NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

RICARDO JOÃO DUARTE PINHEIRO

Licenciado em Engenharia Informática

ANÁLISE SEMI-AUTOMÁTICA DA PARAMETRIZAÇÃO DO MODELO DE MAPAS AUTO-ORGANIZADOS UBÍQUOS COM RECURSO A PARALELISMO

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Dezembro, 2021

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

ANÁLISE SEMI-AUTOMÁTICA DA PARAMETRIZAÇÃO DO MODELO DE MAPAS AUTO-ORGANIZADOS UBÍQUOS COM RECURSO A PARALELISMO

RICARDO JOÃO DUARTE PINHEIRO

Licenciado em Engenharia Informática

Orientador: Nuno Miguel Cavalheiro Marques
Professor Auxiliar, Universidade Nova de Lisboa

Coorientador: Hervé Miguel Cordeiro Paulino
Professor Associado, Universidade Nova de Lisboa

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Dezembro, 2021

Análise Semi-Automática da Parametrização do Modelo de Mapas Auto-Organizados Ubíquos com Recurso a Paralelismo

Este documento foi gerado com o processador (pdf/Xe/Lua)LaTeXe o modelo NOVAthesis (v6.7.4) [44].

Agradecimentos

Um agradecimento aos meus orientadores Professor Nuno Miguel Cavalheiro Marques e Professor Hervé Miguel Cordeiro Paulino por passagem de conhecimento, motivação e apoio ao longo da concepção deste projecto. Aos meus colegas por terem sido uma força extra na longa batalha que é a conclusão de um curso superior, assim como discussão de ideias e fornecimento de estratégias para melhor encarar este projecto. Um especial agradecimento aos meus Pais e restante família por todo o apoio e carinho fornecido ao longo desta jornada. E a ti Joana, por teres sido principal peça em termos de apoio emocional, tendo sido paciente e solidária nas horas mais complicadas.

Resumo

O objetivo desta dissertação é validar o Mapa Auto-Organizado (SOM) para treino paralelo em Unidades de Processamento Central (CPU) e em Unidades de Processamento Gráfico (GPU) de forma a maximizar o uso de recursos para tentar encontrar as melhores parametrizações para vários contextos, validando os modelos SOM e tentando ilustrar a sua aplicação para solucionar problemas que beneficiam desta análise dinâmica e em *stream*. Outro dos objectivos é aliar o protocolo *Multi-Armed Bandit* à execução de múltiplos modelos em paralelo de forma a encontrar as *features* que são mais relevantes para minimizar as métricas de erro associadas ao modelo UbiSOM, encontrando as *features* mais relevantes de um certo conjunto de dados.

O SOM é uma técnica que projecta conjuntos de dados multi-dimensionais para mapas bidimensionais. Isto permite a visualização dos dados e facilita a interpretação humana, sendo possível descobrir padrões desconhecidos. Contudo, o SOM tem dificuldade a reagir a mudanças repentinas na distribuição dos dados, devido às parametrizações do algoritmo, que necessitam de ser definidas antes da execução. O Mapa Auto-Organizado Ubíquo (UbiSOM) resolve esse problema introduzindo métricas globais, que permitem a adaptação do SOM a *streams*. Refazendo a arquitetura do sistema onde treinos do modelo SOM são realizados, pode-se igualmente tirar partido de uma execução de vários treinos em paralelo, acelerando a sua execução, permitindo o processamento de mapas de grande dimensão e acelerando a pesquisa das melhores parametrizações.

Para validar a implementação e os resultados, serão feitas comparações da análise de sensibilidade de parâmetros de trabalhos anteriores no campo do modelo UbiSOM, assim como será analisada a inclusão do protocolo *Multi-Armed Bandit* na pesquisa das melhores *features* de um certo conjunto de dados. A performance da arquitetura paralela para o modelo UbiSOM em execução no CPU será igualmente analisada.

Palavras-chave: Mapas Auto-Organizados, Mapas Auto-Organizados Ubíquos, Computação de Alto Desempenho, Protocolo *Multi-Armed Bandit*, Unidades de Processamento Central, Unidades de Processamento Gráfico

Abstract

The main goal of this dissertion is to validate the Self-Organizing Map to parallel training in Central Processing Units (CPU) and Graphic Processing Units (GPU), in order to maximize the resources use and to find the best parameters regarding many real-life contexts and this way, validating the SOM models to be used in solving problems that could benefit of this dynamic analysis and in streaming mode. Other of the goals is combine the Multi-Armed Bandit Protocol to the execution of multiple models in parallel, in a way that the most relevant features from a certain dataset are found, the ones which minimize the error metrics associated with UbiSOM.

SOM is a technique that reduces multi-dimensional data sets to bi-dimensional maps. This allows the visualization of the data and helps the human perception about it, being possible to discover unknown patterns. Although, the SOM model has difficulties in adapting to sudden changes in the distribution of the data since the parameters of the algorithm needs to be defined prior to the execution. The Ubiquitous Self-Organizing Map (UbiSOM) solves that problem using global metrics, which allows the SOM's adaptation to streams of data. Reworking the system architecture where the SOM traning is made, is also feasible to take advantage of the execution of many parallel trainings, fastening their execution, allowing the processing of maps of large scale and fastening the search for the best parameters.

To validate the implementation and the results, will be done comparsions of the parameters sensibility analysis of past works on the matter, as well will be analyzed the inclusion of the Multi-Armed Bandit to search the best features of a certain dataset. The performance of the parallel architecture regarding the UbiSOM model in execution on CPU will be likewise analyzed.

Keywords: Self-Organizing Maps, Ubiquitous Self-Organizing Maps, High-Performance Computing, Multi-Armed Bandit Protocol, Central Processing Units, Graphic Processing Units

Índice

Índice de Figuras

Índice de Tabelas

Índice de Listagens

Glossário

MapReduce Técnica ou modelo de programação para processar e gerar grandes conjuntos de dados, num ambiente paralelo ou distribuído. É constituído pela fase de mapeamento onde é aplicada uma função a um conjunto de dados, como filtrar ou ordenar, e pela fase de redução, para sumarizar o resultado da operação do mapeamento. i, 24

nesting Fenómeno onde a informação está organizada em camadas ou onde objectos contêm outros objectos similares. Em questão funcional, pode ser vários níveis de subrotinas ou chamadas recursivas. i, 23

Symmetric Multi-Processor Inclusão de dois ou mais processadores idênticos partilhando uma memória principal. Os múltiplos processadores podem ser *chips* separados ou podem ser vários núcleos no mesmo *chip*. i, 25

tf.placeholder Objeto *TensorFlow* que contém o conjunto de dados de entrada, para entregar posteriormente ao algoritmo que está a ser treinado no momento. i, 24

Best Matching Unit Protótipo de um SOM que possuí menor distância Euclidiana a um elemento do conjunto de dados. Ou seja, tem maior similaridade com o elemento do *input*. i

Java Virtual Machine A *Java Virtual Machine* é uma máquina virtual que permite a um computador executar programas Java assim como outros programas escritos noutras linguagens que são compiladas para Java *bytecode*. i, 113

kernel Função implementada em OpenCL que permite ser compilada para execução num dispositivo OpenCL. i, 19, 21, 22

Multi-Armed Bandit Protocol Problema clássico probabilístico que visa resolver o dilema *exploration vs. exploitation*. i, x, xi, 2, 38, 42, 43, 75, 76, 114

protótipo Unidade pertencente a um Mapa Auto-Organizado, que faz parte do mapa bidimensional resultante. i, 7

SIMD Unidades *Single Instruction, Multiple Data* são componentes de *hardware* que executam a mesma operação em múltiplos elementos de dados concorrentemente. i, 20, 22

SPMD Unidades *Single Program, Multiple Data* são componentes de *hardware* que executam o mesmo programa em múltiplos elementos de dados de forma independente. i, 20

Tensor Objeto principal de um programa *TensorFlow*, cuja manipulação e distribuição é essencial. Representa uma matriz (vetor retangular) de dimensão arbitrária, preenchido com dados de um tipo de dados específico. i, 24

texture binding Método utilizado em Computação Gráfica que permite mapear dados que são utilizados frequentemente pelo GPU, criando assim uma *cache*, de forma a evitar cálculos redundantes. i, 25

Unified Distance Matrix Matrix constituída pelas distâncias entre protótipos adjacentes. i, x, xi, 42, 43, 44, 47, 53, 58, 67, 68, 69, 70, 71, 72, 73, 75, 76, 103, 104

Siglas

AED Análise Exploratória de Dados i

BMU *Best Matching Unit* i, xv, 1, 9, 44, 53, 54, 56, 58, 113, 157

GPGPU General-Purpose Graphical Processing Unit i
GPU *Graphical Processing Unit* i
GUI *Graphical User Interface* i, 35

SOM *Self-Organizing Maps* i

List of Algorithms

1

Introdução

1.1 Motivação

Os Mapas Auto-Organizados (SOM) são uma técnica que visa reduzir um conjunto de dados multidimensional para um mapa bidimensional, que preserva a topologia dos dados. Com a apresentação visual dos dados proveniente deste mapa, os dados podem ser facilmente interpretados por humanos onde podem ser analisados vários padrões e *clusters*, descobrindo padrões até então desconhecidos.

No entanto, o SOM não é um algoritmo que se adapta facilmente a mudanças na distribuição dos dados, graças às parametrizações do algoritmo que necessitam de ser definidas anteriormente à execução. Recentemente, surgiu o UbiSOM que resolve essas limitações, utilizando métricas globais para adaptar o SOM a *streams*. Ainda assim, as parametrizações do modelo UbiSOM necessitam de uma análise de sensibilidade prévia de forma que sejam as mais correctas para gerar os melhores resultados.

A motivação desta dissertação é encontrar as melhores parametrizações dos Mapas Auto-Organizados Ubíquos (UbiSOM), utilizando portanto uma execução paralela de vários modelos. Dado que as parametrizações deste modelo são atribuídas anteriormente à execução dos modelos, estas poderão não se adaptar da melhor forma à distribuição subjacente ao conjunto de dados que esteja a ser treinado em determinado momento. Para tal, se se treinar múltiplos modelos em paralelo, a pesquisa pelas melhores parametrizações poderão ser feitas de forma mais célere. Desta forma, é possível validar o UbiSOM para ser utilizado em contextos reais, efectivamente solucionando problemas onde parece existir vantagem em utilizar esta análise dinâmica.

Aliando a essa pesquisa pelas melhores parametrizações um protocolo de pesquisa das melhores *features* de um certo conjunto de dados (*Multi-Armed Bandit Protocol*), podem ser determinadas as *features* que minimizam as métricas de erro do UbiSOM, sendo assim possível diminuir a complexidade do processo de análise em certos conjuntos de dados onde existam um número elevado de *features*.

Os modelos UbiSOM contém oportunidades de paralelização para elevar a sua performance, como é o caso do cálculo da distância Euclidiana e selecção da BMU (e.g. utilizando

técnicas de *MapReduce*). Contudo, é possível reorganizar a arquitetura do sistema onde os treinos do modelo UbiSOM são realizados, em especial, adoptar uma arquitetura onde a computação paralela possa ser feita e assim, aumentar o número de treinos que possam ser feitos simultaneamente, apenas utilizado uma (ou várias) unidade(s) de computação. Para além do número de treinos que se podem fazer em simultâneo, o processamento de mapas de grande dimensão pode ser igualmente acelerado.

O protocolo *Multi-Armed Bandit* é um problema clássico de probabilidades, onde se demonstra o dilema da *exploration versus exploitation* [79]. Este protocolo é descrito com um exemplo relatando a experiência de que num casino, existindo múltiplas *slot machines* e estando associadas probabilidades de recompensa desconhecida a cada uma delas, quais deverão ser escolhidas para ter um valor máximo de recompensa. Esse cenário pode ser replicado num treino do modelo UbiSOM, onde cada *slot machine* é uma *feature* do conjunto de dados a ser treinado no modelo, e após ser aplicado este protocolo é essencialmente revelado quais as *features* que têm mais importância em produzir os melhores mapas resultantes.

Desta forma, em conjuntos de dados que possuem muitas *features* que estão interdependentes, pode-se tirar conclusões minuciosas, retirando assim algumas *features* da análise geral e diminuindo a carga dessa mesma análise. Este mecanismo pode trazer uma análise mais *fine-grained*, sem ter que lidar com muitas *features* em simultâneo e assim amplificando a percepção humana em vários conceitos, como é o caso da análise financeira [18, 36, 11].

1.2 Objectivos e Contribuições

O objectivo principal desta dissertação é validar o modelo UbiSOM para treino paralelo em CPU e GPU com múltiplas parametrizações, de forma a tentar encontrar as parametrizações ótimas. Esse estudo será realizado tendo em conta resultados de análise de sensibilidade obtidas em [72], de forma a refinar essas parametrizações e encontrar conjuntos de parametrizações que otimizem os resultados dos mapas do modelo UbiSOM para vários conjuntos de dados. A validação é feita com base numa implementação de uma aplicação chamada MultiSOM (3.2) que faz o treino do modelo UbiSOM, para além de dinamizar essa aplicação para suportar o treino paralelo de várias instâncias com múltiplas parametrizações. Outro dos objectivos é estudar possíveis adaptações dinâmicas do UbiSOM com um mecanismo simples de selecção de *features*, particularmente utilizando o protocolo *Multi-Armed Bandit*. Essencialmente, pretende-se analisar se este mecanismo tem impacto na performance geral dos treinos do modelo UbiSOM (comparando o tempo de execução de treinos deste modelo sem o mecanismo Multi-Armed Bandit Protocol activo) e se efectivamente a versão implementada para esta dissertação selecciona as melhores *features* num dado conjunto de dados (comparando o treino com *dataset* inicial ao treino com o *dataset* só com as *features* encontradas pelo mecanismo de selecção de *features*).

Para além disso, propõe-se ainda um melhoramento no *debug* dos algoritmos, especialmente na visualização das métricas e parâmetros de treino, obtendo assim um maior controlo na análise de sensibilidade das execuções.

Em suma, os **objectivos** e as **contribuições** do projeto são as seguintes:

1. Estudar o UbiSOM para paralelização de treinos com múltiplas parametrizações
2. Realizar uma análise de sensibilidade aos parâmetros do SOM, tentando encontrar as parametrizações ótimas
3. Estudar a paralelização do UbiSOM num contexto de selecção de *features*, particularmente utilizando o protocolo *Multi-Armed Bandit*
4. Melhorar as ferramentas de monitorização e o *debugging* da implementação para auxiliar a validação das parametrizações de treino dos algoritmos com várias parametrizações em paralelo

1.3 Organização do Documento

Este capítulo é dedicado à motivação, objetivos e contribuições desta dissertação.

No segundo capítulo é efetuada uma revisão de trabalhos relacionados e estado da arte. Na secção 2.1 são analisados métodos para a Análise Exploratória de Dados (AED), que contém a descrição, algoritmo e aplicações do K-Médias, SOM e UbiSOM. A secção 2.2 contém a descrição do GPU e a programação em GPU (GPGPU), referindo os modelos de programação que podem ser utilizados para implementar aplicações de âmbito geral no GPU, como o OpenCL, CUDA e Marrow. Na secção 2.3 é feita uma análise de implementações de algoritmos AED em GPU, tanto para o SOM, como para o UbiSOM. Na secção 2.4 estão apresentados alguns exemplos recentes que motivaram o estudo desta dissertação para aplicação do SOM em contexto financeiro. A secção 2.5 contém a explicação do protocolo *Multi-Armed Bandit*, incluindo a descrição dos vários algoritmos possíveis de usar. Na última secção (2.6) estão exemplificados algumas aplicações que podem ser utilizadas para desenvolver visualizações interactivas, especialmente aplicado ao contexto de Aprendizagem Automática.

No terceiro capítulo (3) está descrito a arquitetura relativa à aplicação utilizada para cumprir os objectivos desta dissertação. Nomeadamente, a arquitetura geral do sistema (3.1), a arquitetura relativa ao MultiSOM (3.2) e a arquitetura relativa à infraestrutura do sistema que treina o modelo UbiSOM em GPU (3.3).

No quarto capítulo (4) é dedicado aos vários detalhes de implementação das várias funcionalidades do sistema: instanciação dos vários treinos em simultâneo (4.1), múltiplas parametrizações (4.2), treino do modelo UbiSOM sem visualização interactiva (4.3), selecção de *features* com o protocolo *Multi-Armed Bandit* (4.4) e ferramentas de *debugging* e monitorização (4.5).

No quinto capítulo (5) estão presentes os testes, resultados e análise. No sexto capítulo (6) estão presentes as conclusões e trabalho futuro associadas a esta dissertação.

2

Revisão de Literatura

2.1 Análise Exploratória de Dados

A Análise Exploratória de Dados (AED) é um método proposto por John Tukey [77] que consiste em analisar conjuntos de dados de forma a explicitar as suas características com recurso a ferramentas de visualização. Este método estatístico engloba os procedimentos para analisar dados, técnicas para interpretar os resultados dos procedimentos anteriores, formas de planeamento estrutural para simplificação da recolha de dados e todas as ferramentas e resultados estatísticos associados à análise de dados.

O objectivo principal da AED é maximizar o conhecimento que um analista tem sobre um conjunto de dados relativamente à sua arquitectura e composição, garantindo que é retornado ao analista ilações que este queira retirar dos dados, como: estimação de parâmetros, incerteza da estimação, uma lista ordenada de factores importantes, concluir se um determinado factor é relevante e configurações optimizadas [71]. Essencialmente, é um processo de realizar investigações primordiais em dados de forma a descobrir padrões, encontrar anomalias, testar hipóteses e verificar pressupostos com o auxílio de estatística e representações gráficas, i.e. retirar conhecimento de um conjunto de dados [63].

Nesta secção, serão enunciadas as técnicas e os modelos que podem ser estudados e facilitar a realização deste projecto, nomeadamente utilizando Mapa Auto-Organizado Ubíquo. Para tal, a secção irá igualmente conter a especificação e caracterização dos modelos de *data mining* K-Médias e dos Mapas Auto-Organizados de forma a introduzir correctamente o Mapa Auto-Organizado Ubíquo, dado que são modelo que servem como base do anteriormente referido.

2.1.1 K-Médias

O agrupamento K-Médias é um método de quantização de vetores cujo objectivo é criar partições de n observações em k grupos (ou *clusters*) de forma a que cada observação pertença ao grupo com o valor médio mais próximo, servindo a anterior como protótipo do agrupamento [27]. O algoritmo K-Médias é um algoritmo relativamente simples de entender e implementar, tornando-o um algoritmo popular para *clustering* [26].

2.1.1.1 Algoritmo K-Médias

O algoritmo utilizado no método K-Médias é um algoritmo iterativo que é constituído por dois passos: actualização e atribuição [32].

Inicialização: Os dois métodos mais comuns para a inicialização do algoritmo são o método de Forgy e o método da partição aleatória. O método de Forgy consiste em escolher aleatoriamente k pontos do conjunto de dados inicial e utiliza-os como valores médios iniciais. O método da partição aleatória atribuí os *clusters* de forma aleatória a cada elemento do conjunto de dados inicial, executando de seguida a fase de actualização, desta forma computando o protótipo de cada um dos *clusters* [26]. O objectivo é o algoritmo começar com estimativas iniciais para os k *clusters*, sejam geradas aleatoriamente ou escolhidas no conjunto de dados inicial [76].

Fase da Atribuição: Dado que cada protótipo define cada um dos k grupos, nesta fase, todos os elementos do conjunto de dados inicial é agregado ao seu agrupamento, ou seja, de acordo com a menor distância euclidiana, aproximando as observações do seu protótipo mais próximo [26].

$$\underset{c_k \in k}{argmin}\ dist(c_k, x)^2 \tag{2.1}$$

Na equação 2.1, $dist$ é uma função que calcula a distância Euclidiana, sendo c_k a colecção de protótipos no conjunto C e x as observações iniciais [76].

Fase da Actualização: Esta fase consiste em actualizar os protótipos, que consiste em calcular o novo valor médio para o novo conjunto de observações. Formalmente, a fase de actualização está assim definida [76]:

$$c_i = \frac{1}{|S_i|} \sum_{x_i \in S_i} x_i \tag{2.2}$$

Nesta fase, cada *cluster* calcula um novo protótipo com base nas observações feitas na fase da atribuição, iterando de seguida cada observação para calcular a contribuição para a média.

A **convergência** do K-Médias ocorre quando não existe mais nenhuma mudança a ser feita nos protótipos nem nos agrupamentos [27]. A função da afiliação é a representação do nível de afiliação que um certo ponto x se relaciona com um protótipo c. Diz-se que a afiliação é forte quando o valor da função é 0 ou 1 e fraca, caso assuma qualquer um dos valores entre 0 e 1. O algoritmo K-Médias se utilizar uma afiliação forte, poderá comprometer a qualidade da solução resultante. No entanto, o algoritmo pode incluir optimizações que não afectam a eficácia do algoritmo, como é o exemplo das árvores KD [1] [26].

[1] Estrutura de dados que organiza pontos num espaço com k dimensões.

Quanto a implementações do **K-Médias em GPU**, existem vários relatórios que comprovam que o K-Médias pode ser efectivamente paralelizado e executado em GPUs, ganhos de *speedup* de por exemplo entre 2.3-600x numa implementação e 30x em outra, sendo comparando com a execução sequencial do algoritmo em CPU, conforme descrito em [43, 29, 74, 8]. Li et. Al em [43] descrevem que o desempenho do algoritmo é sensível ao tamanho do conjunto de dados inicial, inclusive eles especificam que utilizam memórias do GPU diferentes consoante o tamanho do *data set*. No geral, os autores concordam com o facto da fase de actualização ser mais pesada para o CPU, executando essa fase no GPU, enquanto a fase de atribuição é executada no CPU [74, 8].

2.1.2 Mapa Auto-Organizado

Os Mapas Auto-Organizados (proveniente do inglês *Self-Organized Maps*) são um modelo de Aprendizagem Automática, tipificadas como Rede Neuronal Artificial cujo objectivo é reduzir um conjunto de dados multi-dimensional para um mapa de menores dimensões ou geralmente, para uma matriz bidimensional, constituída por protótipos [18]. Os protótipos que constituem o mapa final são constituídos por *features* que tem a mesma dimensão que o conjunto de dados inicial. Para análise das relações entre os dados, são medidas as distâncias das *features* de qualquer protótipo a qualquer ponto do conjunto de dados e entre qualquer protótipo aos seus vizinhos [42].

Os SOM são treinados através de um algoritmo não supervisionado e através de um processo apelidado de auto-organização, geram uma representação topológica do conjunto de dados inicial. Estes mapas possuem a característica de preservar a topologia do conjunto de dados inicial, que significa que os dados que estão mais relacionados entre eles, vão ser mapeados mais próximos. Sendo assim, o SOM para além de ser uma ferramenta de visualização de conjuntos de dados multi-dimensionais, mostra-se igualmente como um modelo para fazer *clustering* [18].

O modelo SOM envolve **3 processos essenciais** para formar o mapa [28]:

Competição: As unidades (ou protótipos) do mapa computam cada um dos valores de uma função discriminante em relação às unidades do conjunto dos valores de entrada. Essa função discriminante possibilita as unidades do mapa competirem entre si. A unidade do mapa com maior resultado da função discriminante, é declarado o vencedor da competição. Se o conjunto de valores de entrada tiver dimensão D (número de *features*), podemos representar esse conjunto como $x = x_i : i = 1,...,D$, ficando o vector de pesos associados a cada unidade do mapa assim descrito $w_j = w_j i : j = 1,...,N; i = 1,...,D$, onde N é o número total de unidades no mapa.

Se for utilizado o índice $i(x)$ para identificar o protótipo que melhor corresponde à observação do conjunto de dados de entrada x, o $i(x)$ pode ser representado pela seguinte condição, que essencialmente sintetiza o processo de competição entre os

protótipos:

$$i(x) = arg\,min_j \left\| x - w_j \right\|, j \in \alpha \tag{2.3}$$

Cooperação: Em estudos neuro-biológicos, observou-se que existem interacções laterais entre neurónios que estão activos. Quando um neurónio é activado, consequentemente, os seus vizinhos mais próximos têm tendência a ficar activos, invés dos que estão mais afastados [10]. No algoritmo SOM está introduzida uma vizinhança topológica (h_j, i), normalmente centrada na unidade vencedora, que decaí de acordo com a distância lateral. A amplitude da vizinhança topológica decresce monotonamente com o aumento da distância lateral (d_j, i), decaindo para o valor zero quando $d_(j, i)$ tende para o infinito, cujo condicionalismo é necessário para a convergência do mapa [28].

Adaptação: O processo de adaptação consiste no aumento dos valores individuais da função discriminante das unidades do mapa em relação ao conjunto de valores de entrada, através da afinação dos seus vectores de distância. Desta forma, o modelo fica treinado para situações futuras, onde a unidade do mapa vencedora responda com mais eficiência a um conjunto de valores de entrada semelhante [28].

2.1.2.1 Algoritmo Online-SOM

O algoritmo SOM idealizado por Kohonen concentra-se na substituição de computações geométricas simples pelas propriedades mais detalhadas da regra de Hebb [2] e de interacções laterais [28]. Essencialmente, é uma reprodução dos fenómenos biológicos em que o SOM se sustenta [40]. Esta sub-secção irá conter a especificação das várias etapas do algoritmo SOM, contendo uma breve descrição de cada uma das mesmas, culminando com o algoritmo completo. A função de vizinhança, taxa de aprendizagem e a *Best Matching Unit* serão igualmente especificados.

