo médio de treinamento é exibida no Gráfico 9.

Tabela 8 Resultados do modelo paralelo aplicado no problema landsat

Nº de ilhas	Tempo de treinamento (ms)				
	Média	Desvio padrão	Maior	Menor	Ganho
1	627062,40	±215383,15	971625	340171	-
2	563975,20	±182096,28	824406	342594	10,06%
3	513906,00	±259755,60	1020297	323281	8,88%
4	456962,40	±142784,70	622797	221531	11,08%
5	390550,00	±56876,03	477797	301047	14,53%

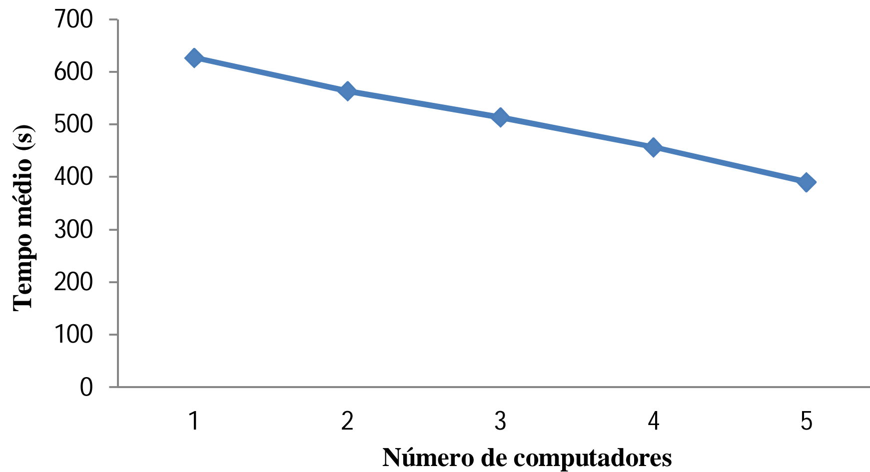

Gráfico 9 Evolução do tempo médio de treinamento em relação ao número de ilhas para resolução do problema landsat

4.2.4 Análise dos resultados

O modelo paralelo apresentou pequenos ganhos de desempenho em relação ao modelo sequencial. Com a utilização dos cinco computadores, a redução total de tempo observada foi de 33,88%, 23,30% e 37,72% para os problemas abalone, splicing e landsat, respectivamente. A configuração que resultou em maiores ganhos em relação ao tempo médio de treinamento foi a utilização de três ilhas para os problemas abalone e splicing e de cinco ilhas para o problema landsat.

A abordagem paralela aumentou a estabilidade do processo de aprendizagem o que pode ser observado a partir da redução do desvio padrão do tempo de treinamento na medida em que o número de ilhas utilizadas aumenta. Nos três problemas a utilização de cinco ilhas resultou na menor variação dos tempos de treinamento, entretanto as melhores execuções foram obtidas em diferentes configurações do modelo.

5 CONCLUSÃO

A utilização de algoritmos genéticos no treinamento das RNAs apresentou desempenho inferior ao BP em relação aos conjuntos de treinamentos dos problemas utilizados. Entretanto, os AEs forneceram bons resultados quanto à capacidade de generalização das RNA aplicadas sobre os problemas de classificação de dados.

É importante observar que a utilização de AGs possui uma desvantagem relacionada com quantidade de parâmetros necessários na execução de um processo evolutivo. Estes parâmetros devem ser configurados de modo a equilibrar a capacidade de exploração e evolução do algoritmo e, portanto, podem aumentar a complexidade do processo de aprendizagem de uma RNA. Também é importante considerar as limitações dos operadores genéticos em *otimizar* uma RNA a partir de pequenas alterações nos pesos que estão associados com as maiores estimativas de erros. Neste aspecto é interessante adotar uma abordagem híbrida com a utilização de um algoritmo de aprendizagem baseado no gradiente do erro.

O modelo de ilhas implementado permite aumentar a velocidade do treinamento e a probabilidade de encontrar boas soluções, contudo o ganho médio em desempenho é pequeno. Desta forma, para trabalhos futuros, outros modelos podem ser utilizados e então comparados com a abordagem desenvolvida.

REFERÊNCIAS

ALBA, E.; TOMASSINI, M. Parallelism and evolutionary algorithms. **IEEE Transactions on Evolutionary Computation**. v. 6, n. 5, p. 443-462, 2002.

ARBIB, M. A. **The handbook of brain theory and neural networks**. Cambridge: The MIT Press, 2003. 1309 p.

BISHOP, C. M. Neural networks and their applications. **Review of Scientific Instruments**. v. 65, n. 6, p. 1803-1831, 1994.

BONISSONE, P. P. Evolutionary Algorithms + Domain Knowledge = Real-World Evolutionary Computation. **IEEE Transactions on Evolutionary Computation**. v. 10, n. 3, p. 256-280, 2006.

BRAGA, A. P.; CARVALHO, A. P. L. F.; LUDERMIR, T. B. **Redes neurais artificiais**: teoria e aplicações. 2. ed. Rio de Janeiro: LTC, 2007. 226 p.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Sistemas distribuídos**: conceitos e projetos. 4. ed. Porto Alegre: Bookman, 2007.

DEITEL, H. M.; DEITEL, P. J. **Java**: como programar. Tradução: Edson Furmankiewicz. São Paulo: Pearson Prentice Hall, 2005.

EIBEN, A. E.; SMITH J. E. **Introduction to Evolutionary Computing**. Heidelberg: Springer, 2003. 299 p.

FRANK, A; ASUNCION, A. UCI Machine Learning Repository. Disponível em: < http://archive.ics.uci.edu/ml>. Acesso em: 02 de novembro de 2011.

FOO, S. K.; SARATCHANDRAN, P.; SUNDARARAJA, N. Parallel implementation of backpropagation neural networks on a heterogeneous array of transputers. **IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics**. v. 27, n. 1, p. 118-126, 1997.

HANCOCK, P. J. B. Genetic algorithms and permutation problems: A comparison of recombination operators for neural net structure specification. In: **Proc. Int. Workshop Combinations of Genetic Algorithms and Neural Networks (COGANN-92)**, D. Whitley and J. D. Schaffer, Eds. Los Alamitos, CA: IEEE Computer Soc., 1992, p. 108–122.

HAYKIN, Simon. **Neural networks and learning machines**. 3rd ed. Upper Saddle River, New Jersey: Pearson Education Inc., 2009. 906 p.

HEATON, J. **Introduction to Neural Networks for C#**. 2nd ed. St Louis: Heaton Research Inc., 2008. 426 p.

IRANI, R.; NASINI, R. Evolving neural network using real coded genetic algorithm for permeability estimation of the reservoir. **Expert Systems with Applications**, v. 38, n. 8, p 9862-9866, 2011.