Inicialização: A inicialização do algoritmo SOM é feita escolhendo valores aleatórios para os vectores de distância de cada uma das unidades do mapa $(w_j(t) \wedge t = 0)$. Os valores de $w_j(0)$ são diferentes para $j = 1, 2, ..., l$, sendo l o número de unidades do mapa. Kohonen [41] comprovou que este tipo de inicialização é eficiente, dado que a auto-organização permite a ordenação dos vectores e consequentemente formando a topologia do mapa.

Contudo, existe um método de inicialização que consiste na selecção dos valores para $w_j(t)$ com base no conjunto de dados inicial, escolhendo elementos desse conjunto de forma aleatória. A vantagem deste método em relação ao anterior centra-se no facto do mapa inicial estar na mesma ordem que o mapa final [28].

[2] A regra de Hebb é uma regra de aprendizagem que descreve como as atividades neurais influenciam a comunicação entre neurónios, ou seja, a plasticidade sináptica.

Fase da Amostragem: Esta fase consiste na seleção de elementos do *input* e como o nome indica, servindo para retirar uma amostra do conjunto de dados [28]. Os elementos são escolhidos de forma aleatória, dado que o algoritmo tem a garantia de convergir a partir de uma certa altura no tempo [40].

Fase de *Matching*: Após termos uma amostra do *input*, o objectivo desta fase é encontrar a unidade do mapa "vencedora", ou seja, a unidade do mapa mais semelhante a essa amostra (x). A comparação é feita com base na equação da BMU (2.4), que demonstra que a unidade seleccionada como BMU é aquela que possuí a menor distância euclidiana entre a observação x do *input* e a unidade do mapa m_i, conforme apresentado na equação 2.4 [40]:

$$\|x - w_c\| = \min_j \left\|x - w_j\right\| \tag{2.4}$$

A unidade m_c é a unidade do mapa definida como BMU de uma observação x. Esta equação pesquisa todas as entradas do mapa, de forma a encontrar então a unidade cuja distância euclidiana para a entrada x é menor.

Fase da Actualização: O objectivo da fase de actualização é ajustar os vectores de pesos associadas a cada unidade do mapa, em que depois de ser determinada a BMU, trata de actualizar os pesos que estão na vizinhança do elemento BMU, de forma a guiar o mapa para a ordenação global. É crucial para a ordenação do mapa que as unidades que estão a fazer a fase de treino não sejam afectadas independentemente [40].

$$m_j(t+1) = w_j + \eta(t) * h_{ci} * [x_i - w_j(t)] \tag{2.5}$$

A equação descrita em 2.5 é a função de actualização definitiva [40]. É constituída por $\eta(t)$ sendo a **taxa de aprendizagem** para o instante t e a $h_ci(t)$ é a **função de vizinhança**, também para o instante t. Passe-se então à especificação desses dois constituintes:

1. **Taxa de Aprendizagem:** A Taxa de Aprendizagem (η) é o ritmo que o mapa faz a aprendizagem, que no caso dos SOMs, auxilia os vetores de pesos associados a cada elemento do mapa a evoluir de forma favorável, evitando que exista erros associados a uma evolução brusca, danificando assim os resultados finais [80]. Parte-se do pressuposto que a taxa de aprendizagem decresce de forma monótona à medida que o mapa converge, durante a fase de ordenação [72, 40]. $\eta(t)$ é um escalar que varia entre $0 < \eta(t) < 1$. Existem várias funções que podem ser utilizadas para determinar os valores da taxa de aprendizagem e no caso do SOM, a mais utilizada é a função linear a seguir apresentada:

$$\eta(t) = \eta(0)\frac{1}{t} \tag{2.6}$$

2. **Função de Vizinhança:** A Função de Vizinhança é a função que determina quais as unidades que serão afectadas numa determinada iteração t do treino do mapa. A centralização é feita de acordo com a BMU nesse instante t em relação ao ponto actual do conjunto de entrada de dados. Esta função, como a taxa de aprendizagem definida anteriormente, deverá ser monótona e decrescente [40]. A função de vizinhança baseia-se em aplicar interações laterais, definindo uma vizinhança N_c em volta de uma unidade c (centrada, claro está, na BMU). São apenas afectadas as unidades dentro da vizinhança, sendo ignoradas as restantes. Segundo Kohonen [40], existe vantagens em aplicar uma vizinhança larga no início do treino do mapa e ir reduzindo monotonamente. Com este factor, permite-se que com um valor de taxa de aprendizagem alto, o mapa consiga adquirir uma ordem global. A seguinte equação (2.7) descreve a função de vizinhança aplicada nos SOMs, que neste caso será uma função **Gaussiana** [41]:

$$h_{ci}(t) = \eta(t) \cdot \exp\left(\frac{\|r_c - r_i\|^2}{2\sigma^2(t)}\right) \tag{2.7}$$

Algoritmo: Depois de definir componentes como a taxa de aprendizagem, a função de vizinhança e a função de actualização, podemos elaborar o algoritmo completo. O treino de um SOM é efectuado ao longo de várias iterações, em que cada iteração é processada uma observação. Inicialmente, calcula-se a distância euclidiana entre o ponto de treino i e todas as unidades do mapa, onde se concluí qual é a BMU em dado instante, sendo o ponto de menor distância a i. Por cada ponto observado incrementa-se um índice temporal t, que representa o instante em que iteração atual é efectuada. Por último, aplica-se a função de actualização (equação 2.5).

O algoritmo 1 adaptado de [9], sintetiza a informação previamente descrita.

Algorithm 1 Algoritmo do Mapa Auto-Organizado [9].

```
1:  procedure SOM(inputs)
2:      map ← inicializar com n vetores aleatórios
3:      t ← 0
4:      for all i ∈ inputs do
5:          t ← t + 1
6:          distances ← ∀_k|map_k − i|
7:          bmu ← argmin(distances)
8:          map ← update_map(t, i, bmu, #inputs)
9:      end for
10: end procedure
```

2.1.2.2 Aplicações dos Mapas Auto-Organizados

Os Mapas Auto-Organizados têm aplicabilidade num vasto leque de áreas, em particular aquelas que têm tendência a operar com conjuntos de dados de grande volume [54].

Alguns dos exemplos estão enunciados de seguida:

- **Estudos Climatéricos:** Cavazos [14] realizou um estudo onde utilizava Mapas Auto-Organizados para examinar eventos climáticos associados à precipitação no Inverno na região dos Balcãs. Hewitson e Crane [6] também possuem trabalho onde utilizam SOMs para investigação climatérica, focando-se no estudo dos ciclones e anticiclones, analisando os seus padrões de circulação e desta forma prevendo o seu aparecimento e possíveis rotas.

- **Análise Financeira:** Deboeck em [17] exemplificou várias aplicações dos SOMs para a área financeira, como a selecção de fundos mútuos e a análise de créditos de risco relativamente a cada país. Este último exemplo é um caso onde é feita uma análise de oportunidades de investimento em mercados emergentes. Afolabi et Al. [2] utiliza uma variante de SOMs, SOM Híbrido, para prever os preços das acções nos mercados, auxiliando os investidores a temporizar melhor as decisões relativamente a compra e venda das acções. O conjunto de dados utilizado foi uma seleção de preços diários de ações de companhias aéreas durante 5 anos consecutivos. M. Carrega em [12] utiliza modelos SOM num sistema de apoio à decisão, com recurso a análise fundamental dos indicadores financeiros.

- **Análise Ambiental:** Postolache et Al. [58] desenvolveu um sistema de telemetria utilizando SOMs, para monitorizar a qualidade da água. Este sistema pode detectar eventos como poluição. Mele e Crowley [51] utilizaram os SOMs para monitorizar a qualidade do solo para agricultura, implementando um sistema de apoio à decisão.

- **Análise de Imagem:** Richardson et Al. [68] realizaram trabalho na área de análise de imagem, utilizando os Mapas Auto-Organizados para localizarem padrões numa grande quantidade de imagens de satélite.

2.1.3 Mapa Auto-Organizado Ubíquo

Apesar do SOM ter inúmeras aplicações, conforme mostrado em 2.1.2, simultaneamente possuí as suas limitações. A isto deve-se o facto de conter parâmetros como a função de vizinhança (2.7), taxa de aprendizagem (2.6) e tamanho do mapa que necessitam de definição antes da execução do algoritmo, algo que impede o algoritmo de se adaptar a fontes de dados ilimitadas.

No entanto, existem vários trabalhos de investigação que visam resolver as dificuldades apresentadas pelo SOM, como é o caso do *parameterless*-SOM [7] e do *dynamic*-SOM [30]. Para colmatar as dificuldades do SOM, estas variações sugerem utilizar métricas globais para adaptar os parâmetros de aprendizagem de acordo com a distribuição utilizada no conjunto de dados que vai chegando ao algoritmo. Contudo, essas variações apresentam problemas ao nível da convergência a partir de conjuntos de dados que não estejam ordenados e ao nível do mapeamento da densidade do conjunto de dados que

provem da fonte, que dificulta a utilização de técnicas de visualização normalmente utilizadas no SOM. Nesta secção vai ser especificado uma variação que permite resolver as dificuldades do SOM apresentadas anteriormente, o UbiSOM (Mapa Auto-Organizado Ubíquo ou *Ubiquitous Self-Organizing Map*), com base na tese de Bruno Silva [72] e a tese de João Borrego [9].

2.1.3.1 Parâmetros do modelo UbiSOM

O algoritmo UbiSOM utiliza duas métricas de avaliação global, nomeadamente o erro médio de quantização ($\overline{qe}(t)$) e a utilidade média de um protótipo ($\overline{\lambda}(t)$), calculadas ao longo do tamanho de uma janela deslizante [72].

Ademais são utilizados **parâmetros específicos** ao modelo UbiSOM:

- β: Peso da função de deriva (apresentada na sub-secção seguinte).
- **T**: Tamanho da janela deslizante.
- σ_i, σ_f: Valor inicial e final do raio da vizinhança normalizado, para o estado de ordenação.
- η_i, η_f: Valor inicial e final da taxa de aprendizagem, para o estado de ordenação.

2.1.3.2 Algoritmo UbiSOM

O algoritmo UbiSOM é tem por base métricas de avaliação global adaptadas do Online-SOM: erro médio de quantização e a utilização média de um protótipo, determinados sobre uma janela deslizante (com tamanho T) [72]. O tamanho do mapa é definido previamente à execução do algoritmo. Possuí ainda uma máquina de estados finita que é constituída por dois estados: ordenação e aprendizagem. Outras métricas poderão ser igualmente adaptadas do Online-SOM, como o erro topológico.

Cada protótipo (representado por k) do mapa UbiSOM é um tuplo $W_k =< w_k, t_k^{update} >$, onde w_k é um protótipo e t_k^{update} guarda o *timestamp* da última vez que o protótipo em questão foi actualizado, funcionando como um mecanismo de "envelhecimento"[72, 73].

Erro Médio de Quantização: O erro médio de quantização é uma métrica que exibe a capacidade de um modelo SOM de se adequar à distribuição subjacente à fonte de entrada [9, 72]. Nas situações em que a distribuição da *stream* de dados é estacionária, é esperado que esta métrica diminua e estabilize. Caso contrário, se a distribuição muda é esperado que esta métrica aumente.

Esta métrica é obtida em primeira instância em observações individuais para calcular o erro local de forma a que posteriormente seja obtida uma média de todas as observações. O erro local de quantização é obtido durante a fase de pesquisa da BMU:

$$E'_q(t) = \frac{||x_t - w_c||}{\Omega} \tag{2.8}$$

A equação é constituída pela observação atual (X_t), pelo protótipo seleccionado para BMU (W_c) e $|\Omega|$ é a maior distância entre duas entradas no espaço de entrada normalizado. Os valores obtidos em 2.8 são depois utilizados numa janela deslizante com tamanho T, cujo valor terá de ser estritamente maior que 1, normalmente na casa dos milhares. Com estes valores já calculados, procedemos então ao cálculo do erro médio de quantização:

$$\overline{qe}(t) = \frac{1}{T}\sum_{t}^{t-T+1} E'_q(t) \tag{2.9}$$

Utilidade Média dos Protótipos: Esta métrica tem como propósito exibir a proporção de protótipos que estão a ser activamente actualizados [72]. A motivação desta métrica assenta-se no facto da métrica apresentada anteriormente não detectar o desaparecimento de *clusters* do mapa, causada pela mudança da distribuição dos dados.

A métrica é calculada com recurso ao *timestamp* t_k^{update}, cuja interpretação do seu valor, podemos concluir que um certo protótipo foi a BMU ou que tenha estado na região de influência da BMU, limitado pela função de vizinhança [9]. Inicialmente, $t_k^{update} = 0$.

A Utilidade de um protótipo $\lambda(t)$ é calculada de acordo com a equação 2.10 de onde resulta um rácio de protótipos que foram actualizados nas últimas T observações sobre o número total de protótipos (K) [9, 73]:

$$\lambda(t) = \sum_{k=1}^{K} 1_{t-t_k^{update} \leq T} \tag{2.10}$$

A Utilidade Média dos protótipos ($\overline{\lambda(t)}$) é posteriormente obtida com recurso à média dos valores de λt, conforme representado na equação 2.11. Caso exista um decréscimo de $\overline{\lambda}(t)$, significa que alguns protótipos não estão a ser utilizados, indicando que existem alterações na distribuição adjacente [9, 72].

$$\overline{\lambda}(t) = \frac{1}{T}\sum_{t}^{t-T+1} \lambda(t) \tag{2.11}$$

Função de Deriva: As métricas especificadas anteriormente são utilizadas numa função de deriva que determina o desempenho do mapa sobre uma determinada fonte de dados [9]. Esta função tem como objectivo determinar os parâmetros de aprendizagem do UbiSOM [73].

A equação está representada em 2.12, onde $\beta \in [0,1]$ que é um parâmetro que determina a relação entre as duas métricas [9].

$$d(t) = \beta\overline{qe}(t) + (1-\beta)(1-\overline{\lambda}(t)) \tag{2.12}$$

Até as primeiras T observações serem processadas, o algoritmo UbiSOM não utiliza esta função para determinar os parâmetros de aprendizagem, mas sim parâmetros de aprendizagem monotonamente decrescentes [72].

Função de Vizinhança: Para obter os valores da função de vizinhança, o algoritmo UbiSOM utiliza um raio de vizinhança normalizado $\sigma(t)$ como parâmetro de aprendizagem [9]. A função de vizinhança é truncada, permitindo assim o cálculo de $\overline{\lambda}(t)$ [72].

A normalização de $\sigma(t)$ é baseada na distância máxima entre dois protótipos no mapa. Em mapas rectangulares, os protótipos mais distantes são aqueles que estão nos extremos, ou seja, posições $(0,\,0)$ e $(width-1, height-1)$. Este processo está ilustrado na figura 2.1.

$$\|DIAG\| = \sqrt{(width-1)^2 + (height-1)^2} \tag{2.13}$$

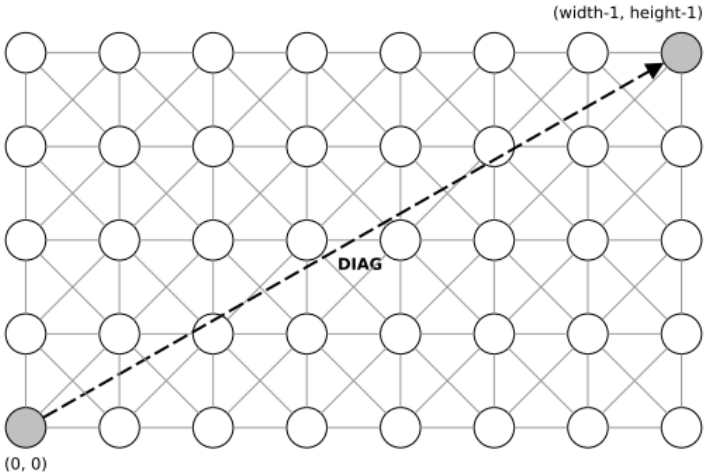


Figura 2.1: $\|DIAG\|$ utilizada para normalização do $\sigma(t)$[72].

Assim sendo, as distâncias dos protótipos do mapa são normalizados pela norma do vector $DIAG$, representado na equação 2.13, limitando assim o raio de vizinhança máximo que o UbiSOM pode utilizar e define $\sigma \in [0,1]$. Desta forma, já podemos definir a função de vizinhança do UbiSOM:

$$h'_{ck}(t) = e^{-\left(\frac{\|r_c - r_k\|}{\sigma(t)\|DIAG\|}\right)^2} \tag{2.14}$$

Comparativamente ao algoritmo Online-SOM, a função de vizinhança só difere no termo de normalização ($DIAG$). Protótipos cujos valores de $h'_{ck}(t)$ estejam abaixo de um limiar de 0.01 não são actualizados, sendo uma etapa crítica para a computação

de $\lambda(t)$ dado que $h'_{ck}(t)$ nunca atinge o valor de 0 e desta forma, todos os protótipos seriam actualizados com valores muito pequenos [72]. Efectivamente, faz com que actualizações negligenciáveis aos protótipos não sejam realizadas.

Máquina de Estados Finitos: No algoritmo UbiSOM está implementada uma máquina de estados finita de 2 estados, conforme representado em 2.2. O algoritmo UbiSOM utiliza uma regra de **actualização** nos 2 estados semelhante ao algoritmo Online SOM, com a excepção do limite mínimo para um protótipo ser actualizado, conforme apresentado em 2.15:

$$W_k(t) = \begin{cases} W_k(t) + \eta(t) h'_{ck}(X_t - W_k(t)) & \text{se } h'_{ck}(t) > 0.01 \\ W_k(t) & \text{c.c.} \end{cases} \tag{2.15}$$

De seguida, serão apresentados os estados constituintes da máquina de estados do algoritmo UbiSOM.

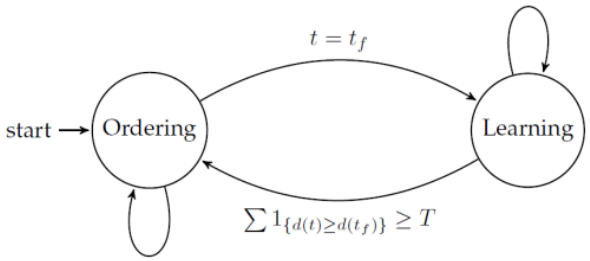


Figura 2.2: Máquina de estados finitos do UbiSOM [72].

– **Estado de Ordenação:** O estado de ordenação permite que o mapa se adapte à distribuição subjacente, onde os parâmetros de aprendizagem são estimados com uma função que decresce monotonamente [9]. Este estado tem a duração de T observações e no final dessas observações, o algoritmo progride para o estado de aprendizagem. Convém salientar que o algoritmo reverte para este estado quando o mesmo não consegue recuperar de uma mudança abrupta na *stream* de dados. Os parâmetros de aprendizagem deverão ser relativamente altos, de forma a que o mapa possa convergir partindo de uma inicialização desordenada dos protótipos [72].

  Formalizando, t_i é a primeira iteração do estado de ordenação e $t_f = t_i + T - 1$ será a iteração final. Para além disso, este estado requer que sejam escolhidos valores adequados para os parâmetros de aprendizagem η_i, η_f, σ_i e σ_f, que representam respectivamente o valor inicial e final para a taxa de aprendizagem e para o raio da vizinhança normalizado. Para obter os valores anteriormente enunciados, utiliza-se uma função exponencial decrescente, formalizada para os parâmetros de aprendizagem em 2.16.

$$\sigma(t) = \sigma_i \left(\frac{\sigma_f}{\sigma_i}\right)^{\frac{t}{t_f}}, \eta(t) = \eta_i \left(\frac{\eta_f}{\eta_i}\right)^{\frac{t}{t_f}} \tag{2.16}$$

Com a conclusão da iteração t_f, o primeiro valor da função de deriva é obtido e o algoritmo UbiSOM transita para o estado de aprendizagem.

– **Estado de Aprendizagem:** O estado de aprendizagem começa em $t_f + 1$ e é o estado principal do algoritmo UbiSOM. Os parâmetros de aprendizagem neste estado são determinados em exclusividade pela função de deriva, aumentando ou diminuindo relativamente ao valor $d(t_f)$ e aos valores finais (η_f, σ_f) determinados no estado de ordenação anterior. Nas equações 2.17 e 2.18 estão representadas as formalizações referentes à determinação dos parâmetros de aprendizagem nesta fase.

$$\eta(t) = \begin{cases} \frac{\eta_f}{d(t_f)} d(t) & \text{se } d(t) < d(t_f) \\ \eta_f & \text{c.c.} \end{cases} \tag{2.17}$$

$$\sigma(t) = \begin{cases} \frac{\sigma_f}{d(t_f)} d(t) & \text{se } d(t) < d(t_f) \\ \sigma_f & \text{c.c.} \end{cases} \tag{2.18}$$

A conclusão que se retira da introdução das duas equações anteriores é que se a distribuição subjacente é estacionária, os parâmetros de aprendizagem acompanham o declínio dos valores da função de deriva, permitindo ao mapa convergir. Caso contrário, se se confirmar que existiram mudanças na distribuição, os valores da função de deriva aumentam e consequentemente, elevam os parâmetros de aprendizagem, aumentando a plasticidade do mapa até ao ponto que $d(t)$ diminua novamente.

Caso a mudança na distribuição dos dados seja abrupta e o mapa não consiga recuperar, o UbiSOM regressa ao estado de ordenação. Na equação 2.19 é feita a avaliação desse acontecimento.

$$se \sum 1_{\{d(t) \leq d(t_f)\}} \leq T \rightarrow ordering\_state \tag{2.19}$$

Formalização: Feitas as definições de métricas de avaliação, regra de actualização, máquina de estados e métodos de estimação dos parâmetros de aprendizagem, o algoritmo UbiSOM encontra-se formalizado da seguinte forma [9, 72]:

2.1.3.3 Análise de Sensibilidade do UbiSOM

Bruno Silva em [72] afirma que qualquer algoritmo têm um conjunto de parâmetros ótimos para um determinado problema. Silva refere que as melhores soluções envolvem o parâmetro β a tomar valores iguais ou superiores a 0.5, mas que a solução fica comprometida se tomar o valor 1. O valor anteriormente referido também sugere que na função de

Algorithm 2 Algoritmo do Mapa Auto-Organizado Ubíquo [9, 72]

Input: $inputs\_list \leftarrow$ todos os inputs carregados até ao momento
Parâmetros: $T \leftarrow$ tamanho da janela deslizante para as métricas de avaliação
$\beta \leftarrow$ determina a relação entre as duas métricas na função de deriva
$map\_size \leftarrow$ tamanho do mapa
procedure UBISOM
    $mapa \leftarrow$ inicializar com map_size vetores aleatórios
    $t_k^{update} \leftarrow$ vetor inicializado a zeros com o último *update* de cada protótipo
    $t_l^{bmu} \leftarrow$ vetor inicializado a zeros com a última vez que o protótipo foi seleccionado como BMU
    $t_i \leftarrow 0$
    $t \leftarrow 0$
    $current\_state \leftarrow ordering\_state$
    for all $i \in inputs\_list$ **do**
        $distances \leftarrow \forall_k \, |map_k - i|$
        $bmu \leftarrow argmin(distances)$
        $t_c^{bmu} \leftarrow t$
        $\lambda(t) \leftarrow$ Calcular utilidade do protótipo com equação 2.10 e meter na fila de $\overline{\lambda}$
        $E'_q(t) \leftarrow$ Calcular erro de quantificação normalizado com a equação 2.8 e meter na fila de $\overline{qe}$
        Calcular $\overline{\lambda}$ e $\overline{qe}$
        if $current\_state = ordering\_state$ **then**
            Calcular $\sigma(t)$ e $\eta(t)$ com equações 2.16 e 2.18
        else
            Calcular $\sigma(t)$ e $\eta(t)$ com equações 2.17 e 2.18
        end if
        Actualizar *mapa* (Equação 2.15) e t_k^{update}
        $t \leftarrow t + 1$
        if $t = t_f$ **then**
            $current\_state \leftarrow ordering\_state$
        else if $\sum 1_{d(t) \geq d(t_f)} \geq T$ **then**
            $current\_state \leftarrow ordering\_state$
            $t_i \leftarrow t$
            $t_f \leftarrow t_i + T - 1$
            $\forall_k^{update} \leftarrow t$
        end if
    end for
end procedure

deriva, o erro médio de quantização seja "favorecido". Os resultados para *streams* estacionárias e não estacionárias sugerem que os melhores valores são os valores no intervalo $\beta \in [0.6, 0.9]$.

Quanto ao tamanho da janela deslizante T, as melhores soluções são obtidas com valores acima de 1000. Silva ainda afirma que um bom intervalo para ser utilizado é $T \in [1000, 2500]$.

Em conclusão, o autor refere que durante os testes experimentais, foi descoberto que as parametrizações de $T = 2000$ e $\beta = 0.8$ executaram favoravelmente em relação a múltiplos *datasets*, sendo portanto consideradas como parametrizações admissíveis.

2.2 Unidades de Processamento Gráfico

As Unidades de Processamento Gráfico ou GPU (do inglês *Graphical Processing Unit*) são um circuito electrónico especializado em manipular memória rapidamente para acelerar a criação de imagens para serem exibidas num computador [13, 64].

A principal aplicação do GPU é a computação gráfica, mas devido ao aumento da potência destes dispositivos e à sua natureza de computação paralela, actualmente o GPU é utilizado noutras áreas. A utilização generalizada do GPU é intitulada de Unidade de Processamento Gráfico de Propósito Geral ou GPGPU (do inglês General-Purpose Graphical Processing Unit), que se pode descrever como a computação de tarefas pelo GPU que estariam normalmente ao encargo do CPU [16]. Desta forma, a capacidade do GPU em executar tarefas de forma paralela é aproveitada em computação de alta performance, nomeadamente nas áreas de computação genética [16], criptografia [45], bioinformática [46], aprendizagem automática [22], entre outras.

Nesta secção irá ser discutido os modelos de programação em GPU mais preponderantes, como o CUDA e o OpenCL, assim como *frameworks* que ainda trazem mais eficiência a esses modelos, como o *Marrow*.

2.2.1 Plataformas de Programação

A motivação de desenvolver modelos de programação proprietários relativamente ao GPU associa-se ao facto de nos primórdios da computação em GPU ser difícil ao programador expressar computações não-gráficas utilizando uma API gráfica como o OpenGL [45]. Para tal, os maiores vendedores de GPUs, NVidia e AMD, para permitir GPGPU nos seus dispositivos utilizam respectivamente o CUDA e o OpenCL. O CUDA é uma tecnologia proprietária da Nvidia, enquanto que o OpenCL pertence ao consórcio Khronos Group, cuja pesquisa e desenvolvimento se concentram em tecnologias de código aberto. O **CUDA** é uma plataforma de computação paralela e um modelo de programação heterogéneo criado pela Nvidia para GPGPU. Esta plataforma tem como objectivo acelerar a computação de aplicações utilizando o poder disponibilizado pelos GPUs, utilizando o CPU e o GPU em uníssono [57, 4].

Esta plataforma permite aos GPUs Nvidia executar código implementado em linguagens como C, C++ ou Python. No caso do C/C++, a compilação é feita pelo *nvcc*, um compilador de C/C++ para CUDA, proprietário da Nvidia, baseado em LLVM, que faz a compilação de linguagem de alto nível para instruções máquina.

Um programa CUDA é dividido numa componente sequencial, constituído por uma ou mais componentes que executam no CPU (denominado como *host*) e numa componente paralela que executa no GPU de acordo com o modelo STMD (*Single Thread Multiple Data*). Neste paradigma, a componente paralela que executa no GPU é denominada como *device*. Neste modelo, o papel do *host* é orquestrar o lançamento de *kernels* e a manipulação da memória (tanto no *host*, tanto no *device*), não executando efectivamente nenhum segmento de código [4].

Os processos no GPU estão organizados em blocos de processos. Um kernel pode ser executado em diferentes processos e em diferentes blocos de processos, que se denomina por grelha (do inglês *grid*) [45]. Os processos de diferentes blocos, desde que estejam na mesma grelha, coordenam-se através de operações atómicas numa memória global partilhada por todos os processos [55].

A plataforma CUDA possuí aplicações em vários ramos: computação financeira, pesquisa espacial e meteorológica, *data mining* e análise, entre outros [56].

Dada a importância do OpenCL no âmbito deste projecto, a secção 2.2.2 será dedicada à descrição do OpenCL.

2.2.2 Open Computing Language

O OpenCL é uma *framework* aberta para programação paralela que utiliza a combinação de CPUs, GPUs entre outros tipos de processadores. O objectivo desta plataforma é programar aplicações paralelas e *portable*, sendo possível executar as mesmas em ambientes heterogéneos e em qualquer dispositivo, independente do seu fabricante. Desta forma, consegue-se elevar a velocidade e responsividade das aplicações com a junção de vários dispositivos de características diferentes [25, 60, 70].

Apesar da plataforma possuir o seu próprio conjunto de APIs (*runtime*, plataforma e *OpenCL C*), permite a portabilidade com outras interfaces, nomeadamente a do OpenGL. As APIs partilham o mesmo tipo de estrutura de dados e localizações de memória utilizadas, sendo descomplicada a interoperabilidade entre as duas interfaces, sem cópias redundantes ou *overhead* na conversão de estruturas de dados [3].

O OpenCL possuí aplicações em áreas como a computação gráfica [1], computação científica [59], criptografia [19], entre outras. As aplicações na área da Aprendizagem Automática são relativamente reduzidas, havendo por exemplo implementações de *Support Vector Machines* [65] e aceleradores da execução de redes neurais convolucionais [5].

Comparando esta *framework* ao CUDA, não existe uma clara vantagem na performance apresentadas pelas duas plataformas, dado que a performance está dependente da qualidade do código apresentado e do *hardware* utilizado. Contudo, a portabilidade

parece ser o factor mais importante em termos da escolha entre uma *framework* e outra: o CUDA é uma plataforma que apenas executa código em *hardware* NVIDIA, enquanto que o OpenCL pode ser adaptado a um leque mais vasto de marcas de *hardware*. Ainda assim, o CUDA tem uma ligeira vantagem em questão das bibliotecas que podem ser utilizadas, sendo mais extensas do que aquelas disponiblizadas pela *framework* OpenCL [20].

2.2.2.1 Modelo da Plataforma

O modelo da plataforma OpenCL é definido como um hospedeiro (do inglês *host*) ligado a um ou mais dispositivos OpenCL [3], conforme representado na figura 2.3. Um hospedeiro pode ser qualquer CPU com vários ou um único núcleo e tem como função interagir com o ambiente externo à aplicação implementada em OpenCL, seja por operações I/O ou interações com o utilizador da aplicação. Os dispositivos OpenCL consistem em colecções de dispositivos que são constituídos por um ou mais núcleos (e.g. GPU, DSP, CPU com múltiplos núcleos). Estes dispositivos são responsáveis pela computação das *streams* de instruções que provém do hospedeiro, sendo frequentemente chamados de Dispositivos de Computação (do inglês *compute devices*) [60].

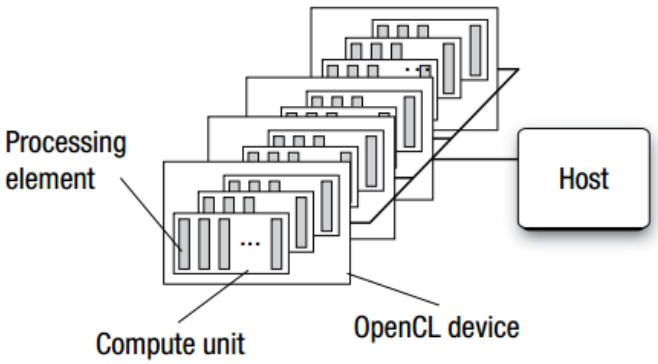


Figura 2.3: Modelo da plataforma OpenCL [60].

Cada um dos dispositivos OpenCL contêm múltiplas **Unidades de Computação**, que por sua vez é constituída por múltiplos **Elementos de Processamento** (EP). Efectivamente, a computação ocorre nos EPs. Os EPs executam instruções SIMD (*Single Instruction Multiple Data*) ou SPMD (*Single Program Multiple Data*). Normalmente, as instruções SPMD são executadas em CPUs, enquanto as instruções SIMD requerem um processador de vetores, que existe em dispositivos como o GPU. Por exemplo, o GPU ATI Radeon HD 5870 é constituído por 20 unidades SIMD, que se traduz em 20 UCs em OpenCL. Cada unidade SIMD contém 16 núcleos que processam *streams* de instruções, onde cada núcleo possuí 5 EPs. Assim sendo, cada UC no GPU anteriormente especificado, possuí cerca de 80 EPs [3].

2.2.2.2 Modelo de Execução

Uma aplicação OpenCL é constituída por um programa principal que executa no hospedeiro e por uma colecção de um ou mais *kernels* que executam nos dispositivos OpenCL [60]. Um kernel é uma sequência de instruções referentes a um certo algoritmo e é definido no hospedeiro. O programa principal emite um comando que submete a execução do kernel num dispositivo OpenCL e cria um espaço de índices, representado na figura 2.4 (apelidado de *NDRange*) onde cada kernel executa para cada ponto desse espaço. Para além dessa funcionalidade, o programa principal define o contexto para a execução dos *kernels*.

Cada instância de um kernel apelida-se de *work-item* e é identificado pelas suas coordenadas no espaço de índices, servindo como um identificador global. Aquando da submissão do kernel, é criada uma coleção de *work-items*, onde todos os elementos executam a mesma sequência de instruções. No entanto, o comportamento individual de cada instância pode variar devido ao algoritmo ou dados em que se opera.

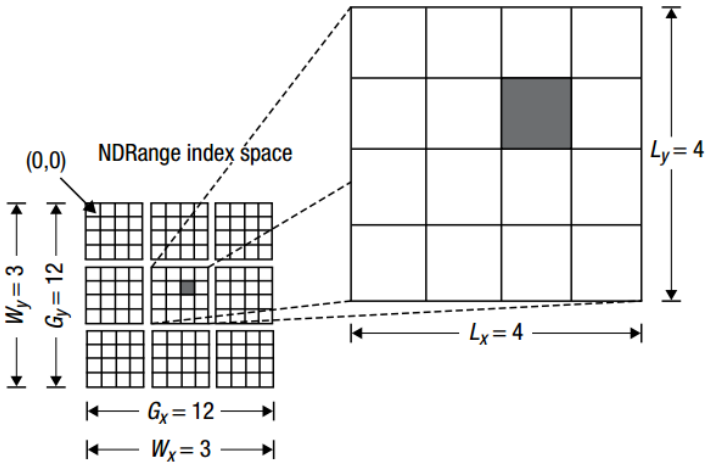


Figura 2.4: Exemplo de como os IDs globais, os IDs locais e os índices dos *work-groups* estão relacionados numa *NDRange* bidimensional [60].

Por sua vez, os *work-items* são decompostos em *work-groups*, que representam com uma granularidade mais grossa a decomposição do espaço de indíces. O OpenCL garante que apenas os *work-items* dentro do mesmo *work-group* executam concorrentemente, partilhando os recursos do dispositivo onde executam. No entanto, fica garantido de igual forma que os *work-groups* ou invocações de um kernel executam concorrentemente.

2.2.2.3 Modelo da Memória

O modelo de memória do OpenCL define 4 regiões de memória acessíveis aos *work-items* aquando da execução de um kernel [3]:

- **Memória Global**: Região de memória em que todos os *work-items* e *work-groups* tem

acessos de leitura e escrita tanto nos dispositivos como no hospedeiro. Esta região só pode ser alocada pelo hospedeiro durante a execução de uma aplicação

- **Memória Constante**: Região da memória global que se mantém constante durante a execução de um *kernel*. Os *work-items* possuem apenas privilégios de leitura e ao hospedeiro é permitida a leitura e escrita de objectos nesta região de memória.
- **Memória Local**: Região de memória que é utilizada para partilhar dados entre *work-items* contidos num *work-group*. Todos os *work-items* no mesmo *work-group* possuem previlégios de leitura e escrita.
- **Memória Privada**: É uma região de memória apenas acessível por um *work-item*.

Em maior parte dos casos, a memória do hospedeiro e a memória dos dispositivos OpenCL são independentes entre si. No entanto, a gestão de memória tem de ser explícita em definir a partilha de dados entre o hospedeiro e os DCs. Desta forma, os dados têm de ser obrigatoriamente movidos da memória do hospedeiro, para a memória global, por sua vez para a memória local e o processo inverso. Relativamente aos objectos de memória, o OpenCL define um modelo de consistência relaxada. Isto significa que os valores lidos da memória por um certo *work-item* não têm como garantido leituras iguais relativamente a todos os *work-items*, em qualquer altura. As leituras e escritas nos objectos de memória do OpenCL, podem surgir em ordem diferente para diferentes *work-items* [60].

2.2.2.4 Modelo de Programação

Os modelos de programação que são adotados no OpenCL são o **paralelismo de dados** e **paralelismo de tarefas**, podendo igualmente ser combinados num modelo híbrido. O modelo de execução do OpenCL foi projetado com o paralelismo de **dados** sendo o principal alvo, de forma a rentabilizar a potência do GPU ao máximo, dado que os GPUs contêm múltiplos núcleos e melhor se adaptam a este modelo, através do espaço de índices *NDRange* [70]. O modelo de paralelismo de dados trata-se de mapear cada elemento (ou um grupo de elementos) de um conjunto de dados para cada *work-item*. Ou seja, trata-se de aplicar uma sequência de instruções concorrentemente aos elementos de uma estrutura de dados. Para enriquecer este modelo programático, o OpenCL disponibiliza paralelismo de dados hierárquico, permitindo que os dados sejam repartidos de duas formas: no modelo **explícito**, o programador tem responsabilidade em definir o número de *work-groups* e como os *work-items* são divididos pelos *work-groups*, sendo que no modelo **implícito**, é a plataforma que define quantos *work-groups* terão de ser criados e mantidos [60].

O modelo de paralelismo de **tarefas** trata-se de definir uma tarefa como um kernel que executa como um único *work-item*, independente do número de *work-groups* que foram inicializados por outros *kernels* na mesma aplicação. O paralelismo neste modelo obtêm-se utilizando as unidades SIMD nos dispositivos OpenCL, que utilizam tipos de dados vetoriais ou utilizando as múltiplas tarefas que estejam em fila de espera para serem executadas assincronamente [70].

2.2.3 *Frameworks* de Alto Nível para GPGPU

Os modelos de baixo nível como o OpenCL e o CUDA dão-nos acesso direto ao potencial máximo que poderá ser extraído de um GPU. No entanto, os detalhes de implementação distraem o programador da tarefa a ser efetuada, dado que o GPU possuí uma arquitetura complexa [9]. Surgem então as *frameworks* de alto nível como o SkelCL [75] e o Caffe [35], que fazem uma abstracção da arquitetura do GPU e permitem uma abordagem que facilite a programação para estes dispositivos. Nesta secção irão ser apresentadas *frameworks* de alto nível que permitem programar para GPUs e permitam um maior foco no problema que queremos implementar.

2.2.3.1 Marrow

O Marrow é uma *framework* de esqueletos algorítmicos para orquestração de computações OpenCL [49, 9]. O Marrow expande o conjunto de esqueletos que existe atualmente disponíveis em GPGPU e permite a sua combinação, através de *nesting*, em estruturas complexas. A *framework* permite a sobreposição de comunicação e computação, juntando a simplicidade na implementação com ganhos de performance em vários cenários aplicacionais. O Marrow disponibiliza esqueletos para GPU como o *pipeline*, *stream* e *loop*, permitindo que sejam "aninhados"(*nested*), oferecendo assim um modelo de programação flexível.

Um dos esqueletos disponibilizados pelo Marrow é a ***pipeline***. A *pipeline* combina eficientemente uma série de tarefas serializáveis, dependente de dados, onde o paralelismo é atingido pela computação de diferentes etapas simultaneamente. Considerando a sobrecarga que é criada pelas transferências de memória entre o hospedeiro e os dispositivos, este esqueleto é ideal para a execução em GPU, dado que os dados intermédios não precisam de ser transferidos de volta para o hospedeiro, de forma a estarem disponíveis para a próxima etapa. Este padrão de execução é adequado para execuções que começam com objectos na memória do dispositivo pré-inicializados e conseguem realizar a computação num ambiente de execução partilhada. Este esqueleto suporta o *nesting*.

2.2.3.2 *TensorFlow*

O TensorFlow é uma plataforma ponto-a-ponto aberta para implementação de algoritmos de Aprendizagem Automática [23]. No fundo, o TensorFlow é uma biblioteca de alto nível que auxilia o programador a implementar algoritmos de Aprendizagem Automática. Apesar de não ser uma *framework* especializada em GPGPU, tem suporte para execução em GPU [24], especialmente ao nível do *Deep Learning*, onde o treino deste tipo de algoritmos é bastante eficiente em GPU [67].

Existem algumas implementações do SOM em TensorFlow [15, 38], implementadas em *Python* e com recurso à API do TensorFlow. Em [15] o autor treina o mapa com uma variante do SOM, o Batch-SOM, admitindo que tem mais velocidade de processamento

se se utilizar o GPU. Acrescentou também suporte a múltiplos GPUs e modificou o *input* para receber os dados por um Tensor invés do *tf.placeholder*, permitindo *pipelines* mais rápidas e que permitem ter dados de entrada mais complexos.

Em [38] apesar da implementação do SOM utilizar a API TensorFlow, o algoritmo proposto possuí algumas mudanças no seu processo de aprendizagem como é o caso da introdução da metodologia de transferência de aprendizagem, que sai fora do escopo desta dissertação e pelo qual não vai ser explorado. Foi demonstrado que implementações do modelo SOM utilizando o CPU e GPU têm complexidade temporal linear. Em termos do *speed-up* (em comparação com uma implementação do modelo SOM sem utilizar a API do TensorFlow), a implementação em CPU tem ganhos médios de 19x utilizando um Intel Core i9 e a implementação em GPU tem ganhos médios de 100x utilizando um Nvidia Tesla e 102x utilizando uma Nvidia GeForce.

2.3 SOM e UbiSOM em Unidades de Processamento Gráfico

Conforme se mostrou na secção 2.2, o GPU atualmente tem mais aplicações que apenas a computação gráfica, podendo agora acelerar a velocidade de processamento de algoritmos e realizar computação de alto desempenho. Os avanços feitos no campo da GPGPU permitiram aos programadores aceder ao potencial máximo de um GPU e inclusive fazendo com que se foquem apenas no algoritmo que querem implementar e não nos detalhes de baixo nível do dispositivo em questão.

O Mapa-Auto Organizado (2.1.2) e o UbiSOM (2.1.3) contêm oportunidades de paralelização e desta forma, os algoritmos podem ser acelerados em GPU, com objectivo de processar mapas de grande dimensão.

2.3.1 Mapas Auto-Organizados

O algoritmo SOM possuí oportunidades de paralelização devido à sua natureza paralela e às computações pesadas envolvidas, como é o caso da fase de atribuição [50]. As computações mais pesadas, no caso do SOM, são ao nível dos dados, comprovando que se pode utilizar um modelo de paralelismo de dados [53].

Existe um consenso na utilização de técnicas de *MapReduce* para efectuar a pesquisa pela BMU, dado que é das operações mais pesadas de todo o algoritmo [50, 81]. As unidades do mapa não estão dependentes entre si para encontrar a BMU, nem na propagação das distâncias para a vizinhança da BMU. No entanto, os elementos do *input* são processados de forma sequencial, havendo uma dependência no resultado do processamento do elemento do *input* anterior [53]. Wittek e Darányi em [81] apresentam uma implementação do Batch-SOM [3], assente num modelo de *MapReduce* que soluciona o problema anterior. Os autores ainda acrescentam uma otimização relativamente ao cálculo da BMU,

[3] Variante do algoritmo SOM em que os *inputs* são analisados em *batch*, invés do método online que os inputs são analisados um a um.

onde omitem o cálculo da distância Euclidiana e optam por calcular o quadrado da distância Euclidiana, justificando que a raiz quadrada é uma operação pesada em GPU.

Os acessos à memória são uma parte fundamental para a eficiência de um algoritmo a executar em GPU. Para isso, o algoritmo tem de conter uma minimização de acessos à memória global e terá de evitar conflitos nas memórias partilhadas de SMPs (*Symmetric Multi-Processor*) [81]. Mathew e Joy em [50] limitam os acessos não coalescidos [4] à memória global com recurso a texture binding do *input*, deixando desta forma o *input* em *cache* e disponível a todos os processos a executar no GPU.

Tabela 2.1: Análise comparativa entre implementações do SOM em GPU. Pes. BMU representa o método utilizado para pesquisar a BMU; D.E. representa o método utilizado para calcular a distância Euclidiana; A.M.G. é o acesso à memória global; *Speedup Máximo* representa o melhor resultado obtido perante implementações do SOM sequenciais em CPU.

Impl.	Pes. BMU	D. E.	A. M. G.	Arqui.	*Speedup* Máx.
[81]	*MapReduce*	$Dist^2$	Não coalescido	Distribuído	10x
[53]	*Classic Reduction*	Omissão	Não coalescido	GPU *Cluster*	40x
[50]	*MapReduce*	Omissão	Não coalescido	1 GPU	84x
[9]	*Marrow Reduction*	$Dist^2$	N/A	1 GPU	77x

Em conclusão, a implementação do SOM em GPU com recurso a técnicas de redução como o *MapReduce*, acelerou o algoritmo. Todas as implementações obtiveram *speedup* em relação a implementações do algoritmo em CPU, seja em mapas de 128x128 ou de 2000x2000. Os ganhos de performance em [50] em relação a outras implementações podem derivar do uso de uma técnica de redução clássica, mas também do facto de possuírem apenas 1 GPU na arquitetura, revelando que os acessos à memória globais são mais difíceis de sincronizar (e mais caros) utilizado um *cluster* de GPUs.

2.3.2 Mapas Auto-Organizados Ubíquos

2.3.2.1 Arquitetura e Otimizações

João Borrego em [9] apresenta uma implementação do UbiSOM em GPU com recurso ao Marrow. O sistema é constituído por duas vertentes, representadas na figura 2.5:

- um **servidor** que recebe de forma contínua conjuntos de dados discretizados de tamanho arbitrário, a partir de uma fonte de dados, através de uma *pipeline* para tratamento do fluxo de dados.
- uma **aplicação visualizadora** que interage com o servidor. Essa aplicação pode fazer vários pedidos ao servidor, como por exemplo, pedir o estado atual do mapa, pedir as BMUs para as ultimas n iterações do treino e ainda pedir a BMU para uma

[4] Acessos à memória global **coalescidos** são múltiplos acessos à memória numa transação apenas, por um grupo de *threads* [78].

determinada observação. Esses pedidos são tratados por uma *pipeline* de tratamento de dados, que é diferente da *pipeline* relativa ao servidor.

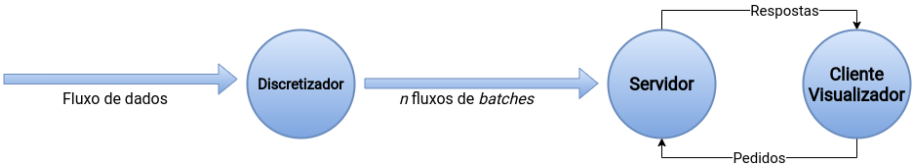


Figura 2.5: Modelo do sistema UbiSOM em GPU [9].

A *pipeline* de processamento da fonte de dados possuí dois estados: recepção de observações e treino, seguindo posteriormente para a escrita dos resultados do mapa. Conforme se observa na figura 2.6, a etapa de treino é constituída por várias fases, entre elas:

- Cálculo das distâncias entre cada unidade e a observação.
- Seleção da BMU.
- Atualização do mapa.

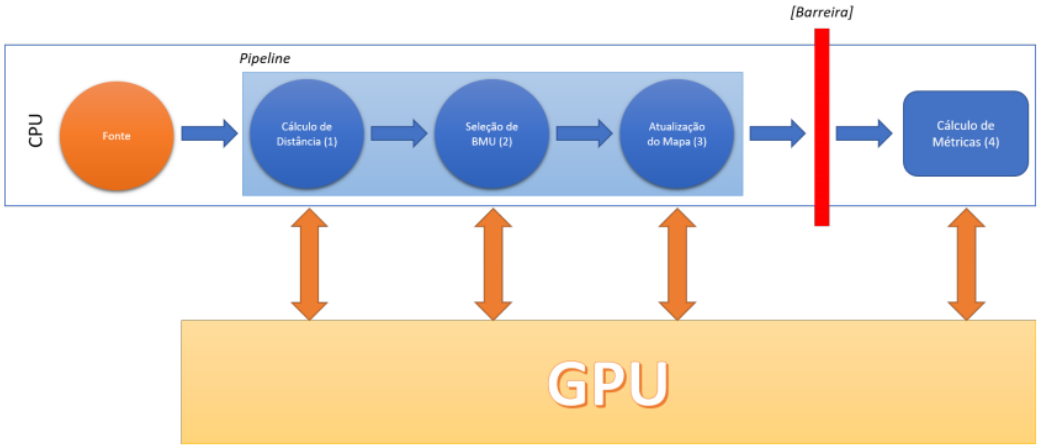


Figura 2.6: *Pipeline* de treino no UbiSOM em GPU [9].

O autor refere que é possível obter um nível de paralelismo ainda maior ao aumentar o número de *pipelines* de treino a executar em paralelo. Ainda refere que para além do paralelismo de dados no GPU, é possível ter várias instâncias do algoritmo de treino a executar em paralelo, de forma semelhante a existir várias múltiplas *pipelines*.

Quanto ao modelo de memória, as informações como o mapa do UbiSOM, cálculo de distâncias e o *array* com a última actualização são carregadas na pré-inicialização do algoritmo, ou seja, operações efetuadas antes da execução da *pipeline*. Essas informações são armazenadas de forma **persistente** no GPU, ao longo de toda a execução. Após a execução da *pipeline* terminar, seguem-se os cálculos das métricas do UbiSOM, para analisar o estado atual do algoritmo, estando essas observações alocadas **temporariamente** na

memória do GPU. Segundo o autor, a implementação UbiSOM em GPU focou-se em 3 componentes principais do algoritmo que são paralelizáveis: cálculo da distância Euclidiana entre as unidades, seleção da BMU e atualização do mapa. Existem também *kernels* relativos ao cálculo das métricas do UbiSOM. Portanto, os métodos são os seguintes:

- No cálculo da distância Euclidiana optou por fazer o quadrado das distâncias, por razões discutidas na secção 2.3.1.