JASON, D. J.; FRENZEL, F. Training product unit neural networks with genetic algorithms. **IEEE Expert**, v. 8, n. 5, p. 26-36, 1993.

LEUNG, F. H. F. et al. Tuning of the structure and parameters of a neural network using an improved genetic algorithm. **IEEE Transactions on Neural Networks**. v. 14, n. 1, p. 79-88, 2003.

LI, T.;CHEN, Y.; Li, G. An optimized method for electric power system harmonic measurement based on back-propagation neural network and modified genetic algorithm. In: **3rd International Conference on Power Electronics Systems and Applications**. p. 1-5, 2009.

LINDEN, R.. **Algoritmos Genéticos**: uma importante ferramenta da inteligência computacional. Rio de Janeiro: Brasport, 2008. 348 p.

MAJDI, A; BEIKE, M. Evolving neural network using a genetic algorithm for predicting the deformation modulus of rock masses. **International Journal of Rock Mechanics and Mining Sciences**. v. 47, n. 2, p. 246-253, 2010.

MEIRELES, M. R. G. ALMEIDA, E. M.; SIMÕES, M. G. A Comprehensive Review for Industrial Applicability of Artificial Neural Networks. **IEEE Transactions on Industrial Electronics**. v. 50, n. 3, p. 585-601, 2003.

MONTANA, D.; DAVIS, L. Training feedforward neural networks using genetic algorithms. In: **11th Int. Joint Conf. Artificial Intelligence**. San Mateo, CA: Morgan Kaufmann, p. 762–767, 1989.

ORACLE. Object-Oriented Programming Concepts. Disponível em: <http://download.oracle.com/javase/tutorial/java/concepts/index.html>. Acesso em: 25 de outubro de 2011.

______. The History of Java Technology. Disponível em: <http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html>. Acesso em: 25 de outubro de 2011.

______. What Is a Socket?. Disponível em: <http://download.oracle.com/javase/tutorial/networking/sockets/definition.html>. Acesso em: 25 de outubro de 2011.

PAN, Shing-Tai; WU, Chih-Hung; LAI, Chih-Chin. The Application of Improved Genetic Algorithm on the Training of Neural Network for Speech Recognition. In: **Proceedings of the Second International Conference on Innovative Computing, Informatio and Control**. Washington: IEEE Computer Society, p. 168-172, 2007.

REZENDE, S. O. **Sistemas Inteligentes**: fundamentos e aplicações. Barueri: Manole, 2005. 517 p.

RIESSEN, G. A.; WILLIAMS, G. J.; YAO, Xin. PEPNet: Parallel evolutionary programming for constructing artificial neural networks. In: **Lecture Notes in Computer Science**. Berlin: Springer-Verlag, p. 35-45, 1997.

VEELENTURF, L. P. J. **Analysis and applications of artificial neural networks**. United Kingdom: Prentice Hall International, 1995. 259 p.

WU, T-K et al. On the parallelization and optimization of the genetic-based ANN classifier for the diagnosis of students with learning disabilities . In: **2010 IEEE International Conference on Systems, Man, and Cybernetics**, p. 4263-4269, 2010.

YAO, Xin. Evolving Artificial Neural Networks. In: **Proceedings of the IEEE**, v. 87, n. 9, p. 1423-1447, 1999.