- A **seleção da BMU** foi feita com o segmento de minimização da equação 2.4, utilizando o *buffer* de distâncias obtido anteriormente, que é parcialmente reduzido no GPU, mas que chegando a um valor pré-determinado de elementos, esses elementos são encaminhados para o CPU, onde a redução é terminada.

- A **fase de atualização** é realizada com recurso a um método que tem por base a equação 2.15. É feito o cálculo da função de vizinhança para cada uma das unidades do mapa em relação à BMU. Após esse cálculo, os valores resultantes são multiplicados à taxa de aprendizagem, para obter a magnitude de modificações para cada unidade. Essa magnitude é aplicada à diferença entre a observação e cada unidade do mapa, sendo o seu resultado guardado e depois agregado ao mapa.

2.3.2.2 Considerações Finais

Borrego validou os resultados obtidos comparando visualmente os resultados da implementação UbiSOM em GPU com a implementação UbiSOM em CPU, feita por Bruno Silva em [72], entre outras medições.

Os parâmetros do UbiSOM utilizados foram uma taxa de aprendizagem a variar entre 0.1 e 0.08, o raio normalizado a variar entre 0.6 e 0.2, T com tamanho de 2000 e $\beta = 0.8$. O autor chegou a ter ganhos de *speedup* entre 10x e 77x, argumentando que a maior parte do tempo que o CPU perde é no cálculo da distância Euclidiana e na fase de atualização.

Borrego argumenta que o aumento das características de um *dataset* beneficia o *throughput* do algoritmo UbiSOM, tendo por exemplo ganhos de performance até 77x com o *dataset* Irís com 16 características. No entanto, mesmo com os ganhos observados pelo uso do GPU, o CPU continua a ter um efeito bastante preponderante na velocidade de processamento; se o CPU não for capaz de acompanhar o ritmo da execução, vai deixar o GPU à espera de operações, reduzindo os ganhos de performance.

O autor conclui dizendo que os resultados com a implementação em GPU ainda não representam a elasticidade perante fontes de dados não estacionárias, declarando que uma possível teoria está relacionada com a implementação da funcionalidade do reinício do algoritmo ou algum detalhe das equações de ajuste dinâmico do modelo ou até com o facto dos estados iniciais do mapa terem sido distintos. Outro aspeto que pode ser melhorado é o número de transferências entre o CPU e o GPU, e.g. na obtenção da 2ª BMU, é preciso transferir coleções inteiras entre o CPU e o GPU, que pode gerar baixa

taxa de utilização dos dois processadores. O autor sugere transferir todas as observações a ser processadas para dentro do GPU, evitando ter que transferir cada observação individualmente à medida que essa observação é necessária.

2.4 Exemplo de aplicação dos Mapas Auto-Organizados

Encontrar padrões úteis em dados financeiros requer perícia analítica e bastante esforço por parte do analista. Por exemplo, o mercado de ações contém uma larga quantidade de dados que vai variando ao longo do tempo. A complexidade dos dados financeiros e as tarefas envolvidas em analisar estes dados requerem uma ferramenta que possa amplificar a percepção humana no conceito financeiro. Desde a criação do SOM, o algoritmo tem sido requisitado para esta análise [18, 36], provando ser uma ferramenta que efectivamente pode auxiliar os investidores, analisando o mercado de ações razoavelmente.

Nesta secção serão exploradas aplicações do modelo SOM (algumas aplicações já foram especificadas em 2.1.2) em contextos financeiros, como ferramenta de visualização de dados dos mercados de ações e de auxílio à decisão. É igualmente apresentado um caso concreto para ilustrar uma área que motivou o presente estudo da parametrização dinâmica do UbiSOM, de forma a enquadrar trabalho futuro onde podem haver aplicações directas de alguns resultados desta dissertação.

2.4.1 SOM em Contexto Financeiro

Introduzindo o contexto financeiro, o algoritmo Online-SOM já foi utilizado em vários estudos para explorar vantagens em análises financeiras, económicas e de mercado [18]. Os ciclos dos mercados (movimentos dos preços) são inconstantes e complexos de analisar, sendo o SOM uma ferramenta de apoio à visualização de padrões que possam estar neste tipo de dados.

Os SOMs podem ser utilizados para visualizar dados financeiros, assim como podem ser utilizados para classificar as empresas como saudáveis ou mais prováveis de falirem. Kiviluoto e Bergius em [39, 18] propõem uma ferramenta baseada numa hierarquia de 2 SOMs para analisar relatórios financeiros. O objectivo desta ferramenta será determinar possíveis falências em pequenas e médias empresas, com modificações no algoritmo Online-SOM. O conjunto de dados utilizado neste estudo consiste numa coleção de pequenas e médias empresas finlandesas, sendo feita uma análise a um histórico parcial, envolvendo indicadores que medem a rentabilidade e solvência. Foram analisados 11072 relatórios financeiros, dados por 2579 empresas, das quais 756 faliram, portanto existindo 2606 relatórios financeiros que foram dados pelo menos 5 anos antes da falência. Utilizando este tipo de recursos, os autores estão a perpetuar uma análise fundamental de indicadores financeiros.

Os dados são pré-processados com uma técnica apelidada de **equalização de histograma**, feita para cada indicador. De seguida, são feitos os treinos para cada nível do SOM.

O SOM do primeiro nível está treinado com os relatórios financeiros anuais, para que num determinado ano, uma empresa possa ser posicionada no primeiro mapa baseada no seu relatório financeiro de um certo ano. O treino do SOM do segundo nível é feito com os vetores de distância obtidos no treino anterior, que são uma concatenação de relatórios durante 2 ou 3 anos consecutivos, ou seja, cada ponto no SOM de segundo nível corresponde a uma **trajetória** no primeiro nível. Desta forma, a trajetória consegue captar mudanças de comportamento nos relatórios financeiros de ano para ano. As trajetórias que surgem deste modelo estão representadas na figura 2.8.

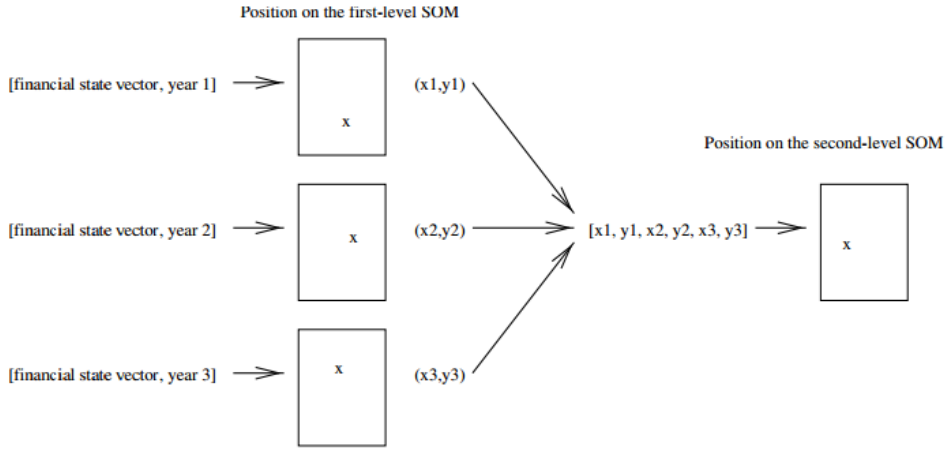


Figura 2.7: O SOM de segundo nível é treinado com vetores que são constituídos pelas posições no SOM de primeiro nível, durante 2 a 3 anos consecutivos [39]

Os autores referem ainda que utilizam um processo **semi-supervisionado** para fazer o treino dos mapas. Resumidamente, cada elemento do vetor de distâncias das unidades do mapa contém uma parte relativa ao relatório financeiro e outra parte relativa à informação do estado da empresa. A **pesquisa da BMU** é então realizada utilizando apenas a parte do vetor relativa ao relatório financeiro, mas todo o conjunto de dados é utilizado para atualizar os vetores de distâncias.

Com um elevado número de mapas resultantes, muitas interpretações podem surgir. No SOM de primeiro nível, as coordenadas correspondem aos indicadores financeiros, i.e. as coordenadas correspondem à solvência e rentabilidade das empresas: a solvência aumenta de cima para baixo e a rentabilidade aumenta da esquerda para a direita. Os mapas mostram como as empresas que convergem para cima e para a esquerda, mais se aproximam da falência. O aumento do risco de falência é proporcional à baixa solvência e à diminuição da rentabilidade.

A análise dos autores relativamente aos SOMs de segundo nível, revelou que estes capturam informação que escapa ao SOM de primeiro nível. Por exemplo, no SOM de primeiro nível, as empresas falidas desapareciam temporariamente da zona de alto risco. No entanto, no segundo nível, existem certos estados de "absorção", i.e. áreas de que empresas falidas geralmente não saem. Devido a esta propriedade, os mapas de trajetórias

aparentam ser uma ferramenta para avaliar empresas.

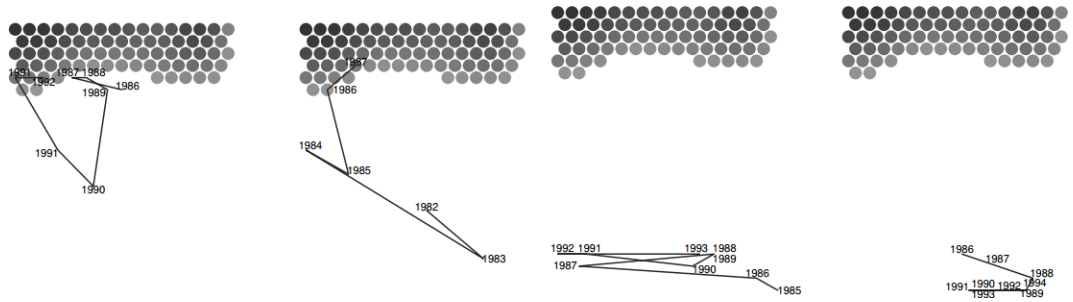


Figura 2.8: As trajetórias das duas primeiras empresas (a partir da esquerda) indicam que eventualmente faliram. Aquelas trajetórias que "fogem"às zonas coloridas, indicam que não têm alto risco de falência [39]

De acordo com os autores, o SOM de duplo nível é uma ferramenta promissora no campo da análise geral de relatórios financeiros, podendo detetar o risco de certas empresas falirem.

Existem outras implementações de modelos SOM em contexto financeiro. Como exemplo, Panosso em [62] utilizou modelos SOM em contexto financeiro, com base em análise técnica. Barroso utilizou um conjunto de dados proveniente do índice *Dow Jones Industrial Average*, num período de 9 anos consecutivos.

2.4.2 *Streams* de Dados para Análise Não Supervisionada de Dados Empresariais

Miguel Carrega et al. em [11] realizaram um trabalho sobre utilizar o modelo UbiSOM com análise fundamental de indicadores financeiros, onde comprovaram a utilidade deste modelo para visualmente localizar onde certas empresas com indicadores semelhantes se agrupam e onde se separam. Para análise de risco, este modelo comprovou-se eficaz para uso em análise financeira para saber se uma certa empresa que está a ser analisada está em risco de falência, assim como se comprovou útil para saber se o valor de uma empresa, consoante a evolução dos indicadores financeiros nos relatórios trimestrais, concluindo se poderá decrescer dentro de uma janela de tempo de um ano.

O conjunto de dados utilizado foi constituído por dados relativos aos relatórios trimestrais de 37 empresas tecnológicas presentes no SP500, o *index* onde estão presentes as 500 empresas que apresentam melhores resultados, retirados de um período de tempo que se alongou por 16 anos, entre 2003 e 2018. Foram escolhidos 12 indicadores fundamentais, tendo ainda sido normalizados em blocos de 4 anos, para o tal período de 16 anos.

Em primeira instância, foi realizado um treino do modelo UbiSOM com o parâmetro T a 2000, durante 30000 iterações para o *dataset* total de 16 anos. Após esse treino, os dados foram aglutinados em períodos bi-anuais, para serem treinados durante mais 20000 iterações, com o parâmetro T a 300, de forma a forçar uma convergência mais rápida do treino. Os resultados quanto ao erro de quantização estão presentes na figura 2.9.

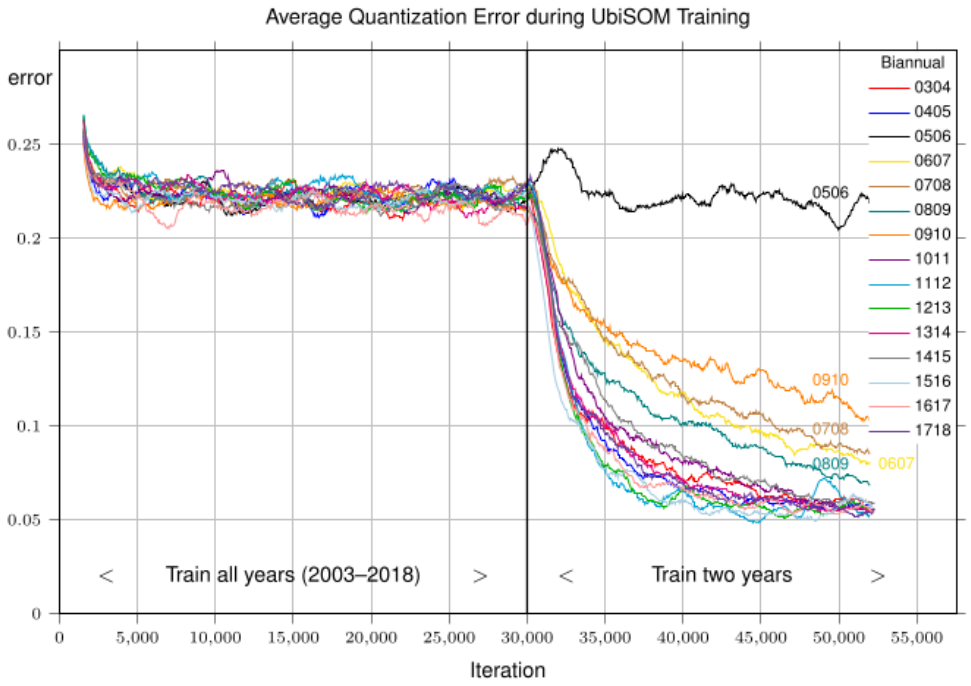
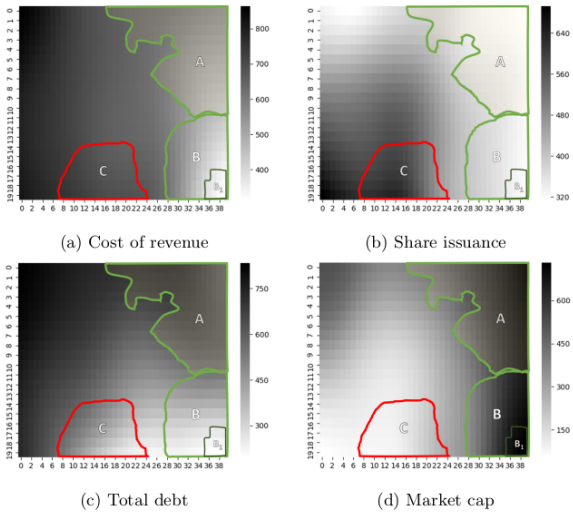


Figura 2.9: Evolução do erro de quantização médio quando se treina o modelo UbiSOM com $T = 2000$ para os dados gerais de todos os anos com 30000 iterações e de seguida, com $T = 300$ para as restantes 20000 iterações.

Para além disso, foram geradas as *component planes* relativamente aos mapas gerados. Estas *component planes* são a versão "cortada"do mapa gerado pelo modelo UbiSOM, em que cada "fatia"corresponde à representação de uma *feature* do *dataset*. As *component planes* geradas estão ilustradas na figura 2.10. Essencialmente, as conclusões retiradas foram que apenas comparando os indicadores fundamentais em trimestres distintos, o mapa gerado pelo treino do modelo UbiSOM consegue localizar diferentes empresas em diferentes áreas do mapa, assim como empresas similares nas mesmas áreas do mapa. Ao longo do tempo, dependendo da evolução dos seus indicadores financeiros fundamentais, as empresas movem-se em áreas distintas do mapa resultante do treino do modelo UbiSOM. Assim sendo, foram encontradas duas áreas (A e B) onde as empresas que mais cresceram num padrão visível. Desta forma, é possível observar a trajetória de uma empresa que tenha sucesso e assim, é plausível utilizar esta metodologia para identificar empresas promissoras para um investidor e da mesma forma, auxiliar os analistas financeiros a entender tendências macro-económicas.



(a) Cost of revenue (b) Share issuance

(c) Total debt (d) Market cap

Figura 2.10: *Component planes* com 3 áreas de estudo, para cada *feature* do conjunto de dados, para o modelo UbiSOM treinado.

2.5 *Multi-Armed Bandit*

O problema ***Multi-Armed Bandit***, na teoria das probabilidades, é um problema clássico que demonstra o dilema de *exploitation vs. exploration* [79].

Para melhor explicar este problema, habitualmente utiliza-se o exemplo em que num casino, existem múltiplas *slot machines* (também podendo ser chamadas de *bandits*) e que cada uma está configurada com uma probabilidade de obtermos uma recompensa desconhecida. Essencialmente, o que este problema/abordagem se propõe a resolver é desvendar qual a melhor estratégia para maximizar lucros a longo prazo. A ilustração 2.11 descreve como todo este processo funciona.

Utilizando ainda o exemplo do casino e das *n slot machines* não idênticas, pode ser explicado então o dilema enunciado no início desta secção. O que o problema ambiciona resolver é entender o *trade-off* entre a exploração (tentar girar todas as máquinas para encontrar a melhor) e a *exploitation* (girar apenas as máquinas que se acredita ter a melhor recompensa).

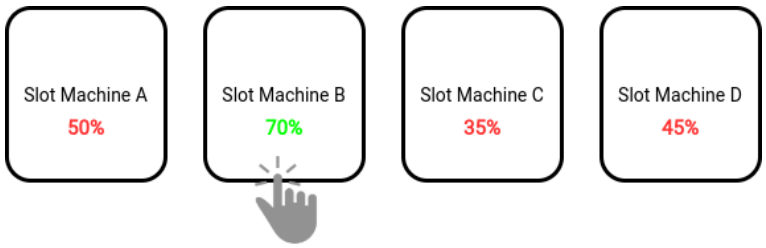


Figura 2.11: Uma ilustração de como o Multi-Armed Bandit de Bernoulli funciona. As probabilidades de recompensa não são conhecidas pelo utilizador *à priori*. A cada acção (ou seja, iteração) decide-se qual a melhor máquina a escolher de forma a maximizar a recompensa.

Resumidamente, as **estratégias** que podem ser utilizadas são as seguintes:

1. **Sem exploração:** esta abordagem é uma estratégia que envolve testar todas as hipóteses até ser conseguido a probabilidade para cada uma das mesmas. Eventualmente, segundo a Lei dos Grandes Números, essa probabilidade será determinada. Contudo, do ponto de vista computacional, é uma abordagem que desperdiça recursos e não garante a melhor recompensa a longo prazo. Esta estratégia pode ser nomeada como gananciosa (do inglês *greedy*).

2. **Exploração Aleatória:** esta estratégia é por vezes chamada de ϵ-*greedy* e trata-se de identificar a melhor acção em qualquer momento, fazendo ocasionalmente exploração aleatória.

3. **Exploração Inteligente:** como o nome indica, esta estratégia trata-se de explorar as hipóteses existentes de uma forma mais sensível que a estratégia anterior. Principalmente, as técnicas utilizadas para esta estratégia são os Upper Confidence Bounds e o Thompson Sampling.

Definição Formal Um problema Multi-Armed Bandit de Bernoulli pode ser descrito como um tuplo $< A, R >$, onde:

- Existem K máquinas com probabilidades de recompensa $< \theta_1, ..., \theta_k >$.
- A cada iteração t, realiza-se uma acção numa *slot machine* e recebe-se uma recompensa r.
- A é um conjunto de acções, cada uma relacionada com uma interacção com uma *slot machine*. O valor da acção a é a **recompensa esperada** ($Q(a) = E[r|a] = \theta$). A acção a_t, na iteração t, na máquina i representa-se da seguinte forma:

$$Q(a_t) = \theta_i \tag{2.20}$$

- R é a função de recompensa. Neste caso, observamos uma recompensa r de uma forma estocástica [5]. Na iteração t, $r_t = R(a_t)$ pode retornar uma recompensa 1 com uma probabilidade $Q(a_t)$ ou 0.

O objectivo é maximizar a **recompensa cumulativa** (representada na equação 2.21):

$$\sum_{t=1}^{T} r_t \tag{2.21}$$

A **probabilidade de recompensa ótima** da acção ótima a^* está representada na equação 2.22:

[5] Processos que não estão submetidos senão a leis do acaso.

$$\theta^* = Q(a^*) = max_{a \in A} Q(a) = max_{1 \leq i \leq K} \theta_i \tag{2.22}$$

A **função de arrependimento** é o total do "arrependimento"que possa existir quando não escolhemos a acção ótima, na iteração t (2.23):

$$L_t = E[\sum_{t=1}^{T} (\theta^* - Q(a_t)] \tag{2.23}$$

Algoritmo ϵ-Greedy Conforme apresentado acima, podem ser utilizadas inúmeras estratégias para resolver o MABP. Uma dessas estratégias é o algoritmo ϵ-Greedy.

No algoritmo ϵ-Greedy, o valor de uma acção é estimado de acordo com experiências anteriores, onde é feita uma média das recompensas associadas a uma certa acção, sendo contabilizadas todas as experiências até a um instante de tempo t.

Na equação 2.24 está representada esse algoritmo formalmente:

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{\tau}^{t} r_\tau 1[a_\tau = a] \tag{2.24}$$

Neste algoritmo, com uma pequena probabilidade ϵ tomamos uma acção aleatória, mas caso essa probabilidade não esteja a favor, é escolhida a melhor acção que se aprendeu até esse momento de decisão (representada na equação 2.25). $N_t(a)$ é a quantidade de vezes que uma acção a foi escolhida até determinado momento.

$$\hat{a}_t^\star = \arg\max_{a \in A} \hat{Q}_t(a) \tag{2.25}$$

Upper Confidence Bounds A exploração aleatória oferece uma oportunidade para testar opções que são desconhecidas previamente à sua escolha. Contudo, devido a essa aleatoriedade, é possível explorar uma má opção que havia sido confirmada como viável anteriormente. De forma a evitar esta exploração ineficiente, podem ser adoptados dois métodos [79]:

- Decréscimo do valor de ϵ ao longo do tempo;
- Favorecer a exploração de opções com alto potencial de obter um valor ótimo.

O algoritmo ***Upper Confidence Bounds*** mede este potencial com recurso a um limite de confiança superior relativo ao valor da recompensa ($\hat{U}_t(a)$) de forma a que o valor real esteja abaixo do limite $Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a)$, com alta probabilidade.

O limite superior $U_t(a)$ é uma função de $N_t(a)$. Um maior valor de tentativas de $N_t(a)$ deverá resultar num limite reduzido de $\hat{U}_t(a)$.

No algoritmo UCB, deverá sempre ser escolhida a acção mais ambiciosa para maximizar o limite de confiança superior.

Thompson Sampling *Thompson sampling* é uma outra das técnicas que pode ser utilizada para resolver o problema *Multi-Armed Bandit*. É um algoritmo utilizado em problemas de decisão *online*, onde as acções são tomadas sequencialmente de uma forma em que exista um equilíbrio entre o *exploit* de opções que se sabe previamente que maximizam a performance imediata e onde é feito um investimento na acumulação de nova informação que possa aumentar a performance no futuro [69].

A cada instante temporal, a preferência é escolher as acções em que a recompensa dessa acção seja ótima [79].

2.6 Ferramentas para a Visualização Interactiva

A visualização interactiva dos dados tem um papel importante na medida em que pode servir para tirar ilações sobre os dados utilizados para treino de um certo algoritmo e para tirar conclusões sobre padrões existentes no resultado desses treinos. Desta forma, torna-se um processo intuitivo e acelera a retirada de conclusões, assim como serve para ajustar os parâmetros associados a um certo algoritmo [34]. Esta secção contém a descrição de algumas linguagens e bibliotecas que poderão ser utilizadas para visualizar interactivamente o comportamento algorítmico perante certos dados e certos condicionalismos.

2.6.1 *Processing*

O *Processing* é uma biblioteca/linguagem gráfica e simultaneamente um IDE, tendo o propósito de ser utilizado para artes electrónicas e *new media art*, essencialmente servindo para ensinar indivíduos que não saibam programação utilizando um contexto visual [66, 21].

Esta linguagem/biblioteca utiliza a linguagem Java com simplificações ao nível de classes adicionais e funções matemáticas polidas, assim como uma *Graphical User Interface* (GUI) para descomplicar o processo de compilação e execução.

O *Processing* incluí um *sketchbook* para organizar os projectos. Cada *sketch* do *Processing* é na verdade uma sub-classe da classe *PApplet*, que implementa maior parte das funcionalidades da linguagem/biblioteca Processing [52].

Esta linguagem/biblioteca permite que os utilizadores criem as própias classes dentro do *PApplet*. Isto permite que tipos complexos de dados possam ser utilizados, invés dos tipos de dados *standard* como inteiros, *char*, números de vírgula flutuante e cores (RGB, RGBA, hexadecimais).

Exemplo de Implementação A integração do *Processing* no Java pode ser feita de dois modos: um modo básico, onde figuras estáticas são desenhadas no *canvas* e o modo contínuo.

O modo contínuo de integração do *Processing* em Java trata-se de utilizar a API do *Processing*, nomeadamente os métodos *setup()* e *draw()*. O método *setup()* é executado

uma vez aquando da inicialização do programa e tem o propósito de definir propriedades do *PApplet*, como o tamanho da janela, a cor do fundo do *canvas*, entre outros.

O método *draw()* é essencialmente um método que executa um *loop* contínuo daquilo que está contido no método. Desta forma, se aglutinarmos o conceito de Aprendizagem Automática ao Processing e quisermos visualizar um treino de um qualquer algoritmo, é neste método que a sua implementação terá de estar feita. Tanto a parte algorítmica do modelo em questão tanto a parte do desenho dos pontos de um mapa, e.g.

Ou seja, no caso dos Mapas Auto-Organizados, o conjunto de dados inicial é iterado dentro deste método e à medida que os pontos são treinados, as mudanças nos protótipos reflectem-se no mapa que é desenhado.

Segue-se um exemplo de pseudo-código (listagem 2.1) utilizando o *Processing* para desenhar uma representação do mapa resultante de um treino do modelo SOM. A figura 2.12 ilustra o exemplo concreto utilizado para esta dissertação.

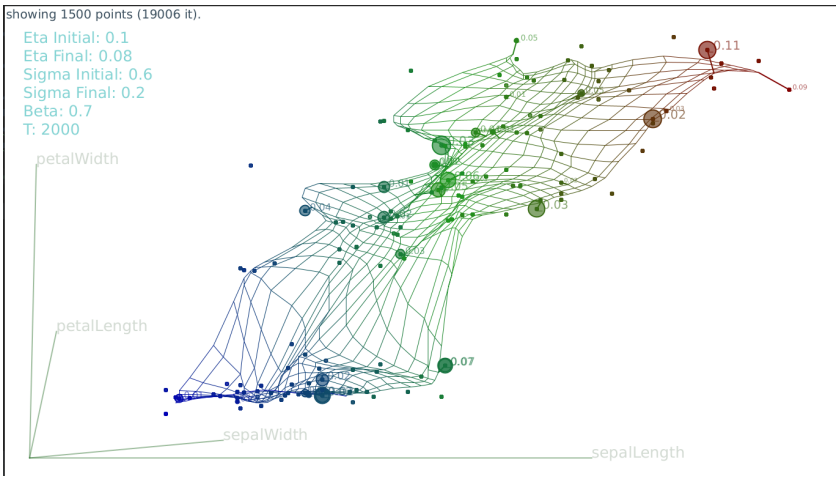


Figura 2.12: Exemplo de mapa resultante gerado pela aplicação MultiSOM.

Listagem 2.1: Pseudo-código de um exemplo de visualização interactiva para o algoritmo SOM.

```
1  public class SOM extends PApplet {
2      Dataset dataset;
3      Integer numberOfIterations; // Iteracoes desejadas    priori da
           execu  o do treino
4
5      public void setup() {
6          size(1000, 1000);
7          noStroke();
8          background(255);
9      }
10
11     public void draw() {
```

```
12          while(dataset.hasNext() || numberOfIterations == currIteration
                ) {
13              trainPoint(dataset.next()); // Entrega um ponto ao modelo
                    para ser treinado
14              updateVisualization(); // Efectivamente, reflecte as
                    mudan as que o ponto anterior gerou nos prot tipos e
                    desenha o estado actual do mapa
15              currIteration++; // Contador do n mero de iteracoes
16          }
17      }
18  }
```

3

Arquitetura

Neste capítulo são abordados os detalhes relativos à arquitetura do sistema proposto para treinar vários modelos do UbiSOM em paralelo, utilizando múltiplas parametrizações e várias unidades de computação. Estão descritas as arquiteturas da execução dos modelos UbiSOM em CPU (secção 3.2) e em GPU (secção 3.3).

3.1 Modelo da Infraestrutura

A infraestrutura da aplicação utilizada neste projecto encontra-se esquematizada na figura 3.1, onde se pode observar com detalhe as várias componentes do sistema e o respectivo fluxo do mesmo. A aplicação recebe um conjunto de dados e as parametrizações referentes ao modelo UbiSOM. Essa inicialização é realizada com auxílio da *Command Line Interface*, dado que o sistema necessita de *scripts* com propriedades (explicados em detalhe na secção 3.2). Após as propriedades serem definidas, é dada a ordem para os treinos do modelo UbiSOM começarem assim como a inicialização do mecanismo Multi-Armed Bandit Protocol.

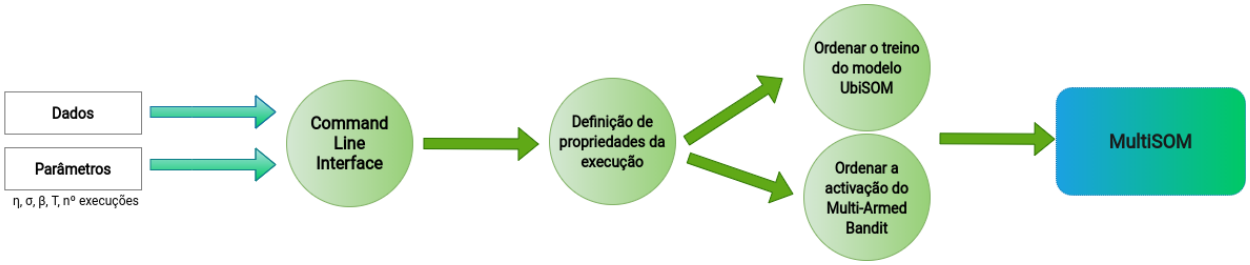


Figura 3.1: Modelo do Sistema.

No início da execução desta aplicação, é introduzido no sistema um conjunto de dados de tamanho arbitrário. Apesar do treino do modelo UbiSOM necessitar uma *stream* de dados para ser executado, neste caso em específico, simula-se uma *data stream* através de um ficheiro com dados históricos.

A parametrização dos treinos pode ficar a cargo do utilizador, sendo este a definir os parâmetros num ficheiro próprio para o efeito ou deixar o programa aleatoriamente

escolher parâmetros para executar os treinos.

No fundo, este sistema é constituído por várias etapas, tratando-se de uma *pipeline* de treino / processamento de dados, conforme ilustrado em 3.2. Esta *pipeline* contém as seguintes fases:

1. Analisar parametrização do UbiSOM definida num ficheiro / gerar parâmetros UbiSOM aleatórios, limitados a intervalos de acordo com a análise de sensibilidade realizada por Bruno Silva em [72], excepto para os parâmetros σ e η, que são gerados dos intervalos $0.0 \leq \eta, \sigma \leq 1.0$

2. Instanciação dos treinos com parametrizações definidas anteriormente.

3. Iteração sobre um conjunto de dados para "alimentar" o processo de treino.

4. Treinar até um número limite de iterações.

5. Guardar os relatórios afectos a cada um dos treinos, com a devida identificação dos parâmetros e da unidade de processamento onde o treino foi realizado.

6. Caso o protocolo de selecção de *features* esteja activo:

   a) Agrupar os resultados de todas as instâncias.

   b) Aplicar uma redução nos valores estimados de cada *feature* em cada instância e calcular os novos valores estimados para essas *features*, de forma a serem utilizados em futuros treinos.

A *pipeline* anteriormente descrita serve para um contexto onde se utiliza apenas uma unidade de computação para realizar os treinos do UbiSOM (e modelos SOM). Contudo, este modelo pode escalar para várias execuções envolvendo mais do que uma unidade de computação. Novamente, se as instâncias não estão dependentes de interação entre si e sendo assim a *pipeline* apenas precisa de ser expandida para quantas unidades de processamento se pretende utilizar. Além disso, não existe um entrave no caso de se gerar parâmetros aleatórios, dado que a probabilidade de dois treinos possuírem exactamente os mesmos parâmetros gerados ao acaso é quase nula.

No **conjunto de parâmetros** de inicialização das execuções, estão contidos tanto os parâmetros específicos do UbiSOM (secção 2.1.3.1) como de índole geral:

- **Hiper-Parâmetros do modelo UbiSOM**: Definição dos valores para os hiper-parâmetros do UbiSOM, nomeadamente T, β, σ e η.

- **Local da execução:** Especificação do local (ou locais) onde se pretende treinar os modelos UbiSOM.

- **Número de execuções:** Número de treinos que se pretendem instanciar.

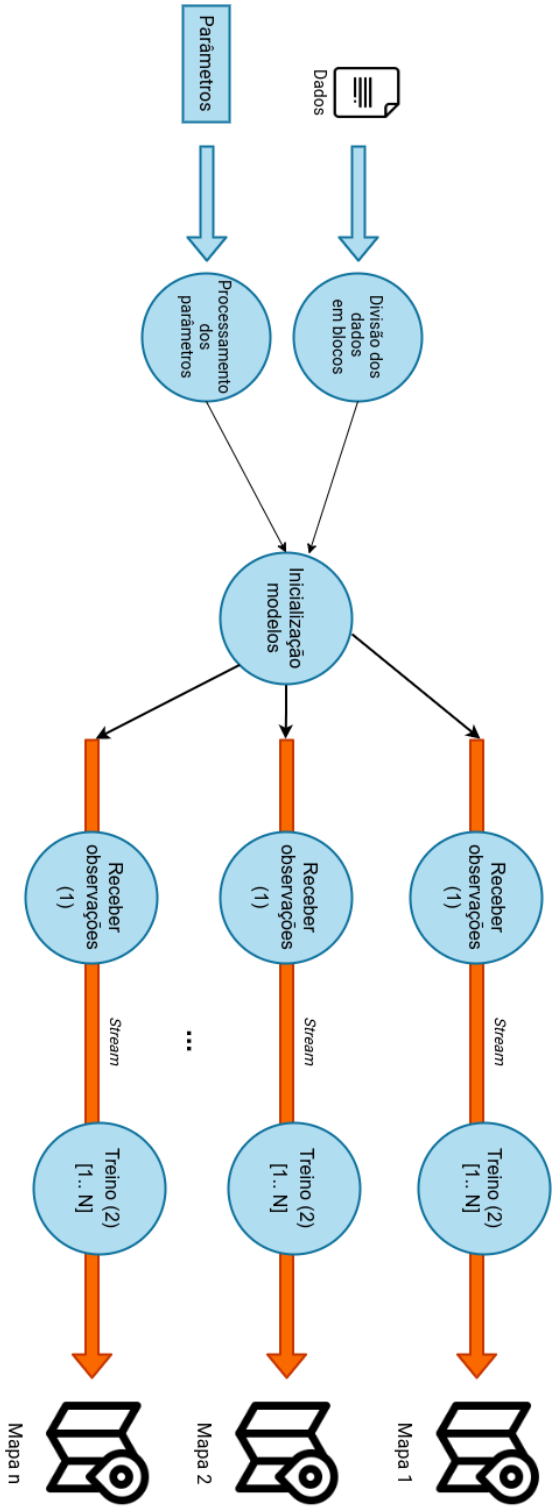


Figura 3.2: *Pipeline* de processamento de dados.

- **Protocolo *Multi-Armed Bandit*:** Activação do protocolo que visa analisar as *features* de um certo conjunto de dados, ligando ou desligado as mesmas e analisando a variação do erro médio de quantização.
- **Modo de iteração:** Os *inputs* dos conjuntos de dados podem ser iterados de uma forma **sequencial** ou de uma forma **aleatória**.
- **Número de iterações:** Valor pretendido para o número de iterações máximo para cada instância de treino.

```
CPU_Multiple # Local da execução (CPU, GPU, Múltiplos CPUs)
2/2 # Número de execuções c/ visualização / Número de execuções s/ visualização
0.1f 0.08f 0.2f 0.07f 0.3f 0.06f 0.4f 0.05f # Valores iniciais e finais da taxa de aprendizagem (eta)
0.6f 0.2f 0.5f 0.1f 0.4f 0.09f 0.3f 0.05f # Valores iniciais e finais do raio da vizinhança normalizado (sigma)
0.7f 0.6f 0.5f 0.4f # Valores do peso da função de deriva (beta)
2000 1700 700 500 # Tamanho da janela deslizante
```

Figura 3.3: Exemplo do ficheiro que contém os múltiplos parâmetros para as várias execuções. Neste ficheiro em específico pode-se observar os **parâmetros referentes aos modelos SOM** e aos parâmetros do protocolo **Multi-Armed Bandit**.

Um exemplo da especificação dos parâmetros está representada em 3.3. O significado daquilo que cada linha representa encontra-se devidamente descrito na figura.

No caso da parametrização aleatória, os hiper-parâmetros do modelo UbiSOM são gerados nos seguintes intervalos:

- Taxa de aprendizagem (η): $0.0 < \eta_i, f < 1.0$
- Peso da função de deriva (β): $0.6 < \beta < 0.9$
- Raio da vizinhança normalizado (σ): $0.0 < \sigma_i, f < 1.0$
- Tamanho da janela deslizante (T): $1000 < T < 2500$

Estes parâmetros foram escolhidos com base na análise de sensibilidade efectuada por Bruno Silva, conforme descrita em 2.1.3.3. Contudo, este sistema não garante que os valores gerados para a taxa de aprendizagem e para o raio da vizinhança normalizado diminuam à medida que o treino do modelo UbiSOM decorre, conforme é necessário para garantir a convergência do mapa resultante.

Após as parametrizações serem geradas aleatoriamente pelo sistema (no caso dos parâmetros específicos do modelo UbiSOM) ou o *parse* do ficheiro de parâmetros ser concluído, o módulo central responsável pela orquestração dos vários processos, instancia os treinos requisitados. Com as instanciações feitas, os dados são continuamente enviados para cada uma das instâncias de treino. Cada uma destas trata os seus dados de forma independente, não havendo uma dependência global entre instâncias. As observações do conjunto de dados são enviadas e processadas individualmente, não estando associado um esquema de iteração de *batching*. Não obstante, o conjunto de dados será igual para todos os treinos instanciados. Naturalmente, a arquitetura apresentada nesta secção tem possibilidade de utilizar *data streams* como *input*.

Com a meta de iterações atingida, os treinos são concluídos e os relatórios de métricas do modelo UbiSOM de cada instância são guardados em disco rígido. No caso do protocolo de selecção de *features*, posteriormente a todos os treinos do modelo UbiSOM instanciados para execução paralela terminarem, existe uma componente que agrupa os resultados da execução do Multi-Armed Bandit Protocol de todas as execuções e faz uma redução dos atributos associados a esse mecanismo para cada *feature*, nomeadamente do número de vezes que uma *feature* foi escolhida e o número de vezes que uma *feature* "gerou"recompensa. Desta forma, obtém-se os valores estimados de cada *feature* actualizados para o âmbito geral das execuções que aconteceram e assim, poderão ser utilizados em execuções futuras, sobre o mesmo conjunto de dados utilizado.

De um ponto de vista geral, é assim que o sistema se encontra definido e estruturado. Um coordenador de modelos que orquestra a instanciação dos treinos, uma *pipeline* que efectivamente faz com que os dados fluam para os treinos e permite a execução destes e um módulo que permite que os valores obtidos pelo protocolo de *feature selection* sejam calculados para futuras execuções. De seguida, iremos então detalhar o MultiSOM, sendo esta a aplicação que efectivamente alberga todos estes mecanismos e explorar alguns detalhes da mesma.

3.2 MultiSOM: Treino do modelo UbiSOM em CPU

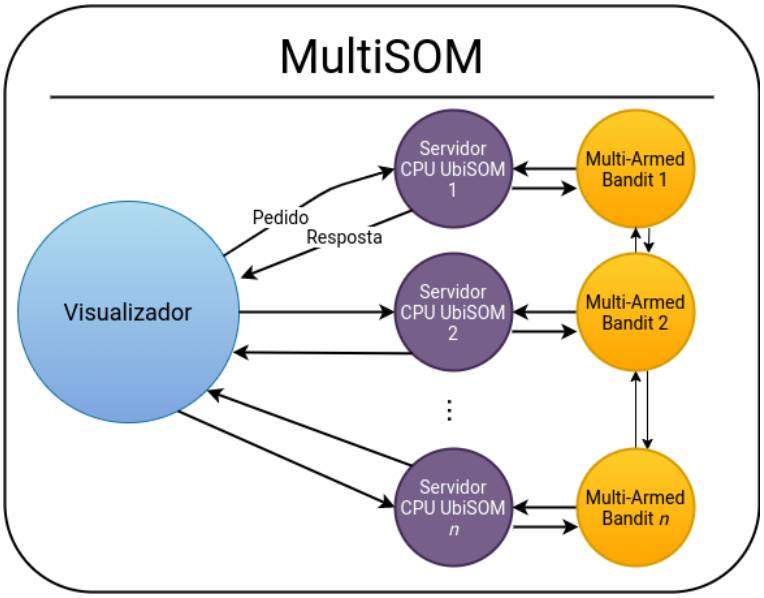


Figura 3.4: Modelo do módulo MultiSOM.

O MultiSOM é um *software* desenvolvido pelo professor Nuno Marques cujo objectivo é treinar Mapas Auto-Organizados [12, 48]. A particularidade desta aplicação é a variedade de níveis de visualização possíveis de serem obtidos a partir de um determinado treino. Nomeadamente, a aplicação permite visualizar o mapa resultante com recurso a uma Unified Distance Matrix ou com recurso a uma rede n-dimensional, onde cada eixo desta rede

é representativo das *features* do conjunto de dados. A esquematização desta aplicação está ilustrado na figura 3.4, onde se pode observar as várias componentes de treino que são instanciadas em paralelo e associadas a estas, instâncias das componentes que fazem o cálculo do protocolo *Multi-Armed Bandit*. As componentes do visualizador e do servidor são as componentes principais, responsáveis pelo treino do modelo UbiSOM (servidor) e pela visualização dos treinos (visualizador). A implementação destas componentes encontram-se descritas no capítulo 4. A componente do Multi-Armed Bandit Protocol (secção 4.4) é uma componente de integração secundária, que posteriormente é testada de forma semi-automática e por *scripting* na secção 5.2.2. As componentes Multi-Armed Bandit Protocol estão associadas individualmente a cada treino do modelo UbiSOM (servidor) que esteja a ocorrer e comunicam entre si os resultados obtidos, ao nível da escolha de *features*, assim como comunicam ao servidor esses mesmos resultados. O servidor comunica com estas componentes secundárias principalmente para comunicar o valor de métricas do modelo UbiSOM em certas alturas temporais.

Este programa permite então a utilização do algoritmo UbiSOM de forma a analisar em tempo real o treino de um conjunto de dados, algo que com o algoritmo SOM em conjunto com outro tipo de programas de treino não seria possível. Com recurso à projecção multi-dimensional, torna-se possível de serem detectadas irregularidades no mapa resultante ou relações não lineares entre as *features*.

Quanto ao processo de treino da aplicação, inicialmente, o programa recebe um ficheiro CSV, devendo este conter nas suas colunas uma etiqueta (relativa ao contexto do *dataset*) seguida por várias colunas relativas às *features* deste conjunto de dados. Este ficheiro CSV pode ser igualmente integrado no programa com recurso a *scripts* de inicialização proprietários. O programa na sua fase inicial pode igualmente receber uma *stream* de dados.

Estes **scripts proprietários** (de extensão *.mSOM*) são utilizados para definir as iterações do treino, gravar os resultados deste, configurar os CSVs pretendidos para execução, parâmetros gráficos para visualização do treino (como o tamanho dos eixos, o *zoom* pretendido em certas zonas do mapa, etc.), imprimir resultados no terminal, entre outros usos. Um exemplo pode ser visto na listagem I.2.

Anteriormente às alterações realizadas no MultiSOM no âmbito deste projeto, o programa apenas necessitava do ficheiro CSV para o treino do mapa. Sendo necessário executar vários treinos do modelo UbiSOM, com múltiplas parametrizações, o programa contém agora um módulo onde é feita a **recepção dos variados parâmetros** pretendidos pelo utilizador. Na sua essência, a listagem dos parâmetros tem uma estrutura semelhante aos *scripts* mencionados acima. Um exemplo está ilustrado na figura 3.5.

Durante o processo de treino é possível observar o estado do mapa através de uma representação gráfica da Unified Distance Matrix ou através de uma rede n-dimensional, conforme pronunciado anteriormente. Para a utilização dos recursos computacionais de uma forma mais eficiente, no caso de estarem a ser treinados múltiplos mapas simultaneamente, foi introduzida a opção de fazer um treino sem qualquer representação gráfica.

```
CPU_Multiple # Local da execução (CPU, GPU, Múltiplos CPUs)
2/2 # Número de execuções c/ visualização / Número de execuções s/ visualização
0.1f 0.08f 0.2f 0.07f 0.3f 0.06f 0.4f 0.05f # Valores iniciais e finais da taxa de aprendizagem (eta)
0.6f 0.2f 0.5f 0.1f 0.4f 0.09f 0.3f 0.05f # Valores iniciais e finais do raio da vizinhança normalizado (sigma)
0.7f 0.6f 0.5f 0.4f # Valores do peso da função de deriva (beta)
2000 1700 700 500 # Tamanho da janela deslizante
```

Figura 3.5: Exemplo do ficheiro que contém os múltiplos parâmetros para as várias execuções. Neste ficheiro em específico pode-se observar os parâmetros referentes aos modelos UbiSOM.

A Unified Distance Matrix é representada através de um mapa de cores (traduzido do inglês *colormap*), onde está exposto a distância de protótipos adjacentes. A cor azul é representativa de distâncias quase nulas, enquanto que a cor vermelha representa distâncias mais elevadas. O mapa contém igualmente várias elipses, onde cada elipse tem um tamanho proporcional à contagem das vezes que um determinado protótipo foi uma BMU num número previamente determinado de iterações. Uma das UMat resultantes de um treino realizado pelo programa MultiSOM pode ser observada na figura 3.6. Esta interface contém igualmente um gráfico com a variação do erro topológico e do erro de quantização ao longo das épocas de treino.

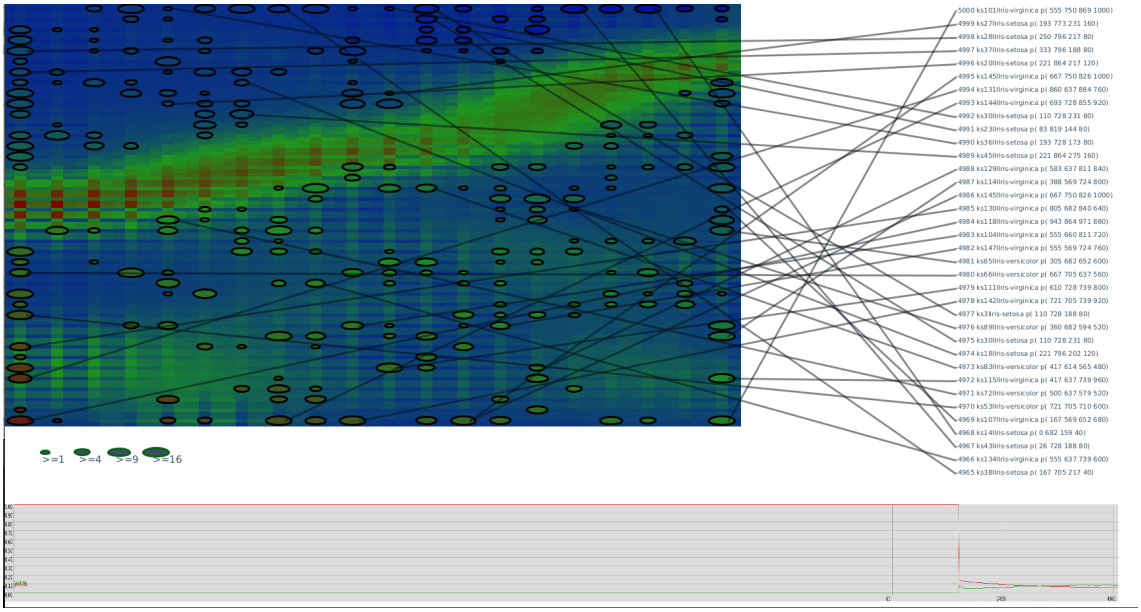


Figura 3.6: Uma Unified Distance Matrix resultante de um treino no MultiSOM. São destacados no lado direito os protótipos com maior número de vezes que foram considerados uma BMU. Em baixo, está o gráfico representativo da variação do erro topológico e do erro de quantização ao longo das iterações.

A projecção do modelo SOM treinado pode ser visualizada numa rede com n-dimensões, contendo tantos eixos conforme as *features* que um determinado conjunto de dados contém. Esta rede é visualizada em 2 dimensões, contudo, utilizando uma vertente de análise chamada PCA (*Principal Component Analysis*), é possível analisar certas zonas do mapa, focando/rodando os sub-conjuntos relevantes ou dados que tenham entrado no treino há menos tempo. Desta forma, disponibiliza-se ao utilizador uma forma diferente de

explorar os dados treinados através de ajustamentos nos pesos dos eixos, sendo possível visualizar os dados numa outra perspectiva, podendo observar o impacto de um certo protótipo numa determinada iteração, algo que não seria possível de concluir apenas com uma UMat [48]. Um exemplo pode ser observado na figura 3.7.

Figura 3.7: Rede bi-dimensional resultante de um treino com a aplicação MultiSOM.