APÊNDICE A – Implementação da RNA

```
/*
 * rna.FuncaoAtivacao
 */

package rna;

public abstract class FuncaoAtivacao {

    public abstract double ativacao(double entrada);

    public abstract double derivada(double entrada);
}
```

```
/*
 * rna.mlp. CamadaMLP
 */

package rna.mlp;

import rna.FuncaoAtivacao;

public class CamadaMLP {

    protected FuncaoAtivacao funcaoAtivacao;
    protected double[][] erros, pesos, vPesos;
    protected double[] entrada, saida, soma, somaEntradas;
    protected int numNeuronios, numEntradas;

    public CamadaMLP(double[] entrada, int num_neuronios, FuncaoAtivacao
funcaoAtivacao){
        this.entrada = entrada;
        // +1 referente ao bias
        this.numEntradas = entrada.length + 1;
        this.numNeuronios = num_neuronios;
        this.funcaoAtivacao = funcaoAtivacao;
        this.saida = new double[num_neuronios];
        this.soma = new double[num_neuronios];
        this.somaEntradas = new double[this.numEntradas];
        this.erros = new double[num_neuronios][this.numEntradas];
        this.pesos = new double[num_neuronios][this.numEntradas];
        this.vPesos = new double[num_neuronios][this.numEntradas];
    }
```

```
protected void inicialiazarPesos(){
    for (int n=0; n<numNeuronios; n++){
        int v;
        for (v=0; v<entrada.length; v++){
            pesos[n][v] = Math.random() * 2 - 1;
            vPesos[n][v] = 0;
        }
        pesos[n][v] = Math.random() * 2 - 1;
        vPesos[n][v] = 0;
    }
}

public void inicializaErros(){
    for (int n=0; n<numNeuronios; n++){
        for (int v=0; v<=entrada.length; v++){
            erros[n][v] = 0;
        }
    }
}

public double[] calcularSinalErro(double[] e) {
    double[] sinais = new double[entrada.length];
    double gradiente;
    int v;

    gradiente = e[0] * funcaoAtivacao.derivada(soma[0]);

    for (v=0; v<entrada.length; v++){
        erros[0][v] += entrada[v] * gradiente;
        sinais[v] = gradiente * pesos[0][v];
    }
    erros[0][v] += gradiente;

    for (int n=1; n<numNeuronios; n++){
        gradiente = e[n] * funcaoAtivacao.derivada(soma[n]);

        for (v=0; v<entrada.length; v++){
            erros[n][v] += entrada[v] * gradiente;
            sinais[v] += gradiente * pesos[n][v];
        }
        erros[n][v] += gradiente;
    }

    return sinais;
}
```

```
public void reduzirErros(int t){
   for (int n=0; n<numNeuronios; n++){
      for (int v=0; v<=entrada.length; v++){
         erros[n][v] = erros[n][v]/t;
      }
   }
}

public void inicializaSomas(){
   for (int n=0; n<somaEntradas.length; n++){
      somaEntradas[n] = 0;
   }
}

public void reduzirSomas(int t){
   for (int n=0; n<somaEntradas.length; n++){
      somaEntradas[n] = somaEntradas[n] / t;
   }
}

public void propagarSinal(){
   for (int n=0; n<numNeuronios; n++){
      int v;
      soma[n] = 0;
      for (v=0; v<entrada.length; v++){
         soma[n] += entrada[v] * pesos[n][v];
      }
      soma[n] += pesos[n][v];
      saida[n] = funcaoAtivacao.ativacao(soma[n]);
   }
}

public void propagarSinalESomar(){
   int v,n;
   n = 0;
   soma[n] = 0;
   for (v=0; v<entrada.length; v++){
      somaEntradas[v] += entrada[v];
      soma[n] += entrada[v] * pesos[n][v];
   }
   somaEntradas[v] += 1;
   soma[n] += pesos[n][v];
   saida[n] = funcaoAtivacao.ativacao(soma[n]);
   n++;

   for (; n<numNeuronios; n++){
```

```
        soma[n] = 0;
        for (v=0; v<entrada.length; v++){
            soma[n] += entrada[v] * pesos[n][v];
        }
        soma[n] += pesos[n][v];
        saida[n] = funcaoAtivacao.ativacao(soma[n]);
    }
}

public double[] getEntrada() {
    return entrada;
}

public double[][] getErros() {
    return erros;
}

public int getNumEntradas() {
    return numEntradas;
}

public FuncaoAtivacao getFuncaoAtivacao() {
    return funcaoAtivacao;
}

public int getNumNeuronios() {
    return numNeuronios;
}

public double[][] getPesos() {
    return pesos;
}

public double[] getSaida() {
    return saida;
}

public double[] getSoma() {
    return soma;
}

public double[] getSomas() {
    return somaEntradas;
}

public double[][] getVPesos() {
```

```
        return vPesos;
    }

    public void setPesos(double[][] pesos) throws CamadaMLPException {
        if (pesos.length != numNeuronios || pesos[0].length != entrada.length + 1){
            throw new CamadaMLPException("Matriz de pesos invalida");
        }
        this.pesos = pesos;
    }

    public void setVPesos(double[][] v_pesos) throws CamadaMLPException {
        if (pesos.length != numNeuronios || pesos[0].length != entrada.length + 1){
            throw new CamadaMLPException("Matriz de variacao pesos invalida");
        }
        this.vPesos = v_pesos;
    }

    public void setPeso(int n, int e, double peso) {
        this.pesos[n][e] = peso;
    }
}


/*
 * rna.mlp.MLP
 */

package rna.mlp;

public class MLP {

    protected CamadaMLP[] camadas;
    protected double[] entrada;
    protected int[][] estrutura;
    protected int numPesos, numNeuronios, numEntradas;

    public MLP(CamadaMLP[] camadas){
        this.entrada = camadas[0].getEntrada();
        this.camadas = camadas;
        numPesos = 0;
        numNeuronios = 0;
        numEntradas = 0;
        estrutura = new int[camadas.length][2];
        int e, n;
        for (int c=0; c<camadas.length; c++){
            n = camadas[c].getNumNeuronios();
```

```
        e = camadas[c].getNumEntradas();
        estrutura[c][0] = n;
        estrutura[c][1] = e;
        numEntradas += e;
        numNeuronios += n;
        numPesos += e * n;
    }
}

public void inicializarPesos(){
    for (int c=0; c<camadas.length; c++){
        camadas[c].inicialiazarPesos();
    }
}

public void processarSinal(){
    int c;
    for (c=0; c<camadas.length; c++){
        camadas[c].propagarSinal();
    }
}

public void processarSinalESomar(){
    int c;
    for (c=0; c<camadas.length; c++){
        camadas[c].propagarSinalESomar();
    }
}

public double[][] processarSinal(double[][] dados){
    double[] saida = getSaida();
    double[][] saidas = new double[dados.length][saida.length];
    for (int d=0; d<dados.length; d++){
        System.arraycopy(dados[d], 0, entrada, 0, entrada.length);
        processarSinal();
        System.arraycopy(saida, 0, saidas[d], 0, saida.length);
    }

    return saidas;
}

public void calcularSinalErro(double[] erros, double total){
    int c = camadas.length - 1;
    for (; c>=0; c--){
        erros = camadas[c].calcularSinalErro(erros);
    }
```

```
}

public void atualizarPesos(double[] valores){
    int p=0;
    for (int c=0; c<camadas.length; c++){
        double[][] pesos = camadas[c].getPesos();
        double[][] vPesos = camadas[c].getVPesos();
        for (int i=0; i<pesos.length; i++){
            for (int j=0; j<pesos[i].length; j++){
                pesos[i][j] = valores[p];
                vPesos[i][j] = 0;
                p++;
            }
        }
    }
}

public double[] getSomas(){
    double[] somas = new double[this.numEntradas];
    int n = 0;
    for (int c=0; c<camadas.length; c++){
        double[] aux = camadas[c].getSomas();
        System.arraycopy(aux, 0, somas, n, aux.length);
        n += aux.length;
    }
    return somas;
}

public double[] getPesos(){
    double[] pesos = new double[numPesos];
    int p=0;
    for (int c=0; c<camadas.length; c++){
        double[][] aux = camadas[c].getPesos();
        for (int i=0; i<aux.length; i++){
            for (int j=0; j<aux[i].length; j++){
                pesos[p] = aux[i][j];
                p++;
            }
        }
    }
    return pesos;
}

public void inicializarErros() {
    for (int c=0; c<camadas.length; c++){
        camadas[c].inicializaErros();
```

```
        }
    }

    public void inicializarSomas() {
        for (int c=0; c<camadas.length; c++){
            camadas[c].inicializaSomas();
        }
    }

    public void reduzirErros(int length) {
        for (int c=0; c<camadas.length; c++){
            camadas[c].reduzirErros(length);
        }
    }

    public void reduzirSomas(int length) {
        for (int c=0; c<camadas.length; c++){
            camadas[c].reduzirSomas(length);
        }
    }

    public CamadaMLP[] getCamadas() {
        return camadas;
    }

    public double[] getEntrada() {
        return entrada;
    }

    public int[][] getEstrutura(){
        return estrutura;
    }

    public int getNumNeuronios() {
        return numNeuronios;
    }

    public int getNumPesos(){
        return numPesos;
    }

    public double[] getSaida() {
        return camadas[camadas.length-1].getSaida();
    }
}
```

APÊNDICE B – Implementação do Backpropagation

```
package rna.mlp.treinamento;

import ag.Ilha;
import ag.Individuo;
import ag.IndividuoPadrao;
import aplicacao.ModeloIlhas;
import java.util.Collection;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import rna.FuncaoAtivacao;
import rna.mlp.CamadaMLP;
import rna.mlp.MLP;

public class Backpropagation extends AprendizagemSupervisionada
implements Ilha {

    public static class Iteracao implements IteracaoAprendizagem {

        private double eqm;
        private long tempo;
        private int epoca;

        public Iteracao(double eqm, long tempo, int epoca){
            this.eqm = eqm;
            this.tempo = tempo;
            this.epoca = epoca;
        }

        @Override
        public double getEQM() {
            return eqm;
        }

        @Override
        public long getTempo() {
            return tempo;
        }

        @Override
        public int getIteracao() {
            return epoca;
        }
    }
```

```
    public static final int GERACAO = 10;

    private ModeloIlhas agp;
    private Individuo<double[], Object> individuo;
    private IteracaoAprendizagem estadoCorrente;
    private String origem;
    private double[] erro;
    private double taxaAprendizagem, momento;
    private long inicio;
    private int epoca;

    public Backpropagation(MLP rna, double taxaAprendizagem, double
momento) {
        super(rna);
        this.iteracoes = new LinkedList<IteracaoAprendizagem>();
        this.taxaAprendizagem = taxaAprendizagem;
        this.momento = momento;
        this.erro = new double[saida.length];
    }

    @Override
    public void configuraModelo(ModeloIlhas agp){
        this.agp = agp;
        agp.setIteracoes(iteracoes);
        origem = agp.getOrigem();
    }

    @Override
    public void treinar(double[][] padroes, CondicaoParada condicaoParada) {
        if (agp == null){
            treinamentoSequencial(padroes, condicaoParada);
        } else {
            treinamentoParalelo(padroes, condicaoParada);
        }
    }

    public void treinamentoSequencial(double[][] padroes, CondicaoParada
condicaoParada) {
        double eqm;
        int cont;
        cont = 0;
        epoca = 0;
        inicio = System.currentTimeMillis();

        if (iteracoes.size() == 0){
```

```
            estadoCorrente = new Iteracao(calcularEMQ(padroes), 0, 0);
            iteracoes.add(estadoCorrente);
        }
        if (individuo == null){
            individuo = new IndividuoPadrao(rna.getPesos(), this.agp.getOrigem(), 0,
-estadoCorrente.getEQM());
        }

        while (condicaoParada.condicaoParada(estadoCorrente)){
            for (int p=0; p<padroes.length; p++){
                calcularErro(padroes[p], erro);
                backpropagationOnline();
            }

            eqm = calcularEMQ(padroes);
            estadoCorrente = new Iteracao(eqm, System.currentTimeMillis() - inicio,
epoca);
            iteracoes.add(estadoCorrente);

            epoca++;
            cont++;
            eventoIteracao(estadoCorrente);
        }
    }

    public void treinamentoParalelo(double[][] padroes, CondicaoParada
condicaoParada) {
        double eqm;
        int cont;
        cont = 0;
        epoca = 0;
        inicio = System.currentTimeMillis();

        if (iteracoes.size() == 0){
            estadoCorrente = new Iteracao(calcularEMQ(padroes), 0, epoca);
            iteracoes.add(estadoCorrente);
        }
        if (individuo == null){
            individuo = new IndividuoPadrao(rna.getPesos(), origem, 0, -
estadoCorrente.getEQM());
        }

        while (condicaoParada.condicaoParada(estadoCorrente)){
            if (cont == GERACAO){
                individuo.codificar(this.rna.getPesos());
                individuo.atualizarFitness(-estadoCorrente.getEQM());
```

```
                individuo.atualizarIdade(individuo.idade()+1);
                agp.migracao();
                cont = 0;
            }

            for (int p=0; p<padroes.length; p++){
                calcularErro(padroes[p], erro);
                backpropagationOnline();
            }

            eqm = calcularEMQ(padroes);
            epoca++;
            cont++;

            estadoCorrente = new Iteracao(eqm, System.currentTimeMillis() - inicio,
epoca);
            iteracoes.add(estadoCorrente);
            eventoIteracao(estadoCorrente);
        }
        agp.fimExecusao();
    }

    @Override
    public double getErroAtual() {
        return estadoCorrente.getEQM();
    }

    @Override
    public int getEpoca() {
        return epoca;
    }

    private void backpropagationOnline() {
        // Ultima camada
        int c = this.camadas.length - 1;
        CamadaMLP camada = this.camadas[c];
        FuncaoAtivacao ativacao = camada.getFuncaoAtivacao();
        double[] soma = camada.getSoma();
        double[][] pesos = camada.getPesos();
        double[][] v_pesos = camada.getVPesos();

        // Entrada da ultima camada
        double[] entrada_j = camada.getEntrada();
        int tn = camada.getNumNeuronios();

        // Vetor para calculo do gradiente dos pesos da ultima camada
```

```
        double[] gradiente_j = new double[tn];

        // Vetor para calculo do gradiente dos pesos da penultima camada
        double[] gradiente_i = new double[entrada_j.length];

        // Inicializa o gradiente da penultima camada
        int i;
        for (i=0; i<entrada_j.length; i++){
            gradiente_i[i] = 0;
        }

        // - Atualizacao dos pesos da camada de saida
        for (int n=0; n<tn; n++){
            // Calculo do gradiente * (-1)
            gradiente_j[n] = erro[n] * ativacao.derivada(soma[n]);

            for (i=0; i<entrada_j.length; i++){
                // Somatorio para calculo dos gradientes da camada anterior
                gradiente_i[i] += gradiente_j[n] * pesos[n][i];

                // Atualizacao o peso
                v_pesos[n][i] = taxaAprendizagem * gradiente_j[n] * entrada_j[i] +
momento * v_pesos[n][i];
                pesos[n][i] = pesos[n][i] + v_pesos[n][i];
            }

            // Atualizacao do bias
            v_pesos[n][i] = taxaAprendizagem * gradiente_j[n] + momento *
v_pesos[n][i];
            pesos[n][i] = pesos[n][i] + v_pesos[n][i];
        }

        // - Atualizacao dos pesos das camadas escondidas
        c--;
        while (c >= 0){
            camada = this.camadas[c];
            ativacao = camada.getFuncaoAtivacao();
            entrada_j = camada.getEntrada();
            i = entrada_j.length;
            soma = camada.getSoma();
            pesos = camada.getPesos();
            v_pesos = camada.getVPesos();

            gradiente_j = gradiente_i;
            gradiente_i = new double[i];
```

```
        for (i=0; i<entrada_j.length; i++){
            gradiente_i[i] = 0;
        }

        // Atualizacao dos pesos da camada escondida
        for (int n=0; n<camada.getNumNeuronios(); n++){
            // Calculo do gradiente
            gradiente_j[n] = ativacao.derivada(soma[n]) * gradiente_j[n];

            for (i=0; i<entrada_j.length; i++){
                // Somatorio para calculo dos gradientes da camada anterior
                gradiente_i[i] += gradiente_j[n] * pesos[n][i];

                // Atualiza a variacao do peso (utilizado para o calculo do momento)
                v_pesos[n][i] = taxaAprendizagem * gradiente_j[n] * entrada_j[i] +
momento * v_pesos[n][i];
                // Atualiza o peso
                pesos[n][i] = pesos[n][i] + v_pesos[n][i];
            }

            // Atualizacao do bias
            v_pesos[n][i] = taxaAprendizagem * gradiente_j[n] * 1 + momento *
v_pesos[n][i];
            pesos[n][i] = pesos[n][i] + v_pesos[n][i];
        }
        c--;
    }
}

public double getMomento() {
    return momento;
}

public void setMomento(double momento) {
    this.momento = momento;
}

public double getTaxaAprendizagem() {
    return taxaAprendizagem;
}

public void setTaxaAprendizagem(double taxaAprendizagem) {
    this.taxaAprendizagem = taxaAprendizagem;
}

@Override
```

```
    public List<Individuo> imigracao(Collection<Individuo> imigrantes) {
        Iterator<Individuo> it = imigrantes.iterator();
        Individuo melhor = individuo;
        while (it.hasNext()){
            Individuo imigrante = it.next();
            if (imigrante.fitness() > melhor.fitness()){
                melhor = imigrante;
            }
        }

        if (! melhor.equals(individuo)){
            rna.atualizarPesos((double[]) melhor.obterFenotipo());
            individuo = new IndividuoPadrao(melhor.obterFenotipo(),
melhor.origem(), melhor.idade(), melhor.fitness());
        }

        return null;
    }

    @Override
    public void inserirIndividuo(Individuo imigrante) {
        rna.atualizarPesos((double[]) imigrante.obterFenotipo());
        individuo = new IndividuoPadrao(imigrante.obterFenotipo(),
imigrante.origem(), imigrante.idade(), imigrante.fitness());
        epoca++;
        estadoCorrente = new Iteracao(-1 * imigrante.fitness(),
System.currentTimeMillis() - inicio, epoca);
        iteracoes.add(estadoCorrente);
        eventoIteracao(estadoCorrente);
    }

    @Override
    public Individuo melhorIndividuo() {
        return individuo;
    }

    @Override
    public double getIndiceConvergencia() {
        return 0.0;
    }
}
```