Aquando do término de um treino de um determinado modelo, o *output* pode ser guardado de duas formas: **um ficheiro CSV** contendo os protótipos (e os seus respectivos pesos) do mapa, a sua contagem de BMUs e valor na UMat ou num **ficheiro proprietário do treino UbiSOM**, onde este pode ser revisto (ou visto pela primeira vez, caso se tenha decidido treinar sem visualização). Naturalmente, os treinos guardados num ficheiro serão executados e visualizados desde a primeira iteração, não podendo ir directamente para a fase intermédia ou final do mesmo. Pode-se considerar a execução de um treino como atómica, não sendo possível interferir no treino do mesmo. Contudo, o acrescento do protocolo *Multi-Armed Bandit* não elimina essa característica da aplicação, estando integrado no processo de treino.

Selecção de *features* A aplicação MultiSOM, no âmbito desta dissertação, possibilita a utilização de um mecanismo de selecção de *features*, que tem como objectivo seleccionar as *features* mais preponderantes de um conjunto de dados.

O mecanismo de selecção de *features* trata-se de uma componente que que utiliza o protocolo *Multi-Armed Bandit* aliado a um mecanismo de *feature switching*. Objectivamente é uma componente que corre paralelamente ao processo de treino mas que age sobre as componentes que sustentam o treino de um conjunto de dados utilizando o algoritmo UbiSOM e não interfere com outros treinos que estejam a executar em simultâneo. Ou seja, é única para cada treino do modelo UbiSOM que é executado num determinado

instante de tempo. Nesta iteração da aplicação, pode ser utilizado o algoritmo ϵ-Greedy, um dos algoritmos utilizados no protocolo *Multi-Armed Bandit* (explicado em detalhe em 4.4). Contudo, existe extensibilidade para utilizar algoritmos como aqueles expostos na secção anterior (secção 2.5).

O mecanismo de *feature switching* é um mecanismo que activa ou desactiva *features*. Essencialmente retira (ou adiciona) uma *feature*, relativamente ao *parsing* de uma observação do conjunto de dados inicial. Ou seja, em caso da *feature* ser retirada, essa *feature* em questão vai deixar de ter os pesos actualizados (sendo igualmente relevante para o cálculo das distâncias entre protótipos e observações do *input*) enquanto a *feature* estiver desactivada, fazendo com que essa *feature* perca relevância para o treino do modelo UbiSOM. Contudo, a complexidade temporal (O(n), sendo n o número de *features* das observações do *input*) para o aspeto de actualização dos pesos/cálculo das distâncias mantém-se.

Esta componente é uma adição relevante porque poderá trazer benefícios em questão da redução do tempo de execução de um treino e porque de igual forma, reduz as *features* a um conjunto de *features* que potencialmente minimiza os erros de topologia e de quantização, facilitando a análise de um conjunto de dados que esteja a ser treinado.

3.3 Treino de modelos UbiSOM em GPU

Relativamente ao treino dos modelos UbiSOM em GPU, idealmente seria utilizado como ponto de partida a implementação do João Borrego, aquando da sua dissertação [9]. Para permitir o paralelismo de treinos, a arquitetura do modelo UbiSOM em GPU utiliza um nó distribuidor e nós de treino. A arquitetura dos modelos UbiSOM em GPU encontra-se representada na figura 3.8.

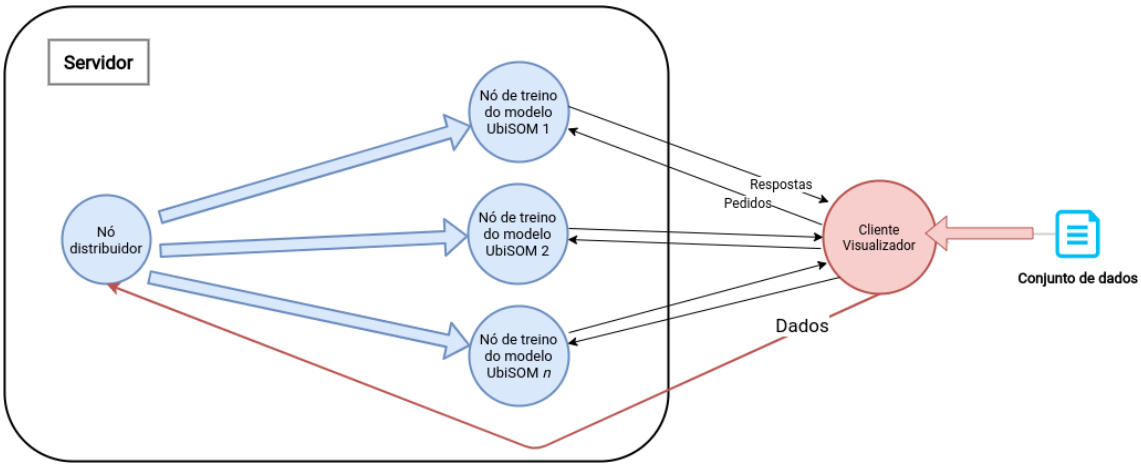


Figura 3.8: Modelo da arquitetura da execução de treinos do modelo UbiSOM em paralelo utilizando o GPU.

O nó distribuidor é responsável pela distribuição do *dataset* que é pedido para ser treinado pelo cliente visualizador, fazendo a discretização dos dados, separando-os em blocos e encaminhando para os nós de treino.

O cliente visualizador é a componente que dá início ao processo de treino, enviando comandos e o conjunto de dados para o nó distribuidor. Os nós de treino comunicam com o cliente através de *sockets*.

Os nós de treino encontram-se implementados em C++, especificamente utilizando a *framework Marrow* para permitir a computação de alta performance utilizando o GPU. Internamente, a *framework* anteriormente mencionada utiliza ainda outra *framework*, a *Thread Building Blocks*. Esta *framework* é utilizada de duas formas: uma delas sendo a divisão das tarefas em blocos para permitir a computação paralela dessas mesmas tarefas e a outra, é a arquitetura geral de nós, onde está aplicado a ideia geral de um gráfico de fluxo [33]. Para além disso, cada nó de treino possuí a *pipeline* representada em 2.6.

O cliente visualizador comunica com os nós de treino e nó distribuidor através dos *Protocol Buffer*, um método de serializar dados que poderão ser transmitidos por rede ou guardados em ficheiros. Essencialmente, é um método que permite a aplicações heterogéneas comunicarem entre elas, sem aumentar a complexidade deste processo.

No aspecto da visualização dos mapas resultantes do modelo UbiSOM, esta aplicação permite que os resultados sejam apresentados numa Unified Distance Matrix no final do treino de um modelo UbiSOM. As métricas de treino do algoritmo UbiSOM são requisitadas aos nós de treino através de mensagens *ProtoBuf*, analisadas através da *pipeline* de comandos, com um tipo de mensagem referente a esse tipo de pedido.

4

Implementação

De forma a cumprir os objectivos relativos a esta dissertação, a implementação focou-se em possibilitar a existência de múltiplas *pipelines* com múltiplas parametrizações de treino em cada uma das unidades de processamento. De forma geral, as mudanças não foram relativas ao algoritmo dos modelos UbiSOM e mais incidentes na respectiva agilização e coordenação dos módulos do sistema para permitir a execução de múltiplas *pipelines*, dado que a versão atual do algoritmo utilizado já possuí optimizações nas partes paralelizáveis do mesmo.

Como ponto de partida, foi utilizado a aplicação MultiSOM para o treino no CPU.

Nas próximas sub-secções irá ser feita uma distinção entre as mudanças realizadas na aplicação MultiSOM e no *back-end* em GPU do UbiSOM, dado que efectivamente são dois sistemas heterogéneos.

4.1 Instanciação de vários treinos em simultâneo

MultiSOM: A aplicação MultiSOM utiliza para a visualização da rede n-dimensional resultante do treino uma biblioteca gráfica chamada *Processing* (secção 2.6). Esta biblioteca permite observar a evolução do treino do mapa desenhando os protótipos numa rede n-dimensional, podendo ser observado as mudanças em tempo real, como as suas respectivas interacções entre protótipos. Esta biblioteca utiliza uma classe chamada *PApplet* para estabelecer um *canvas* onde irão ser desenhados os protótipos e inicialmente, estaria agregada a um treino de um modelo. Como estava agregada a um só modelo, seria a classe mais alta da hierarquia. Houve então necessidade de criar uma classe que estaria num nível mais elevado, para então ser possível inicializar vários treinos e ver então o progresso de cada um dos mesmos. Não só ter acesso à visualização gráfica do progresso do treino mas igualmente inicializar modelos que se pretenda não possuir visualização.

Para tal, introduziu-se uma classe (*ModelsLauncher*, figura 4.2) onde é feito a instanciação dos modelos que se pretende executar. As diferentes fases deste processamento estão descritas na figura 4.1 e a sua respectiva implementação encontra-se descrita no anexo I. Passa-se a enunciar e a descrever cada uma das fases:

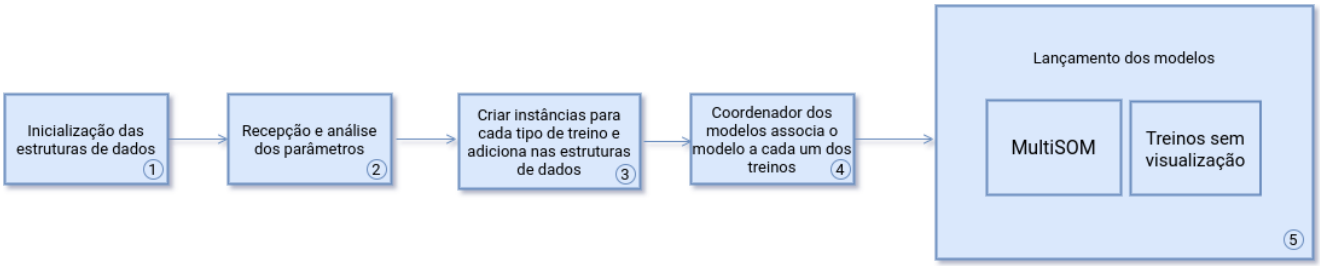


Figura 4.1: Esquema do funcionamento da instanciação de vários treinos.

- **Fase 1**: Na primeira etapa desta funcionalidade, as estruturas de dados que irão albergar cada uma das execuções dos treinos serão inicializadas. Esta etapa encontra-se descrita nas linhas 28-29 da listagem I.1.

- **Fase 2**: Nesta fase, é criada uma instância (linha 36, I.1) do controlador de parâmetros (que será visto em maior detalhe adiante). Este controlador tem como propósito fazer o *parse* dos parâmetros e encaminhá-los para os modelos correctos.

- **Fase 3**: Com as parametrizações definidas e preparadas para ser encaminhadas para as instâncias certas, é então feito a instanciação do MultiSOM, cujo treino pode ser visualizado, conforme discutido anteriormente. Este passo tem de ser realizado devido à utilização da biblioteca *Processing*. Antes de um *PApplet* ser instanciado, tem de possuir argumentos como o comprimento e largura da janela que neste caso será a janela de visualização do treino. Após a inicialização do MultiSOM, é feita a inicialização do treino sem visualização do mapa.

- **Fase 4**: Nesta fase, os parâmetros são efectivamente interligados aos modelos correctos (sejam eles com ou sem visualização) e também é associado o módulo responsável pela aprendizagem e ordenação do modelo. Esta associação é necessária porque o módulo (implementado por Bruno Silva [72]) está responsável pela fase de aprendizagem e ordenação, assim como do cálculo das métricas. Todas as outras fases do algoritmo SOM como a pesquisa da BMU e cálculo das distâncias entre protótipos, são mantidos "manualmente" pelo MultiSOM.

- **Fase 5**: Com os parâmetros nos modelos certos, começa então a fase de lançamento efectivo dos treinos. Os treinos no MultiSOM são lançados primeiro e de seguida os treinos sem visualização. Cada MultiSOM é lançado com o método *runSketch()*, proprietário da biblioteca *Processing*. Cada treino sem visualização é lançado com a biblioteca *ExecutorService*, que basicamente lança cada treino numa *thread*, correndo os treinos em processos independentes ao MultiSOM.

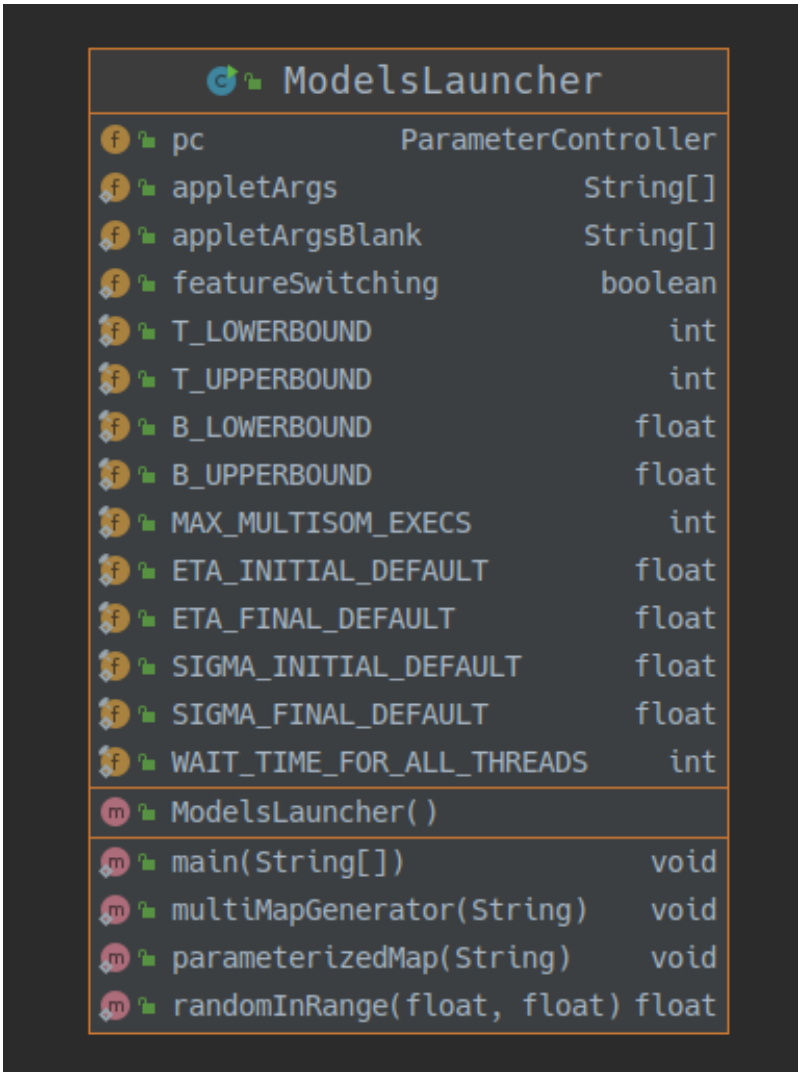


Figura 4.2: Diagrama de classe do módulo *ModelsLauncher*.

4.2 Parametrizações

A recepção e *parsing* dos parâmetros para cada treino de modelos SOM foi implementada com recurso a um ficheiro que contém os parâmetros e um módulo onde é feita a separação e *parsing* de cada um desses parâmetros. Um exemplo de um desses ficheiros pode-se observar em 4.1. A estrutura deste ficheiro é semelhante àquela dos *scripts* proprietários do MultiSOM, discutidos na secção anterior.

Listagem 4.1: Exemplo do ficheiro que contém os múltiplos parâmetros para as várias execuções. Neste ficheiro em específico pode-se observar os parâmetros referentes aos modelos UbiSOM.

```
1 CPU_Multiple # Local da execu  o (CPU, GPU, M ltiplos CPUs)
2 2/2 # N mero de execu  es c/ visualiza  o / N mero de
      execu  es s/ visualiza  o
```

```
3 0.1f 0.08f 0.2f 0.07f 0.3f 0.06f 0.4f 0.05f # Valores iniciais e
    finais da taxa de aprendizagem (eta)
4 0.6f 0.2f 0.5f 0.1f 0.4f 0.09f 0.3f 0.05f # Valores iniciais e finais
    do raio da vizinhan a normalizado (sigma)
5 0.7f 0.6f 0.5f 0.4f # Valores do peso da fun  o de deriva (beta)
6 2000 1700 700 500 # Tamanho da janela deslizante
```

O ficheiro dos parâmetros contém todas as informações referentes tanto às parametrizações a atribuir a cada um dos modelos, como o número de execuções e as plataformas onde se pretende executar os treinos. De seguida, vai ser explicado o que tem de estar representado em cada linha desse ficheiro:

1. Na linha inicial está representado o **tipo de execução pretendido**. Ou seja, se se pretender fazer treinos apenas no CPU, pode-se utilizar as opções CPU_Simple para uma execução com visualização do mapa resultante, $CPU\_Multiple$ se é pretendido treinar mapas com e sem visualização e $GPU\_Multiple$ caso se pretenda utilizar o treino em GPU.

2. Nesta segunda linha está representado o **número de execuções** pretendidas em cada unidade de processamento. No caso de serem pretendidas execuções em todas as plataformas: a primeira divisória é representativa do número de execuções em CPU com visualização do treino, a segunda divisória representa o número de execuções em sem visualização do treino e a terceira representa o número de execuções pretendidas em GPU.

3. As próximas quatro linhas são afectas aos **hiper-parâmetros do UbiSOM**. Em primeiro lugar é definido os valores iniciais e finais para a taxa de aprendizagem (σ), seguindo-se os valores iniciais e finais para o raio da vizinhança normalizado (η), o valor para β e por fim o valor da janela deslizante T. Naturalmente, os parâmetros que são constituídos por valor final e inicial, serão analisados em pares, sendo o primeiro valor o valor inicial e o último, o valor final.

A implementação da integração de vários parâmetros nas várias execuções está feita num módulo nomeado "controlador de parâmetros". Este módulo realiza o *parsing* dos parâmetros, guardando-os em estruturas de dados e posteriormente, este módulo é instanciado no "controlador de modelos", onde efectivamente os parâmetros são associados a cada um dos modelos a serem treinados. Esta implementação encontra-se esquematizada na figura 4.3.

O método que faz o ***parsing*** dos parâmetros está descrito na listagem II.1, do anexo II. A leitura das linhas do ficheiro foi feito com recurso à biblioteca *Scanner*, contida no pacote *java.util*. A limpeza dos espaços é feito com recurso ao *Pattern*, uma classe utilizada para fazer pesquisa em texto utilizando expressões regulares. Tirando os espaços de cada uma das linhas, utilizando o método desta classe *collect()*, os parâmetros são guardados correctamente na estrutura de dados.

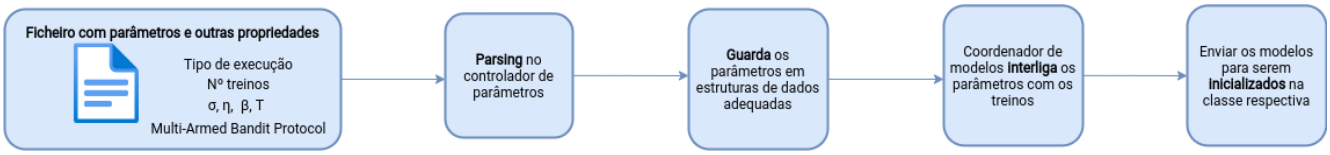


Figura 4.3: Esquema do funcionamento da recepção e análise dos parâmetros (e outras propriedades).

Após recolha dos parâmetros, são então entregues ao módulo *ModelCoordinator*, onde são iterados e atribuídos a cada um dos treinos que se pretendem inicializar. O método encontra-se detalhado na listagem II.2 do anexo II. Os parâmetros são iterados dois a dois, no caso do parâmetro em questão possuir um valor inicial e final. A esta decisão de implementação deve-se o facto de cada linha representar um parâmetro, havendo só distinção dos valores iniciais e finais aquando da análise de cada um deles.

Com os treinos parametrizados correctamente, os modelos são devolvidos à classe responsável pela inicialização de cada um dos treinos, tendo o seu processo sido detalhado na sub-secção anterior.

Parametrização aleatória Conforme descrito ao longo desta sub-secção, a parametrização pode ser feita a partir de um ficheiro onde é feita a descrição de cada um dos parâmetros dos modelos SOM. Contudo, para facilidade de interacção com o *software*, está implementado uma opção de gerar parâmetros aleatórios para cada um dos treinos.

No caso da geração de números inteiros (para o parâmetro T do modelo UbiSOM), a sua geração é feita com o método *ThreadLocalRandom.current().nextInt()*. Este método recebe dois parâmetros: um limite inferior e um limite superior.

No caso de geração de números com vírgula flutuante (para o parâmetro η, σ ou β do modelo UbiSOM), esses números são gerados com o método da listagem 4.2. No entanto, esta implementação não garante que os valores gerados para a taxa de aprendizagem (η) e para o raio da vizinhança normalizado (σ) diminuam à medida que o treino do modelo UbiSOM decorre, conforme é necessário para garantir a convergência do mapa resultante (i.e. o processo de gerar valores pode gerar um valor inicial menor que um valor final, não respeitando os condicionalismos para o algoritmo atingir a fase de convergência, assim descrito em 2.1.3).

Listagem 4.2: Método para gerar números de vírgula flutuante tendo como parâmetro um limite inferior e um limite superior.

```
1         protected static Random random = new Random();
2
3         public static float randomInRange(float min, float max) {
4         float range = max - min;
5         float scaled = random.nextFloat() * range;
6         float shifted = scaled + min;
7         return shifted;
```

```
8           }
```

4.3 Treino sem visualização do mapa resultante

Na aplicação MultiSOM, o treino dos Mapas Auto-Organizados pode ser visto em tempo real, cujos resultados são projectados numa rede n-dimensional ou com recurso à Unified Distance Matrix. Para a construção da rede n-dimensional é necessário que os pontos sejam transferidos do CPU para o GPU, de forma que a *framework Processing* desenhe esses pontos a cada iteração do treino do modelo UbiSOM. Essa transferência no caso de um treino único não é substancial, mas no cenário de inúmeros treinos em paralelo, torna-se numa operação pesada.

Para evitar esse tipo de *bottleneck* quando se executa treinos em CPU, foi realizada a implementação de um módulo que permite o treino dos modelos SOM sem a visualização quer da rede resultante, quer da Unified Distance Matrix. O algoritmo de treino (pesquisa de BMUs, cálculo da distância euclidiana entre dois protótipos) e respectivas estruturas de dados mantiveram-se semelhantes àquelas implementadas para o treino de modelos SOM com visualização. Igualmente, foram retirados métodos que não serviriam para este tipo de treino, como é exemplo do cálculo do gráfico de erro topológico/erro de quantização e do *rendering* da rede n-dimensional.

Contudo, de forma a poder avaliar os resultados dos treinos efectuados, a cada treino está associada uma Unified Distance Matrix. Esta Unified Distance Matrix apesar de não ser mostrada durante o treino, é calculada no final do treino e os seus valores são guardados num ficheiro CSV, em conjunto com os valores das *features* dos protótipos e a contagem das vezes que um certo protótipo foi BMU. Esta forma de guardar os resultados já existia na versão inicial do MultiSOM, mas a extensão feita para ser incluída neste tipo de treino é ainda mais relevante. O ficheiro de resultados em CSV pode ser utilizado posteriormente em conjunto com outros *softwares* (como é o caso da ferramenta *SOMToolbox*) para então computar a Unified Distance Matrix, justificando a sua importância.

De seguida, irão ser enunciadas as características deste tipo de treino e como está implementado:

- **Leitura de um conjunto de dados:** A aplicação base do MultiSOM já contém um *parser* de CSVs, que utiliza módulos da biblioteca *Processing* como o módulo *Table*, que alberga os valores do conjunto de dados vindos do ficheiro CSV inicial e posteriormente os guarda numa matriz.

  Anteriormente aos dados serem guardados na matriz, estes são ainda normalizados de forma a possuírem uma natureza inteira (alguns dados inicialmente são apresentados como sendo *double* ou *float*) e expandidos/diminuídos de forma a estarem num certo intervalo ($[0;1000]$). Essa expansão/diminuição é uma forma de não se

perder substância nos dados, quando se transforma os mesmos do tipo inicial decimal para um valor inteiro. O código referente a esta parte da implementação está descrito no anexo VI, listagem VI.1.

- **Treino dos modelos / *WrapperUbiSOM*:** O treino dos modelos UbiSOM está dividido em duas componentes: uma delas é a componente que faz efectivamente o treino do UbiSOM e efectua a mudança de estado de aprendizagem para ordenação (e vice-versa) e a outra componente é a componente chamada *WrapperUbiSOM* que encapsula o treino do algoritmo UbiSOM e é constituída por métodos/estruturas de dados que o módulo UbiSOM não possui, como é o caso de retornar a BMU em determinada iteração, o cálculo dos erros médios de quantização e de topologia, desligar/ligar *features*, entre outros. O módulo *WrapperUbiSOM* foi inicialmente implementado por Nuno Marques, tendo sido utilizado como ponto de partida para este projecto. Foi feito um processo de engenharia reversa para entender as várias fases existentes na componente e foram adicionadas algumas funcionalidades que não existiam. O módulo UbiSOM está representado na figura 4.4 e o este foi implementado por Bruno Silva em [72]. O diagrama de classe do módulo *WrapperUbiSOM* está ilustrado na figura 4.5, contendo apenas os métodos referentes ao módulo.