APÊNDICE C – Implementação do AGR

```
package rna.mlp.treinamento;

import ag.FuncaoAvaliacao;
import ag.Ilha;
import ag.Individuo;
import ag.SelecaoPais;
import ag.real.IndividuoReal;
import ag.real.OperadorMutacao;
import ag.real.OperadorRecombinacao;
import aplicacao.ModeloIlhas;
import java.util.*;
import rna.mlp.MLP;

public class AGReal extends AprendizagemSupervisionada implements Ilha,
FuncaoAvaliacao {

    private final static int MAX_CONVERGENCIA = 5;
    private final static int AMOSTRA_CONVERGENCIA = 6;

    private IndividuoReal melhorIndividuo;
    private IteracaoAG estadoCorrente;
    private ModeloIlhas agp;
    private List<Individuo> populacao;
    private OperadorMutacao opMutacao;
    private OperadorRecombinacao opRecombinacao;
    private SelecaoPais selecaoPais;
    private String origem;
    private double[][] conjuntoTreinamento;
    private double pCrossover, mediaConvergencia, somaConvergencia;
    private long inicio;
    private int contAmostra, contConvergencia;
    private int geracao, numGenes, tamPopulacao;

    public AGReal(MLP rna, SelecaoPais selecao,
            OperadorRecombinacao opRecombinacao, OperadorMutacao
opMutacao, double pCrossover, int tamPopulacao) {
        super(rna);
        this.iteracoes = new LinkedList<IteracaoAprendizagem>();
        this.selecaoPais = selecao;
        this.opMutacao = opMutacao;
        this.opRecombinacao = opRecombinacao;
        this.pCrossover = pCrossover;
        this.tamPopulacao = tamPopulacao;
```

```
    numGenes = rna.getNumPesos();
    populacao = new ArrayList<Individuo>(tamPopulacao);
}

@Override
public void configuraModelo(ModeloIlhas agp){
    this.agp = agp;
    origem = agp.getOrigem();
    agp.setIteracoes(iteracoes);
}

@Override
public void treinar(double[][] padroes, CondicaoParada condicaoParada) {
    inicio = System.currentTimeMillis();
    conjuntoTreinamento = padroes;
    geracao = 0;
    contConvergencia = 0;
    contAmostra = 0;
    somaConvergencia = 0;
    mediaConvergencia = 0;
    inicializarPopulacao();

    while (condicaoParada.condicaoParada(estadoCorrente)){
        List<Individuo> novaGeracao = agp.migracao();
        if (novaGeracao == null){
            novaGeracao = geracao();
        }
        geracao++;
        avaliarIndividuos(novaGeracao);
        selecionarSobreviventes(novaGeracao);
        eventoIteracao(iteracoes.getLast());
    }

    agp.fimExecusao();
}

@Override
public double getErroAtual() {
    return iteracoes.getLast().getEQM();
}

@Override
public int getEpoca() {
   return geracao;
}
```

```
    @Override
    public Individuo melhorIndividuo() {
        return melhorIndividuo;
    }

    @Override
    public void inserirIndividuo(Individuo imigrante) {
        double[] pesos = (double[]) imigrante.obterFenotipo();
        melhorIndividuo = new IndividuoReal(rna, pesos, imigrante.origem(),
imigrante.idade(), imigrante.fitness());
        geracao++;
        estadoCorrente = new IteracaoAG(-1 * imigrante.fitness() , 0,
System.currentTimeMillis() - inicio, geracao);
        iteracoes.add(estadoCorrente);
        eventoIteracao(estadoCorrente);
    }

    @Override
    public double avaliarIndividuo(Individuo individuo) {
        return avaliarIndividuo((IndividuoReal) individuo);
    }

    public double avaliarIndividuo(IndividuoReal individuo) {
        rna.atualizarPesos(individuo.obterFenotipo());
        double fitness = -calcularEQMESoma(conjuntoTreinamento);
        individuo.atualizarFitness(fitness);
        individuo.atualizarSomas(rna.getSomas());
        return fitness;
    }

    private void inicializarPopulacao() {
        IndividuoReal individuo = new IndividuoReal(rna, rna.getPesos(),
agp.getOrigem(), 0, 0);
        double eqm = -avaliarIndividuo(individuo);
        double soma = eqm;
        populacao.add(individuo);
        melhorIndividuo = individuo;

        for (int i=1; i<tamPopulacao; i++){
            individuo = new IndividuoReal(rna, numGenes, agp.getOrigem(), 0);
            avaliarIndividuo(individuo);
            populacao.add(individuo);
            soma += -individuo.fitness();
            if (individuo.fitness() > melhorIndividuo.fitness()){
                melhorIndividuo = individuo;
            }
```

```
        }

        estadoCorrente = new IteracaoAG(-1 * melhorIndividuo.fitness(), soma /
tamPopulacao, 0, 0);
        iteracoes.add(estadoCorrente);
        selecaoPais.setPopulacao(populacao);
    }

    private ArrayList<Individuo> geracao(){
        ArrayList<Individuo> novaGeracao = new
ArrayList<Individuo>(tamPopulacao);
        int i=0;

        if (contConvergencia > MAX_CONVERGENCIA){
            novaGeracao.add(gerarNovoIndividuo((IndividuoReal)
selecaoPais.selecionar()));
            i++;
        }

        novaGeracao.add(melhorIndividuo);
        i++;

        int numFilhos = tamPopulacao - i;
        for (i=0; i<numFilhos; i++){
            IndividuoReal novoIndividuo;
            if (Math.random() < pCrossover){
                novoIndividuo = opRecombinacao.recombinar((IndividuoReal)
selecaoPais.selecionar(), (IndividuoReal) selecaoPais.selecionar(), origem);
            } else {
                novoIndividuo = opMutacao.mutacao((IndividuoReal)
selecaoPais.selecionar(), origem);
            }
            novaGeracao.add(novoIndividuo);
        }

        return novaGeracao;
    }

    private void avaliarIndividuos(List populacao) {
        Iterator<IndividuoReal> it = populacao.iterator();
        IndividuoReal individuo;
        double temp;

        double eqm = -melhorIndividuo.fitness();
        double soma = 0;
```

```
        int numIndividuos = populacao.size();
        for (int i=0; i<numIndividuos; i++){
            individuo = it.next();
            temp = avaliarIndividuo(individuo);
            soma += -temp;

            if (temp > melhorIndividuo.fitness()){
                melhorIndividuo = individuo;
                eqm = -temp;
            }
        }

        melhorIndividuo.atualizarIdade(melhorIndividuo.idade()+1);

        if (estadoCorrente.getEQM() == eqm){
            contConvergencia++;
        } else {
            contConvergencia = 0;
        }

        contAmostra++;
        somaConvergencia += contConvergencia;
        if (contAmostra == AMOSTRA_CONVERGENCIA){
            mediaConvergencia = (mediaConvergencia + somaConvergencia) /
(AMOSTRA_CONVERGENCIA + 1);
            somaConvergencia = 0;
            contAmostra = 0;
        }

        estadoCorrente = new IteracaoAG(-1 * melhorIndividuo.fitness(), soma /
numIndividuos, System.currentTimeMillis() - inicio, geracao);
        iteracoes.add(estadoCorrente);
    }

    private void selecionarSobreviventes(List<Individuo> novaGeracao) {
        populacao = novaGeracao;
        selecaoPais.setPopulacao(populacao);
    }

    private IndividuoReal gerarNovoIndividuo(IndividuoReal modelo){
        double[] cromossomo = modelo.obterGenotipo();

        int tc = cromossomo.length;
        double[] novoCromossomo = new double[tc];
        System.arraycopy(cromossomo, 0, novoCromossomo, 0, tc);
```

```
        int porcao = (int) Math.ceil(0.2 * tc);
        int p1 = (int) (Math.random() * tc);
        int p2 = p1 + porcao;

        double inf = IndividuoReal.getlInf();
        double sup = IndividuoReal.getlSup();

        if (p2 > tc){
            int p3 = p2 - tc;
            for (int i=0; i<p3; i++){
                novoCromossomo[i] += Math.random()*3 - 1.5 -
novoCromossomo[i]/20;
                if (novoCromossomo[i] > sup){
                    novoCromossomo[i] = sup;
                } else if (novoCromossomo[i] < inf){
                    novoCromossomo[i] = inf;
                }
            }
            p2 = tc;
        }
        for (int i=p1; i<p2; i++){
            novoCromossomo[i] += Math.random()*3 - 1.5 - novoCromossomo[i]/20;
            if (novoCromossomo[i] > sup){
                novoCromossomo[i] = sup;
            } else if (novoCromossomo[i] < inf){
                novoCromossomo[i] = inf;
            }
        }

        return new IndividuoReal(rna, novoCromossomo, origem, 0);
    }

    @Override
    public List<Individuo> imigracao(Collection<Individuo> imigrantes) {
        int numImigrantes = imigrantes.size();
        int tamGeracao = tamPopulacao > numImigrantes ? tamPopulacao :
numImigrantes;

        ArrayList<Individuo> novaGeracao = new
ArrayList<Individuo>(tamGeracao);
        IndividuoReal novoIndividuo, aux;
        int i=0;

        for (Individuo imigrante : imigrantes){
            aux = new IndividuoReal(rna, (double[]) imigrante.obterFenotipo(),
imigrante.origem(), imigrante.idade(), imigrante.fitness());
```

```
            avaliarIndividuo(aux);
            if (aux.fitness() > melhorIndividuo.fitness()){
                melhorIndividuo = aux;
            } else {
                novaGeracao.add(aux);
                i++;
            }

            novoIndividuo = opRecombinacao.recombinar(aux, (IndividuoReal)
selecaoPais.selecionar(), origem);
            novaGeracao.add(novoIndividuo);
            i++;
        }

        for (; i<tamGeracao; i++){
            if (Math.random() < pCrossover){
                novoIndividuo = opRecombinacao.recombinar((IndividuoReal)
selecaoPais.selecionar(), (IndividuoReal) selecaoPais.selecionar(), origem);
            } else {
                novoIndividuo = opMutacao.mutacao((IndividuoReal)
selecaoPais.selecionar(), origem);
            }
            novaGeracao.add(novoIndividuo);
        }

        return novaGeracao;
    }

    @Override
    public double getIndiceConvergencia(){
        return mediaConvergencia;
    }
}
```