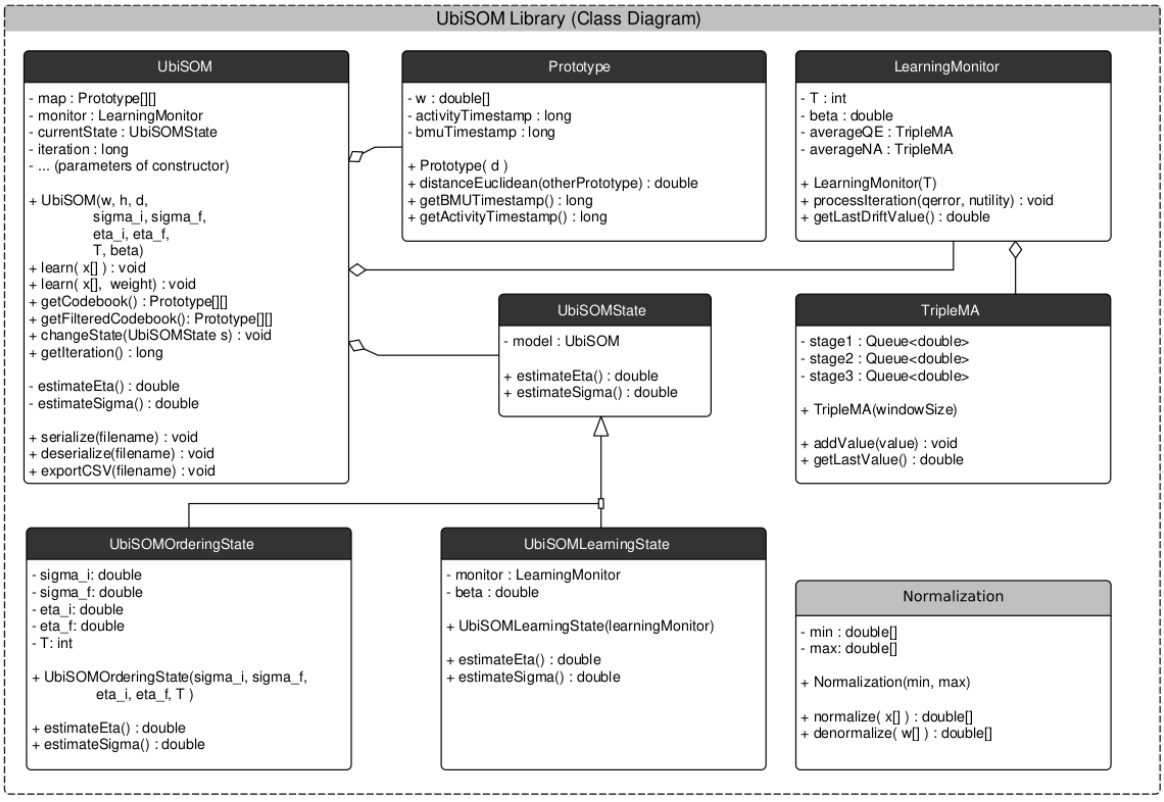


Figura 4.4: Módulo responsável pelo treino do algoritmo UbiSOM, implementado por Bruno Silva [72].

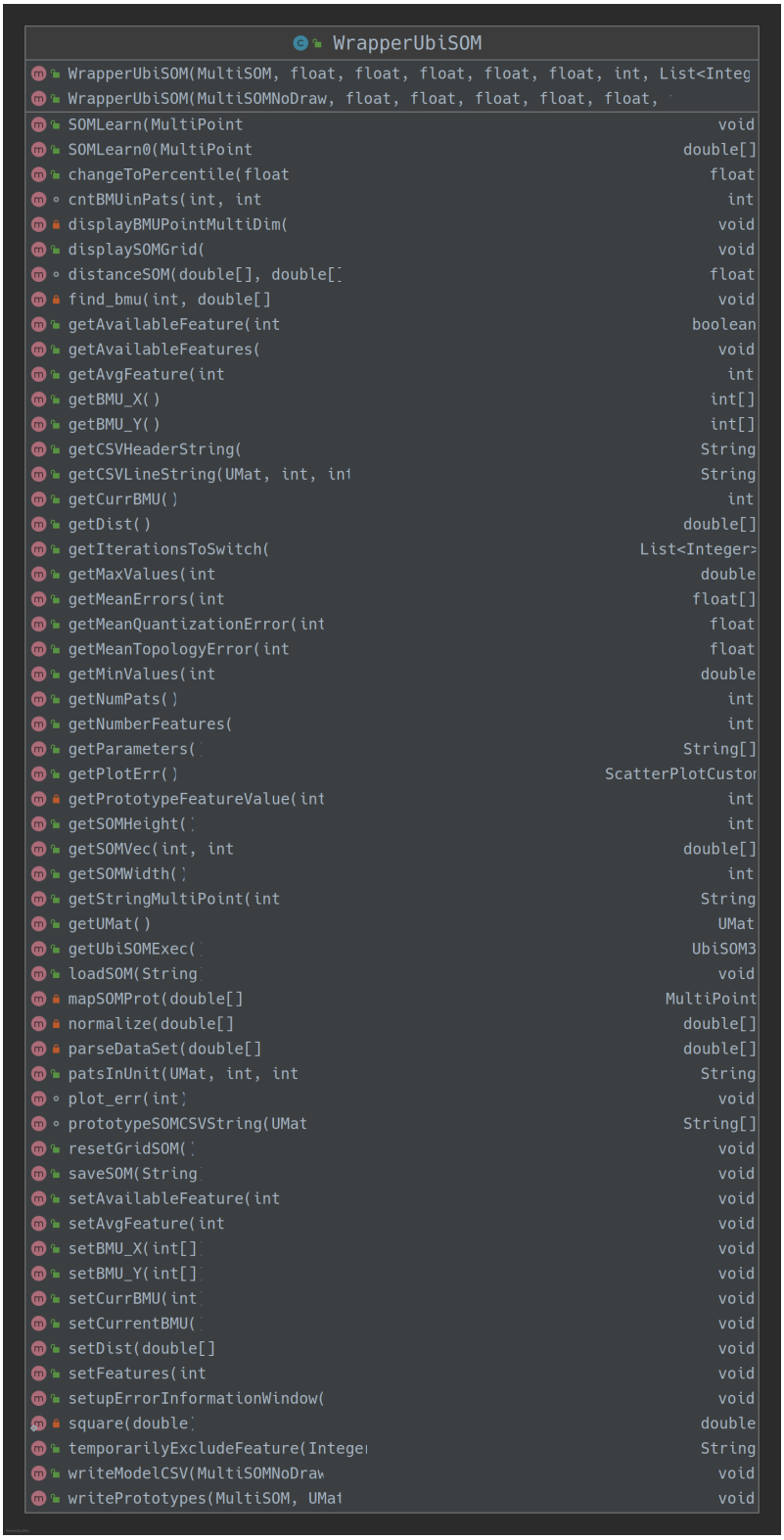


Figura 4.5: Diagrama de classe do módulo *WrapperUbiSOM*.

Conforme expressado anteriormente, o treino fica encapsulado numa só classe, permitindo então que sejam criadas várias instâncias, todas com parametrizações diferentes, utilizando apenas um conjunto de dados. A instanciação do processo de treino é feita através de uma *thread*, permitindo que vários treinos sejam instanciados em paralelo, tirando proveito de todos os recursos da unidade de computação onde estes sejam instanciados.

Conforme ilustrado na figura 4.7, neste sistema o treino de um *dataset* utilizando o algoritmo UbiSOM é feito com recurso ao *MultiSOMNoDraw* (uma classe clone da classe MultiSOM, sendo esta última a que disponibiliza a versão do treino com visualização), à classe *WrapperUbiSOM* e ao módulo UbiSOM implementado por Bruno Silva. O diagrama de classe do módulo *MultiSOMNoDraw* encontra-se representado em 4.6.

Na lista seguinte vão ser enunciadas e descritas as fases deste treino:

- ***Parsing* do conjunto de dados inicial (1)**: Esta parte do algoritmo já está explicada no parágrafo anterior a este sobre o treino do UbiSOM. Resumidamente, o *dataset* é transformado numa matriz de inteiros e conduzida até à parte onde se realiza o treino do UbiSOM. O código referente a esta parte da implementação está descrito no anexo VI, listagem VI.1.
- **Pré-aprendizagem de uma observação (2):** Com o *parsing* feito, é escolhido um ponto do *dataset* e este é entregue ao algoritmo UbiSOM para realizar a fase de aprendizagem. A aprendizagem dos pontos está limitada ao número de iterações requeridas pelo utilizador, conforme mostrado na condição do *loop*. Nomeou-se esta fase como a pré-aprendizagem porque apenas é feito o encaminhamento da observação corrente para a componente responsável pelo treino do algoritmo UbiSOM. O código referente a esta parte da implementação está descrito no anexo VI, listagem VI.2.
- **Normalização de uma observação (3):** Após o ponto ser "entregue" à componente *WrapperUbiSOM*, essa observação será normalizada, conforme descrito anteriormente nesta secção. O código referente a esta parte da implementação está descrito no anexo VI, listagem VI.3.
- **Fase de aprendizagem (4):** Com a observação normalizada, esta é encaminhada para o módulo UbiSOM para efectivamente ser realizado o treino sobre a mesma, sendo chamado o método *learn()* do módulo UbiSOM, sobre a observação p.
- **Pesquisa da BMU (5):** Apesar do módulo UbiSOM efectuar múltiplos cálculos relativos ao algoritmo UbiSOM, não tem uma forma expressa de retornar a BMU de um protótipo a cada iteração. Como tal, existe a implementação no *WrapperUbiSOM*, que após ser feita a chamada para aprendizagem de uma observação, é feita a pesquisa pela BMU dessa observação, para posteriormente

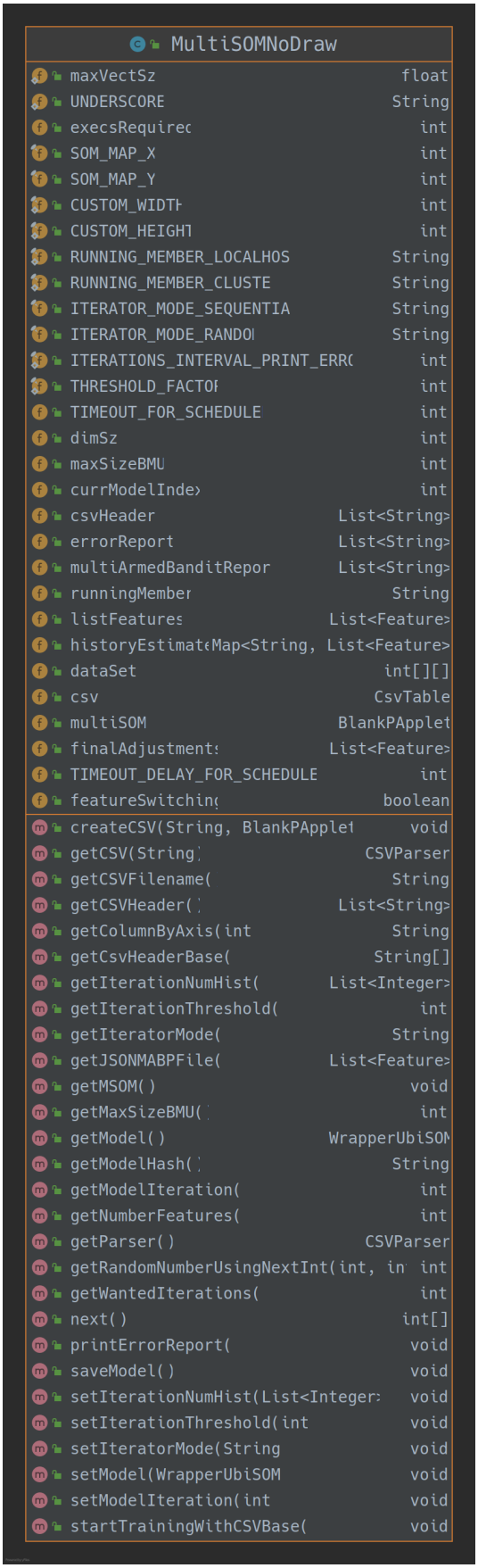


Figura 4.6: Diagrama de classe do módulo *MultiSOMNoDraw*.

ser utilizada nos cálculos da Unified Distance Matrix, assim como para efeitos de análise enquanto o treino decorre. O código referente a esta parte da implementação está descrito no anexo VI, listagem VI.4.

– **Atualização das estruturas de dados do *WrapperUbiSOM* (6):** Com a observação treinada, é feita uma actualização geral às estruturas de dados associadas ao erro médio de quantização e topológico e ao valor médio das *features* relativamente a cada protótipo. Isto é realizado após a aprendizagem de uma observação de forma a que o mapa estabilize, para que as medições sejam as mais precisas possível.

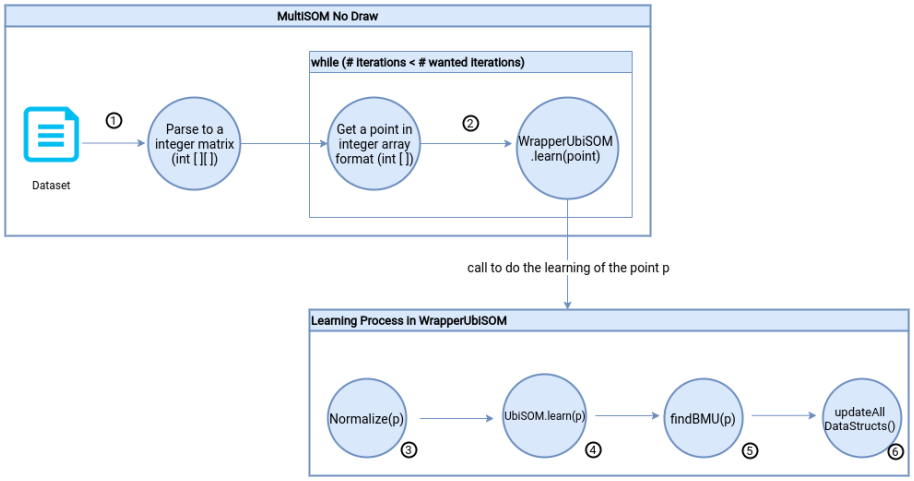


Figura 4.7: Processo do treino sem visualização do mapa resultante, utilizando o algoritmo UbiSOM.

- **Iteração dos *inputs*:** O consumo de *inputs* pelo algoritmo UbiSOM está implementado de duas formas: iteração sequencial e iteração aleatória. A forma utilizada por omissão é a forma sequencial, sendo esta a forma mais comum utilizada para iterar os *inputs*.

  A iteração aleatória dos *inputs* está implementada com recurso a uma lista que é preenchida com o tamanho do conjunto de dados inicial, isto é, cada posição dessa lista contém um número que está no intervalo entre 0 e o tamanho total dos *inputs*. Após esse preenchimento, essa estrutura de dados é "misturada", utilizando o método do Java *Collections.shuffle()*. Aquando a progressão do iterador, invés de ser processado incrementalmente, é-lhe atribuído uma das posições dessa lista, tornando assim o processo de iteração aleatório.

  A implementação dos dois iteradores está presente na listagem VI.5.

- **Gerar ficheiros de resultados:** Para além dos *logs* resultantes das execuções, existe ainda a opção de guardar os atributos dos protótipos presentes no mapa: os valores das *features*, o valor associado à Unified Distance Matrix e o valor associado à contagem de vezes que um certo protótipo foi a BMU. A implementação feita para

esta funcionalidade não foi feita com recurso ao mesmo *parser* que é utilizado para a leitura do conjunto de dados inicial, mas sim com recurso à biblioteca *Apache CSV Commons*, uma biblioteca utilizada frequentemente para este tipo de operações sobre ficheiros CSV.

Esta decisão recai sobre o facto de não existir necessidade de criar mais *overhead* e utilizar a implementação já feita para esta funcionalidade, dado que necessitava de existir uma *PApplet* que mostrasse a rede n-dimensional, guardando assim os valores directamente dos cálculos feitos por esse módulo (que seriam efetuados qualquer das formas para gerar pontos na janela). Caso se utilizasse a *PApplet*, deixaria de ser um treino do modelo UbiSOM sem visualização e poderia ter sérios impactos na performance geral que se pretende atingir com execuções de treinos em paralelo.

Sendo assim, aquando o final de um treino, os resultados são consultados às estruturas de dados utilizadas para o cálculo do algoritmo UbiSOM e são guardados linha a linha. O número máximo de linhas deste ficheiro está relacionado com as dimensões do próprio mapa (i.e. $tamanho\_output = no\_linhas * no\_colunas$).

A implementação encontra-se descrita na listagem III.1, anexo III.

4.4 *Multi-Armed Bandit Protocol*

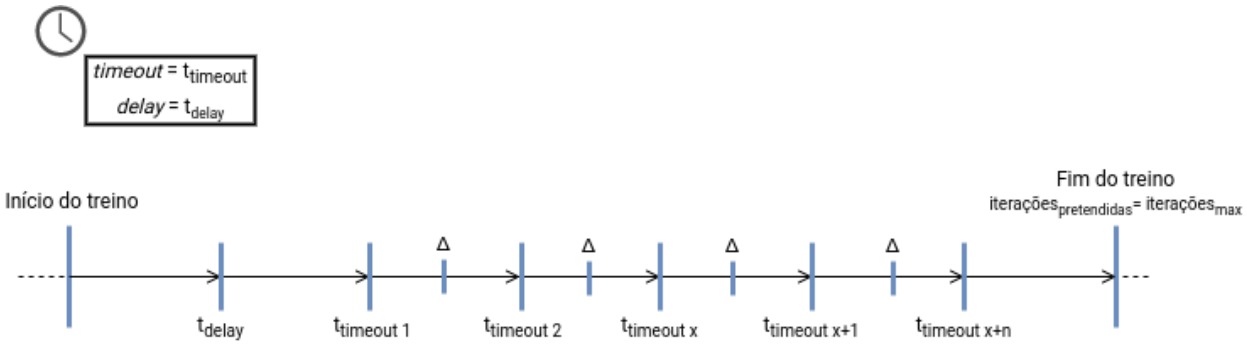


Figura 4.8: Esquema do funcionamento do agendamento de acções sobre *features*.

Conforme explicado na sub-secção 2.5, o Multi-Armed Bandit é um problema probabilístico que tenta resolver o dilema da *exploration* contra a *exploitation*. Neste caso em específico, dos Mapas Auto-Organizados Ubíquos, o problema pode ser adaptado para uma versão em que o algoritmo analisa as *features* mais relevantes de um certo conjunto de dados, utilizando as métricas como o erro médio de quantização do modelo UbiSOM, para detectar a variância e escolher quais as *features* que potenciam os maiores ganhos, i.e. reduzir o erro médio de quantização/topológico ao fim de algumas iterações, caso se desligue/ligue essa *feature*.

A ideia é alterar o contexto da existência de *slot machines* para a existência de *features*, em que ligando ou desligando uma determinada *feature*, pode ser vantajoso ou pejorativo

para os valores das métricas adjacentes ao modelo UbiSOM e podendo afectar a convergência do algoritmo. Invés de uma *slot machine* gerar uma recompensa (ou perda), este acto de ligar/desligar *features* traduz-se num ganho ou perda de erro médio de quantização ($\overline{qe}(t)$), essencialmente comprovando quais as melhores *features* que deverão perdurar (ou não) para gerar um mapa com os melhores resultados. Optou-se por escolher como valor de comparação o erro médio de quantização, contudo poderia ter sido escolhido qualquer métrica do modelo UbiSOM (e.g. erro topológico), ou mesmo actuar sobre as *features* de acordo com um número fixo de iterações.

O algoritmo escolhido para esta análise foi o ϵ-*Greedy*. De forma concisa, o algoritmo realiza exploração aleatória de acordo com um baixo valor probabilístico ϵ.

A implementação deste mecanismo é efectuada com recurso a uma *thread* agendada para executar em certos intervalos de tempo. Esta *thread* não é executada assim que o treino do mapa começa, estando associado um pequeno *delay* antes deste processo começar. Normalmente, o *delay* é o tempo médio que o algoritmo UbiSOM demora a entrar na fase de convergência (aproximadamente 3000 iterações, podendo variar para outros *datasets* e para outras configurações de hiper-parâmetros).

Para a implementação deste mecanismo ter sucesso, foi utilizado o módulo *Timer*, da biblioteca *java.util* [61]. Este módulo é utilizado para agendar a execução de *threads* no *background* da execução de uma aplicação. Efectivamente não interfere com a execução da *thread* do treino que está a acontecer num determinado instante de tempo, contudo, age sobre os mesmos objectos, em específico, actua sobre os objectos referentes ao treino de um modelo UbiSOM.

Processo de selecção de *features* A ideia geral pretendida para este mecanismo encontra-se ilustrada na figura 4.8, onde se pode ver a correlação para com o treino dos modelos. No início de um treino requisitado, ao protocolo está associado um compasso de espera, de forma a que o algoritmo UbiSOM consiga convergir nas 3000 iterações que normalmente o mesmo converge. Esta foi a forma encontrada para permitir que o algoritmo evolua e que as acções sobre *features* reflictam o real valor que uma determinada *feature* tem para o modelo resultante.

Ou seja, no exacto momento que o *delay* acaba, o protocolo ainda não começa, iniciando só no instante temporal do primeiro *timeout*. Dentro deste *timeout* e conforme representado na figura, o protocolo faz a medição do erro médio de quantização no instante que uma *feature* é ligada/desligada e faz um outro compasso de espera (iremos chamar-lhe de *timeout delta* em diante para facilitismo na percepção deste processo) para então medir a evolução do erro médio de quantização. O *timeout delta* é normalmente um segundo, representando 200 iterações (naturalmente descoberto com as experiências feitas neste projecto para as unidades de computação utilizadas) do modelo UbiSOM. Este agendamento é repetido até ao instante $t_{timeoutx+n}$. Obviamente, o protocolo não é perpetuado para além do treino do mapa, sendo o *scheduler* parado quando faltarem 100 iterações para o fim do treino. Este número foi escolhido para criar uma condição de

paragem deste protocolo e não permitir que o mesmo continue o seu processo para além do fim de um treino. Assim, o protocolo termina e retorna os valores associados a cada *feature* do conjunto de dados, nomeadamente o valor estimado, o número de vezes que uma *feature* foi escolhida e o número de vezes que uma *feature* gerou recompensa (i.e. o valor do erro médio de quantização durante o *timeout delta* baixou).

Metodologia A implementação utilizada para este processo está contida no anexo III, na listagem III.1. Dentro dessa implementação, existem algumas notações com comentários que estão relacionadas com o conteúdo presente neste paragrafo sobre a metodologia utilizada para implementação do processo.

No constructor deste processo, é associado um treino e é criada uma lista que vai albergar cada uma das *features*, inclusive o nome da *feature*, o número de vezes que foi seleccionada, valor estimado de escolha dessa *feature* e o número de vezes que essa *feature* gerou recompensa. O valor estimado da *feature* inicialmente é 1.0 para todas as *features*, considerando-se uma inicialização optimista. Quando se executa mais do que uma vez este mecanismo para o mesmo conjunto de dados, os valores considerados são os valores de execuções anteriores (explicado posteriormente em detalhe).

O valor de ϵ é que determina se o algoritmo irá fazer a exploração das *features*, ou seja, escolher uma *feature* de forma aleatória, ou se irá fazer a *exploitation*, onde irá escolher a *feature* que em determinado momento do treino irá ter o valor estimado máximo (equação 4.2). A respeito do ϵ, convém referir que está programado para ir decaindo (equação 4.1) à medida que as iterações do mapa avançam. Isto deve-se ao facto de não se pretender estar frequentemente a explorar certas *features* que não geram recompensa (i.e. das oportunidades que existam para uma *feature* ser analisada, comprova-se que não causa um decréscimo do erro médio de quantização).

$$\epsilon = \frac{1}{\#search\_actions} \tag{4.1}$$

Quando o processo efectivamente é inicializado, é então gerado um valor aleatório entre 0 e 1, que irá determinar se o algoritmo escolhe exploração ou *exploitation*. Este valor é comparado ao valor de ϵ. Quando é escolhida a exploração, é novamente gerado um número que está no intervalo da cardinalidade de *features* do *dataset* a ser treinado, para determinar qual a *feature* a ser ligada/desligada. Se for escolhida a *exploitation*, a lista que contém as *features* é analisada para determinar a *feature* com valor estimado máximo dentro do conjunto destas.

Com a *feature* escolhida, o número de acções sobre a *feature* é incrementado e é feita um compasso de espera para determinar a variação do erro de quantização. Se o erro médio de quantização baixou, significa que a *feature* gera recompensa. Após essa análise, é calculado o novo valor estimado para a *feature* escolhida, que segue a seguinte fórmula:

$$E.V.(feature) = \frac{\#reward(feature)}{\#chosen(feature)} \tag{4.2}$$

Na equação 4.2, $E.V.$ é o **valor estimado de uma *feature***, $\#reward(feature)$ é o número de vezes que a *feature* escolhida **gerou recompensa** e $\#chosen(feature)$ é o número de vezes que essa *feature* **foi escolhida**.

O algoritmo é parado quando faltar um certo valor para o fim das iterações do mapa (normalmente, 100). Esta decisão na implementação recai sobre o facto de o mecanismo necessitar de ser parado (para não executar para além das iterações requeridas num treino do modelo UbiSOM com este mecanismo activo) e porque se o número de iterações (aproximado) em que se faz a análise da variação do erro médio de quantização é normalmente 200, a 100 iterações do fim do treino do modelo UbiSOM considera-se que já não há análise das *features* para ser efectuada.