APÊNDICE D – Implementação da classe Aplicação

```
/*
 * Classe principal do projeto, responsavel por inicializar a GUI,
 * o servidor e gerenciar as conexões. O protocolo de comunicacao
 * e implementado nesta classe e na classe ModeloIlhas.
 */
package aplicacao;

import gui.FormPrincipal;
import gui.FormularioRNA;
import java.awt.Color;
import java.io.IOException;
import java.util.*;
import rede.Conexao;
import rede.Servidor;

public final class Aplicacao {

    public static void main(String args[]) {
        Aplicacao aplicacao = new Aplicacao();
        aplicacao.iniciarGUI();
    }

    // Constantes para representar os possiveis estados do servidor
    public static final byte ESTADO_DISPONIVEL = 1;
    public static final byte ESTADO_OCUPADO_USUARIO = 2;
    public static final byte ESTADO_OCUPADO_RNA = 3;
    public static final byte ESTADO_AGUARDANDO = 4;

    private byte estado;
    private String nomePeer;
    private int porta;
    private FormPrincipal gui;
    private FormularioRNA formRNA;
    private Servidor servidor;
    private Map<String, Conexao> conexoes;
    private Map<Byte, String> estadoDescricao;
    private Map<String, Color> estadoCores;
    private ModeloIlhas agp;

    public Aplicacao(){
        conexoes = Collections.synchronizedMap(new HashMap<String,
Conexao>());

        estadoDescricao = new HashMap<Byte, String>();
```

```
        estadoDescricao.put(ESTADO_DISPONIVEL, "Disponível");
        estadoDescricao.put(ESTADO_OCUPADO_USUARIO, "Ocupado");
        estadoDescricao.put(ESTADO_OCUPADO_RNA, "Ocupado");
        estadoDescricao.put(ESTADO_AGUARDANDO, "Aguardando");

        estadoCores = new HashMap<String, Color>();
        estadoCores.put(estadoDescricao.get(ESTADO_DISPONIVEL), new
Color(0,150,50));
        estadoCores.put(estadoDescricao.get(ESTADO_OCUPADO_USUARIO),
new Color(200,50,0));
        estadoCores.put(estadoDescricao.get(ESTADO_OCUPADO_RNA), new
Color(200,50,0));
        estadoCores.put(estadoDescricao.get(ESTADO_AGUARDANDO), new
Color(180,150,0));

        porta = 4545;
        nomePeer = System.getProperty("user.name");
        if (nomePeer == null){
            nomePeer = "Ilha";
        } else {
            nomePeer = "Ilha do "+ nomePeer;
        }
    }

    public void iniciarGUI(){
        this.gui = new FormPrincipal(this);
        javax.swing.SwingUtilities.invokeLater(this.gui);
    }

    public void iniciarServidor() throws Exception{
        servidor = new Servidor(this, porta);
        servidor.iniciar();
    }

    public void desconectarServidor(){
        servidor.desconectar();
    }

    public void desconectarCliente(String ip, String porta) {
        Conexao conexao = conexoes.get(ip+":"+porta);
        if (conexao != null){
            conexao.desconectar();
        }
    }

    public void log(String texto) {
```

```
        gui.log(texto);
    }

    public synchronized void adicionarConexao(Conexao conexao){
        this.conexoes.put(conexao.cliente(), conexao);
        gui.adicionarCliente(conexao.getNomePeer(), conexao.getIP(),
conexao.getPorta(), estadoDescricao.get(conexao.getEstado()));
        if (formRNA != null){
            formRNA.atualizarListaPeers(conexao);
        }
        gui.log("> Cliente "+ conexao.cliente() +" conectado!");
    }

    public synchronized void removerConexao(Conexao conexao){
        this.conexoes.remove(conexao.cliente());
        gui.removerCliente(conexao.getIP(), conexao.getPorta());
        if (formRNA != null){
            formRNA.atualizarListaPeers(conexao);
        }
        if (agp != null){
            agp.removerIlha(conexao);
        }
        gui.log("> Cliente "+ conexao.cliente() +" desconectado!");
    }

    public boolean servidorIniciado() {
        return (this.servidor != null && this.servidor.iniciado());
    }

    public void conectar(String ip, int porta) throws IOException {
        Conexao conexao = new Conexao(this, ip, porta);
        conexao.enviar(getMensagemInicial());
        conexao.start();
    }

    public Map<String, Color> getMapeamentoEstadoCores() {
        return estadoCores;
    }

    public boolean existeConexao(String id){
        return conexoes.containsKey(id);
    }

    public String getNomePeer() {
        return nomePeer;
    }
```

```
public void setNomePeer(String nome_peer) {
    this.nomePeer = nome_peer;
}

public int getPorta() {
    return porta;
}

public void setPorta(int porta) {
    this.porta = porta;
}

public FormularioRNA getFormRNA() {
    return formRNA;
}

public void setFormRNA(FormularioRNA formRNA) {
    this.formRNA = formRNA;
}

public Collection<String> getPeersDisponiveis(){
    LinkedList<String> peers = new LinkedList<String>();
    Collection conj = conexoes.values();
    Iterator i = conj.iterator();
    while (i.hasNext()) {
        Conexao conexao = (Conexao) i.next();
        byte e = conexao.getEstado();
        if (e == ESTADO_DISPONIVEL || e == ESTADO_AGUARDANDO){
            peers.add(conexao.cliente());
        }
    }
    return peers;
}

public Conexao getConexao(String peer) {
    return conexoes.get(peer);
}

public Conexao getConexaoFonte(){
    if (formRNA == null){
        return null;
    } else {
        return formRNA.getConexaoFonte();
    }
}
```

```
public void setAGP(ModeloIlhas agp) {
    this.agp = agp;
}

// ================================================
//  Funcoes para implementacao do protocolo de comunicacao
// ================================================

public void mensagemTreinamentoIniciado(Conexao conexao) {
    try {
        conexao.enviar("RE EXECUTAR\n1\n");
    } catch (IOException ex) {
        gui.log("* Falha ao enviar a resposta para "+ conexao.cliente());
    }
}

public void atualizarEstado(byte novoEstado){
    String descricao = estadoDescricao.get(novoEstado);
    gui.atualizarEstado(descricao, estadoCores.get(descricao));
    estado = novoEstado;
    enviarMensagem("ESTADO\n"+ estado +"\n");
}

private void enviarMensagem(String mensagem){
    Collection conj = conexoes.values();
    Iterator i = conj.iterator();
    while (i.hasNext()) {
        Conexao conexao = (Conexao) i.next();
        try {
            conexao.enviar(mensagem);
        } catch (IOException ex) {
            gui.log("* Falha ao enviar uma mensagem para "+ conexao.cliente());
        }
    }
}

public void atualizarNome(String nome) {
    this.nomePeer = nome;
    enviarMensagem("NOME\n"+ nome +"\n");
}

public void adicionarIlha(Conexao conexao) throws IOException {
    if (formRNA == null){
        return;
    }
```

```
      String mensagem = "CRNA\n";

      mensagem = mensagem +
formRNA.getConfiguracaoRNA().gerarXML(false, true);
      conexao.enviar(mensagem);
   }

   public void removerIlha(Conexao conexao) throws IOException {
      if (formRNA == null){
         return;
      }
      String mensagem = "DE CRNA\n";
      conexao.enviar(mensagem);
   }

   private String getMensagemInicial(){
      return "RNAED 0.1\n"+ nomePeer +"\n"+ estado +"\n";
   }

   public synchronized void processarMensagemInicial(String msg, Conexao
conexao)
         throws RequisicaoIncorretaException, VersaoIncompativelException,
IOException {
      try {
      String aux = msg.substring(0, 5);
      if (! aux.equals("RNAED")) throw new RequisicaoIncorretaException();

      int p1 = msg.indexOf("\n");
      aux = msg.substring(6, p1);
      if (new Float(aux) <= 0.1) throw new VersaoIncompativelException();

      int p2 = msg.indexOf("\n", p1+1);
      aux = msg.substring(p1+1, p2);
      conexao.setNomePeer(aux);

      p1 = msg.indexOf("\n", p2+1);
      aux = msg.substring(p2+1, p1);
      conexao.setEstado(Byte.parseByte(aux));

      if (! conexao.iniciouConexao()){
         conexao.enviar(getMensagemInicial());
      }

      adicionarConexao(conexao);
      } catch (Exception e){
         e.printStackTrace();
```

```
        }
    }

    public synchronized void processarMensagem(String msg, Conexao conexao)
{
        int p1 = msg.indexOf("\n");
        String comando = msg.substring(0, p1);

        if (comando.equals("NOME")){
            int p2 = msg.indexOf("\n", p1+1);
            String aux = msg.substring(p1 + 1, p2);
            conexao.setNomePeer(aux);
            gui.atualizarNomeCliente(conexao.getIP(), conexao.getPorta(), aux);
            if (formRNA != null){
                formRNA.atualizarNomePeer(conexao.getIP(), conexao.getPorta(),
aux);
            }

        } else if (comando.equals("ESTADO")){
            int p2 = msg.indexOf("\n", p1+1);
            String aux = msg.substring(p1 + 1, p2);
            try {
                byte novoEstado = new Byte(aux);
                conexao.setEstado(novoEstado);
                if (formRNA != null){
                    formRNA.atualizarListaPeers(conexao);
                }
                if (agp != null && (novoEstado == ESTADO_DISPONIVEL ||
novoEstado == ESTADO_OCUPADO_USUARIO)){
                    agp.removerIlha(conexao);
                }

                gui.atualizarEstadoCliente(conexao.getIP(), conexao.getPorta(),
estadoDescricao.get(novoEstado));
            } catch (Exception e){
                gui.log("* Erro ao processar a mensagem "+ aux);
            }

        } else if (comando.equals("CRNA")){
            String aux = msg.substring(p1+1);

            try {
                ConfiguracaoRNA crna = ConfiguracaoRNA.importarXML(aux);
                ConfiguracaoRNA configuracao = null;
                if (formRNA != null){
                    configuracao = formRNA.getConfiguracaoRNA();
```

```
        }

        // Ilha ocupada com um projeto diferente
        if (this.estado == ESTADO_OCUPADO_RNA || this.estado ==
ESTADO_OCUPADO_USUARIO
            || (this.estado == ESTADO_AGUARDANDO && !
crna.getID().equals(configuracao.getID()))){
            String mensagem = "RE CRNA\n0\n";
            conexao.enviar(mensagem);
            return;
        }

        // Ilha ocupada com o mesmo projeto
        if (configuracao != null && crna.getID().equals(configuracao.getID())){
            Conexao fonte = formRNA.getConexaoFonte();
            String mensagem = "RE CRNA\n";

            // Torna o cliente uma ilha subordinada se for a origem do projeto
            // O retorno indica a necessidade de atualzar a ilha de controle
            if (fonte == null){
                conexao.setIlhaSubordinada(true);
                mensagem += "4\n";

            // Fica subordinado ao cliente
            } else if (! formRNA.possuiPeer(fonte)){
                mensagem += "3\n";

            // Apenas confirma a conexao
            } else {
                mensagem += "2\n";
            }
            conexao.enviar(mensagem);

            formRNA.adicionarPeer(conexao);
            if (fonte != null && ! formRNA.possuiPeer(fonte)){
                formRNA.setConexaoFonte(conexao);
            }

        // Ilha disponivel
        } else {
            formRNA = new FormularioRNA(this, crna, conexao);
            new Thread(formRNA).start();
            String mensagem = "RE CRNA\n1\n";
            conexao.enviar(mensagem);
        }
```

```
    } catch (Exception e){
        e.printStackTrace();
        gui.log("* Erro ao processar a mensagem CRNA");
    }

} else if (comando.equals("RE CRNA")){
    if (formRNA == null) return;

    int p2 = msg.indexOf("\n", p1+1);
    String aux = msg.substring(p1 + 1, p2);
    // Retorno de uma ilha ocupada
    if (aux.equals("0")){
        formRNA.falhaAdicionarPeer(conexao);

    // Retorno de uma ilha disponivel
    } else if (aux.equals("1")){
        conexao.setIlhaSubordinada(true);
        formRNA.adicionarPeer(conexao);

    // Retorno de uma conexao confirmada
    } else if (aux.equals("2")){
        formRNA.adicionarPeer(conexao);

    // Retorno de uma ilha subordinada
    } else if (aux.equals("3")){
        conexao.setIlhaSubordinada(true);
        formRNA.adicionarPeer(conexao);

    // Retorno de uma ilha controle
    } else if (aux.equals("4")){
        formRNA.adicionarPeer(conexao);
        Conexao fonteAnterior = formRNA.getConexaoFonte();
        formRNA.setConexaoFonte(conexao);

        String mensagem = "RE CRNA\n5\n";
        try {
            fonteAnterior.enviar(mensagem);
        } catch (IOException ex) {
            gui.log("* Erro ao processar atualizar a ilha controle");
        }

    // Atualizacao da ilha controle
    } else if (aux.equals("5")){
        conexao.setIlhaSubordinada(false);
        formRNA.atualizarListaPeers(conexao);
```

```
        } else {
            gui.log("* Erro ao processar a mensagem RE CRNA");
        }

    } else if (comando.equals("DE CRNA")){
        if (formRNA == null) return;
        conexao.setIlhaSubordinada(false);
        formRNA.desconectarPeer(conexao);

    } else if (comando.equals("EXECUTAR")){
        String aux = msg.substring(p1+1);

        try {
            ConfiguracaoRNA crna = ConfiguracaoRNA.importarXML(aux);
            ConfiguracaoRNA configuracao = null;
            if (formRNA != null){
                configuracao = formRNA.getConfiguracaoRNA();
            }

            if (this.estado == ESTADO_AGUARDANDO &&
crna.getID().equals(configuracao.getID())){
                formRNA.executarTreinamento(conexao, crna);
            }

        } catch (Exception e){
            e.printStackTrace();
            gui.log("* Erro ao processar a mensagem CRNA");
        }

    } else if (comando.equals("RE EXECUTAR")){
        if (agp != null){
            agp.adicionarIlha(conexao);
        }

    } else if (comando.equals("INDIVIDUO")){
        if (agp == null){
            gui.log("* Individuo ignorado");
            return;
        }

        String aux = msg.substring(p1+1);
        agp.receberIndividuo(conexao, aux);

    } else if (comando.equals("FIM ILHA")){
        if (agp == null) return;
```

```
            try {
                int p2 = msg.indexOf("\n", p1+1);
                String aux = msg.substring(p1 + 1, p2);
                agp.fimExecusao(conexao, Integer.parseInt(aux));

            } catch (Exception e){
                gui.log("* Erro ao processar a mensagem FIM ILHA");
            }

        } else if (comando.equals("FIM REDE")){
            if (agp == null) return;

            try {
                agp.fimRede(conexao);

            } catch (Exception e){
                gui.log("* Erro ao processar a mensagem FIM REDE");
            }
        }
    }
}
```