De forma a seleccionar as melhores *features*, consideram-se em primeira instância dois parâmetros:

- Uma *feature* tem de possuir um **valor estimado maior** que 0.3.
- Uma *feature* tem de ter sido **escolhida** 3 ou mais vezes. A escolha deste valor deve-se ao facto de empiricamente tratar-se de um valor admissível para servir como distinção entre *features* a escolher. Obviamente, o valor de 0 não serviria, dado que uma *feature* nunca teria sido analisada. O valor de 1 trata-se de um valor em que pode representar que uma *feature* tenha sido escolhida aleatoriamente e não significando que é a melhor para ser escolhida. O valor de 2 pode significar que exista uma situação de empate (uma escolha aleatória e uma escolha por se tratar de uma boa *feature*). Contudo, o valor 3 retira todas as dúvidas faladas anteriormente e representa então uma *feature* que pode entrar para o lote de *features* escolhidas.

Os parâmetros anteriormente especificados fazem sentido quando existe a garantia de um treino com duração grande o suficiente para existir dados suficientes em que se possa discernir quais as melhores *features* a escolher. No caso do modelo UbiSOM estar a utilizar iterações, possuir um valor alto de iterações e no caso de *streaming*, ter uma duração igualmente elevada.

Naturalmente, a escolha final é feita de acordo com o número de *features* que um *dataset* contém. No caso de das *features* escolhidas possuir um número maior do que as do *dataset* original, são escolhidas aquelas que tenham sido escolhidas mais vezes. Esta segunda parte do processo de escolha, recai apenas sobre as *features* que já tenham sido escolhidas pelos condicionalismos anteriormente descritos.

No caso de existirem mais do que um treino do modelo UbiSOM a executar em simultâneo, após o término de todos eles, os valores para as *features* de todos os treinos são agregadas, de forma a que os atributos finais das *features* (valor estimado, número de vezes que foi escolhida, número de vezes que gerou recompensa) sejam calculados.

Após ser feita a agregação dos resultados, as *features* escolhidas são apresentadas num ficheiro de texto, para além de estarem todas listadas no ficheiro *JSON* (V.2) onde estão incluídos os resultados gerais das *features* para um determinado *dataset*.

Agregação dos resultados Caso exista mais do que uma execução em simultâneo e caso se utilize um ficheiro com resultados do protocolo *Multi-Armed Bandit* como valores base para o cálculo do valor estimado das *features*, no fim da execução de todos os treinos, os resultados são agregados e reduzidos, como se tratasse de um treino apenas. Para este mecanismo ser eficaz, foi implementado um temporizador que dispara no fim dos treinos com um certo *timeout*, *timeout* esse que foi estabelecido depois de algumas execuções de treinos para perceber quanto tempo demora em média um treino seja com poucas ou um elevado número de iterações. Após a recolha dos resultados de cada um dos treinos, é feito o *parsing* por um módulo que é responsável pelos cálculos finais deste protocolo.

Para obter os resultados de um treino em específico, os resultados são subtraídos ao ficheiro base inicialmente entregue aos treinos para ser utilizado como ponto de partida. Isto serve para determinar quantas vezes foram escolhidas as *features* num determinado treino e para não forjar os resultados com números que estarão influenciados pelo ficheiro base. Após essa subtracção ser feita a cada um dos treinos instanciados, são somados os valores de todas as instâncias, incluindo o ficheiro base, para concluir então o valor estimado final de cada uma das *features* para um *dataset* em específico, sendo guardado para utilização futura.

A implementação do mecanismo de agregação dos resultados encontra-se descrita na listagem III.2, no anexo III.

4.5 Ferramentas de *debugging* e monitorização

***Logs* dos erros de topologia e quantização:** Os erros de quantização e de topologia são guardados em intervalos de 10 iterações. Para manter este *log* actualizado, os erros são guardados numa lista ao longo dos treinos e no término destes, são guardados para um ficheiro de texto convencional. Se os treinos são executados com recurso ao protocolo *Multi-Armed Bandit*, serão igualmente integrados os valores de erro nos dois intervalos de tempo em que o impacto de uma acção sobre uma *feature* será analisado. O ficheiro de *output* é um simples ficheiro de texto, organizado convenientemente para uma análise visual ser facilmente realizada. Um exemplo está representado na listagem V.1, no anexo V.

Medição do tempo de um treino: A medição do tempo de um treino é feito com dois "carimbos" de tempo, com o auxílio do método *System.getCurrentMillis()*. Existe uma variável no início do treino que regista o início deste e uma no final, que após feita a sua subtracção, é encontrado o tempo em milissegundos que um determinado treino demorou. Este valor é guardado nos *logs* acima descritos.

***Flag* para etiquetar a unidade de processamento utilizada:** Para facilidade de distinção entre unidades de processamento e os treinos em cada uma destas, aquando a inicialização destes, é integrada uma variável que contém um nome referente à unidade

de processamento utilizada. Essencialmente, é uma etiqueta e estará presente em todos os ficheiros de *output* relativos a um treino.

***Log* dos resultados do algoritmo ϵ-Greedy:** Os resultados afectos a cada uma das *features* são igualmente guardados, nomeadamente o número de vezes que uma *feature* foi escolhida, o número de vezes que gerou recompensa e o seu valor estimado. O resultado final é guardado em ficheiro, contudo, os resultados são apresentados no terminal aquando a execução dos treinos, assim como a decisão do algoritmo (se foi escolhida a *exploration* ou a *exploitation*).

No caso de haver mais do que uma execução em simultâneo, o processo que gere as instâncias faz um compasso de espera (a escolha do valor para este compasso foi analisado na secção anterior) para todas os treinos terminarem e agrega esses resultados. Os resultados são depois reduzidos a uma lista simples contendo todas as *features*, com os valores estimados calculados a partir do âmbito geral. Estes valores são depois guardados num *JSON*, para futuras leituras em futuros treinos. A decisão de utilizar um ficheiro *JSON* remete-se ao facto de ser mais uma fase de *debug* aos resultados, em que se possa visualizar os mesmos, invés de se utilizar um ficheiro binário, cujo *output* não é visível sem o software estar em *runtime*. Um exemplo de um *output* deste género está ilustrado no apêndice V.2.

5

Testes, Avaliação e Validação

Um dos objectivos da aplicação desenvolvida para este projecto é maximizar a performance e escalabilidade do sistema, de forma que o número máximo de mapas seja calculado num determinado instante de tempo e assim possibilitar a pesquisa pelas melhores parametrizações para um certo conjunto de dados. Aliado a este objectivo está um outro objectivo focado na pesquisa pelas *features* mais relevantes para um certo conjunto de dados, podendo assim ser feita uma selecção das *features* mais relevantes (i.e. que produzem uma maior redução nas métricas de erro do modelo UbiSOM), minimizando o trabalho de análise em conjunto de dados onde o número de *features* é elevado.

Este capítulo irá conter a descrição das experiências utilizadas para comprovar a eficácia do sistema proposto, assim como os seus respectivos resultados e discussão sobre os mesmos e ainda detalhes globais como os conjuntos de dados utilizados para treinar os modelos UbiSOM, as unidades de computação utilizadas em cada uma das experiências, etc.

Para então testar e validar o sistema proposto para a utilização do algoritmo UbiSOM com vários treinos em paralelo com várias parametrizações sobre o mesmo *dataset*, foi decidido separar as experiências em três modalidades distintas: Aprendizagem Automática, Performance e Qualidade do Software. Nas secções subsequentes irá ser feita uma descrição breve sobre as experiências levadas a cabo para validar essas modalidades e no fundo, validar o sistema proposto. As experiências de análise dos vários conceitos de Aprendizagem Automática (qualidade dos resultados dos treinos do modelo UbiSOM, selecção de *features* e hiper-parametrizações) estão na secção 5.2, as experiências de análise à performance estão na secção 5.3 e detalhes sobre a Qualidade do *Software* estão na secção 5.4.

5.1 Caracterização dos Dados e Metodologia de Testes

Esta secção irá conter detalhes gerais sobre a metodologia utilizada para testar e avaliar o sistema proposto. Serão discutidos as parametrizações a utilizar para o algoritmo UbiSOM, os conjuntos de dados a utilizar e as unidades de computação que serviram para

treinar os vários modelos.

5.1.1 Parametrizações

As parametrizações escolhidas para as experiências apresentadas nesta secção são aleatórias, dado que é o objectivo geral encontrar as melhores parametrizações para um determinado conjunto de dados. Para tal, é necessário apenas respeitar os intervalos ótimos determinados pelo trabalho de Bruno Silva [72]. Efectivamente, o parâmetro β está concentrado nos valores $[0,6;0.9]$ e o parâmetro T está entre os valores $[1000;2500]$.

Contudo, embora maior parte das experiências estejam com estes valores concentrados nos intervalos acima descritos, existe uma experiência onde se testa a otimalidade dos parâmetros η e σ. Na maioria das vezes, é utilizado para o η a variação entre 0.1 e 0.08 e para o σ a variação entre 0.6 e 0.2. Nas experiências que se pretende encontrar as melhores combinações destes valores, irá ser gerado valores entre 0.0 e 1.0 para os dois parâmetros. Essencialmente, esta variação de valores em η e σ será induzida para testar a robustez destes parâmetros, de forma a analisar o seu impacto nos resultados obtidos (ou seja, se o erro topológico ou o erro de quantização sobe para valores muito acima dos anteriormente determinados) e no atraso da convergência do algoritmo UbiSOM (situada frequentemente às 3000 iterações para alguns conjunto de dados).

Convém salientar que todas as experiências serão realizadas com as dimensões dos mapas fixas em 20x40, contendo 800 observações no mapa em cada uma das iterações do algoritmo UbiSOM.

5.1.2 Conjuntos de Dados

Os conjuntos de dados utilizados para realizar as experiências foram os seguintes: *Iris*, *Hepta*, *Clouds*, *Complex*, *Chain*.

Nesta dissertação, os *datasets* vão ser treinados com o modelo UbiSOM para os resultados serem comparados àqueles obtidos por Bruno Silva em [72]. Sobretudo, servirá para analisar quais as melhores parametrizações para cada um dos *datasets* e de certa forma, comprovar que as parametrizações já estabelecidas são as melhores para este conjunto de dados em específico ou se outras encaixam melhor aquando do uso do modelo UbiSOM.

De seguida, vão ser elaboradas as descrições sobre cada um dos conjuntos de dados e sobre as suas características, assim como as variações feitas para estarem adequados às experiências pretendidas para testar o sistema proposto.

Iris O *dataset Iris* é um conjunto de dados que representa amostras de várias flores da espécie Iris. O *dataset* possuí 4 *features*, nomeadamente as dimensões da sépala e da pétala de cada uma dessas flores. É um *dataset* frequentemente utilizado em algoritmos de *data mining*, *clustering*, entre outros, servindo como um bom exemplo para testar o UbiSOM.

As suas 4 *features* descrevem o seguinte:

- Comprimento da sépala em centímetros
- Largura da sépala em centímetros
- Comprimento da pétala em centímetros
- Largura da pétala em centímetros

O *dataset* é algo reduzido, apenas contendo 150 observações. Apesar disso, é um conjunto de dados que já foi testado em inúmeros algoritmos de Aprendizagem Automática, servindo então como um bom ponto de partida.

Para as experiências que irão ser relatadas de seguida, o *dataset* sofreu algumas modificações, existindo outras versões do mesmo *dataset* mas com o acrescento de algumas colunas. Enuncia-se então algumas das variações utilizadas no âmbito das experiências neste projecto:

- ***Dataset* regular:** Conjunto de dados Iris na sua forma natural. Para mais facilidade na utilização deste *dataset* no *software* MultiSOM, os dados foram pré-normalizados por *rescaling* (normalização $min - max$) e arredondados para o intervalo $[0;1000]$ [48].
- ***Dataset* com 20, 40 e 80 *features* aleatórias:** Realizou-se um acrescento de *features* com valores aleatórios para este *dataset* ser utilizado em testes de *stress* relativamente ao *feature switching*. As colunas acrescentadas foram geradas com a função *rand()* do *software* de folhas de cálculo LibreOffice Calc. Os valores utilizados encontram-se no intervalo $[0;100]$. Contudo, as colunas originais têm um intervalo mais abrangente, mas para simplificação do processo, decidiu-se apenas gerar valores para as colunas no intervalo anteriormente descrito.
- ***Dataset* com 20% de ruído nas colunas base:** Para além de um *dataset* com colunas aleatórias (colunas chamadas *artificial1* e *artificial2*), foi também criado um *dataset* com colunas que são duplicadas das colunas base (colunas chamadas *duplicate1*, *duplicate2*, *duplicate3* e *duplicate4*) só que contém uma percentagem de ruído. Para obter estas colunas, duplicou-se as colunas base e multiplicou-se o factor $0,20$ para obter colunas com ruído. Este *dataset* também contém colunas aleatórias, no entanto sendo apenas duas. Este *dataset* servirá essencialmente para testar o mecanismo de selecção de *features*.

A listagem anterior serve essencialmente para esclarecer as menções sobre este *dataset* nas secções posteriores.

Complex O *dataset* Complex é um *dataset* constituído por 100.000 pontos bi-dimensionais (i.e., constituído por duas *features*) que descrevem uma estrutura de *clusters* complexa, com sete *cluster* distintos, tendo sido utilizado originalmente para validar se esses *clusters* eram detectáveis pelo uso da Unified Distance Matrix [9, 72].

Clouds O conjunto de dados *Clouds* é constituído por 200.000 pontos bi-dimensionais (i.e., constituído por duas *features*) que descrevem uma mudança gradual da estrutura de três *clusters* gaussianos. Este conjunto de dados foi utilizado inicialmente para analisar como os diferentes algoritmos do modelo SOM (UbiSOM incluído) reagem a mudanças graduais na distribuição subjacente [9, 72], sendo portanto um conjunto de dados com uma distribuição não-estacionária.

Chain O conjunto de dados *Chain* é constutuído por 100.000 pontos tri-dimensionais (i.e., constutído por três *features*) que representam um par de anéis interligados [9, 72]. Originalmente, este *dataset* foi utilizado para validar se os agregamentos eram detectáveis pelo uso da Unified Distance Matrix.

Hepta O conjunto de dados *Hepta* é constituído por 150.000 pontos tri-dimensionais (i.e., possuí três *features*) que descrevem uma estrutura de *clusters*. Dos sete *clusters* Gaussianos iniciais, um deles desaparece durante o treino do algoritmo UbiSOM. Inicialmente, este *dataset* tinha o propósito de avaliar como os diferentes algoritmos baseados no modelo SOM reagem a mudanças repentinas na distribuição subjacente e para validar a concepção da métrica de utilidade média de um neurónio (ou protótipo).

5.1.3 Unidades de Computação

Para realizar as experiências referentes a este projecto, foram utilizadas várias unidades de computação com variadas características. A utilização de variadas unidades de computação foi sobretudo para testar os múltiplos treinos do modelo UbiSOM em paralelo e analisar como é a reacção em utilizar máquinas com um maior número de *threads*.

A tabela 5.1 descreve cada uma dessas unidades de computação, contendo o número de núcleos, de *threads*, a velocidade de *clock* do processador em questão, o nome de cada um dos processadores, entre outros factores. A existência da atribuição de uma "etiqueta" a cada um dos processadores serve para distinguir entre processadores utilizados em ambiente local e ambiente de *cluster*.

Nome U.C.	**CPU**	***Clock***	***Cores***	***Threads***	**RAM**	**HDD/SSD**
localhost	Intel i7-7700HQ	3.8 GHz	4	8	8GB	250GB
node-19	2xAMD Opteron 2376	2.3 GHz	8	8	16GB	150GB
node-17	2xIntel Xeon E5-2609 v4	1.7 GHz	16	16	32GB	110GB
node-6,7	2xIntel Xeon E5-2620 v2	2.10 GHz	12	24	64GB	100GB
node15	Intel Xeon E5-2620 v2	2.10 GHz	6	12	32GB	110GB

Tabela 5.1: Unidades de computação utilizadas nas experiências e as suas características. Nome U.C. é o nome da unidade de computação, CPU é o processador e o seu fabricante, o *clock* é a velocidade do *clock*, os *cores* é o número de núcleos do processador, *threads* é o numero de *threads* por processador, RAM é a capacidade de memória e HDD é a capacidade do disco da máquina em questão.

Excepto a máquina *localhost*, todas as outras estavam localizadas no *cluster* do Departamento de Informática da Faculdade de Ciências e Tecnologia, da Universidade Nova de Lisboa. Algumas das unidades de computação são relativamente iguais em termos do processador a ser utilizado (caso do nó 6,7 e do nó 15), contudo, diferem um pouco na capacidade de RAM e de disco e mesmo na forma como os recursos estavam alocados aquando da sua utilização.

O objectivo foi sempre encontrar máquinas com relativamente diferentes números de *cores* e *threads* para testar o algoritmo UbiSOM de forma paralela.

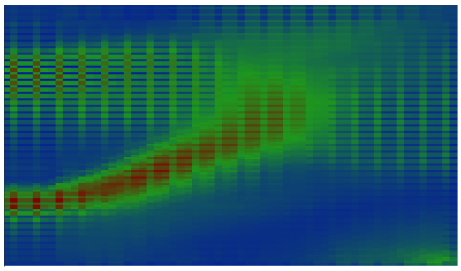
5.2 Aprendizagem Automática

5.2.1 Qualidade dos Resultados dos *Datasets*

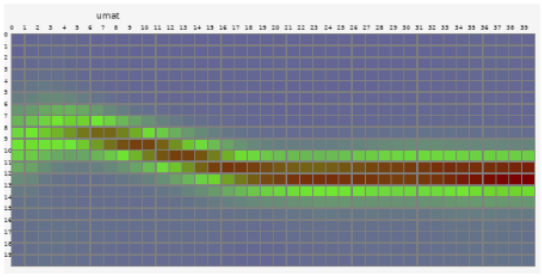
O objectivo desta dissertação é tentar obter as melhores parametrizações relativamente a certos conjuntos de dados, tendo estes conjuntos de dados sido estudados em trabalhos anteriores de João Borrego e de Bruno Silva [72, 9]. Como parametrizações diferentes são estudadas, é necessário testar as Unified Distance Matrix obtidas pelos treinos do modelo UbiSOM feitos pela aplicação MultiSOM e compará-las aos resultados anteriores.

Uma particularidade que é necessário apontar é o facto de que as Unified Distance Matrix obtidas pelos treinos do modelo UbiSOM, podem não aparentar semelhanças numa primeira observação. Isso deve-se ao facto das Unified Distance Matrix geradas, para um mesmo conjunto de dados, podem sofrer transformações nas imagens de treino para treino como rotações ou simetrias. Outra das particularidades é o facto de as experiências feitas ao longo deste capítulo estão limitadas às 19000 iterações, dado que a versão utilizada para fazer os treinos dos modelos UbiSOM estar truncada nesse número de iterações.

Clouds Conforme se pode observar nas figuras 5.1a e 5.1b, as Unified Distance Matrix resultantes do treino do modelo UbiSOM com o *dataset* Clouds são semelhantes. Aparentemente, ainda contêm uma zona verde que se verifica em 5.1a mas não em 5.1b. Esse factor atribui-se ao facto de não estarem tantas iterações associadas ao treino da figura 5.1a como na figura 5.1b.



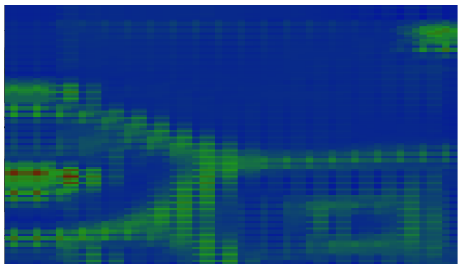
(a) Unified Distance Matrix resultante de um treino do modelo UbiSOM utilizando o conjunto de dados Clouds com 19000 iterações.



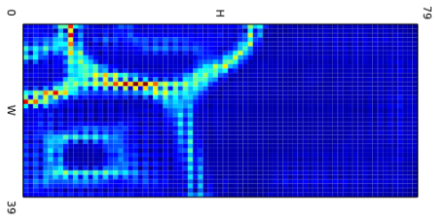
(b) Unified Distance Matrix resultante de um treino do modelo UbiSOM utilizando o conjunto de dados Clouds, obtida no âmbito da dissertação do João Borrego [9].

Figura 5.1: Unified Distance Matrix obtida fazendo o treino do modelo UbiSOM com o conjunto de dados Clouds.

Complex Conforme se pode observar nas figuras 5.2a e 5.2b, as Unified Distance Matrix resultantes do treino do modelo UbiSOM com o *dataset* Complex são semelhantes. Aparentemente, o mapa resultante sofre de uma rotação e de aplicação de simetria, com origem no software utilizado para gerar a Unified Distance Matrix.



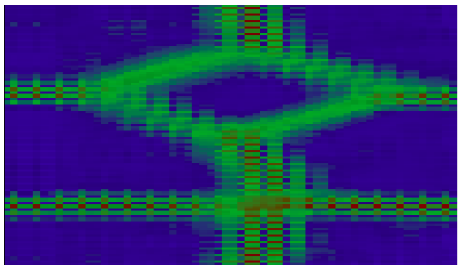
(a) Unified Distance Matrix resultante de um treino do modelo UbiSOM utilizando o conjunto de dados Complex com 19000 iterações.



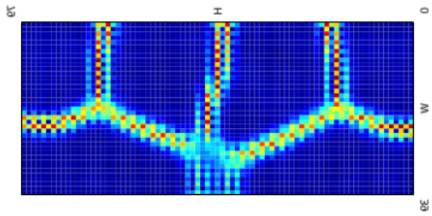
(b) Unified Distance Matrix resultante de um treino do modelo UbiSOM utilizando o conjunto de dados Complex, obtida no âmbito da dissertação de Bruno Silva [72].

Figura 5.2: Unified Distance Matrix obtida fazendo o treino do modelo UbiSOM com o conjunto de dados Complex.

Hepta Conforme se pode observar nas figuras 5.3a e 5.3b, as Unified Distance Matrix resultantes do treino do modelo UbiSOM com o *dataset* Hepta são semelhantes. Aparentemente, o mapa resultante sofre de uma rotação e de aplicação de simetria, com origem no software utilizado para gerar a Unified Distance Matrix. Apesar disso, o mapa resultante de 5.3a não está estritamente igual ao mapa de 5.3b, devendo-se a isso o facto do mapa em 5.3a estar apenas com 19000 iterações, enquanto que a outra figura é representante de um treino com 150000 iterações.



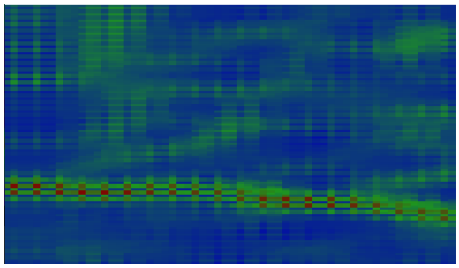
(a) Unified Distance Matrix resultante de um treino do modelo UbiSOM utilizando o conjunto de dados Hepta com 19000 iterações.



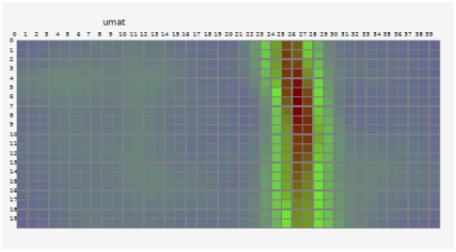
(b) Unified Distance Matrix resultante de um treino do modelo UbiSOM utilizando o conjunto de dados Hepta, obtida no âmbito da dissertação de Bruno Silva [72].

Figura 5.3: Unified Distance Matrix obtida fazendo o treino do modelo UbiSOM com o conjunto de dados Hepta.

Iris Conforme se pode observar nas figuras 5.4a e 5.4b, as Unified Distance Matrix resultantes do treino do modelo UbiSOM com o *dataset* Iris são semelhantes. Aparentemente, o mapa resultante sofre de uma rotação, mas que não parece perturbar a conclusão que o algoritmo UbiSOM continua a funcionar correctamente utilizando este *dataset*, verificando-se até as mesmas zonas verdes nos dois mapas resultantes.



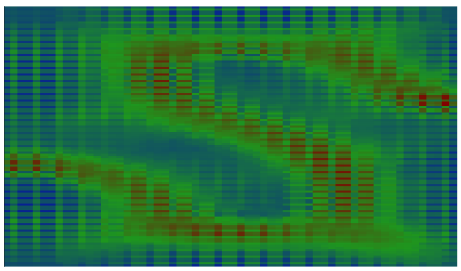
(a) Unified Distance Matrix resultante de um treino do modelo UbiSOM utilizando o conjunto de dados Iris com 19000 iterações.



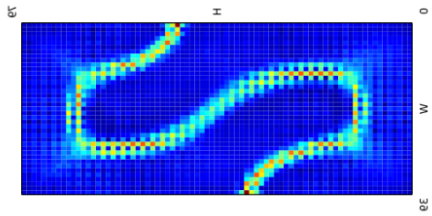
(b) Unified Distance Matrix resultante de um treino do modelo UbiSOM utilizando o conjunto de dados Iris, obtida no âmbito da dissertação do João Borrego [9].

Figura 5.4: Unified Distance Matrix obtida fazendo o treino do modelo UbiSOM com o conjunto de dados Iris.

Chain Conforme se pode observar nas figuras 5.5a e 5.5b, as Unified Distance Matrix resultantes do treino do modelo UbiSOM com o *dataset* Chain são bastante similares, notando poucas diferenças entre os mapas resultantes, embora que em 5.5a se utilize um número reduzido de iterações para o treino do modelo UbiSOM.



(a) Unified Distance Matrix resultante de um treino do modelo UbiSOM utilizando o conjunto de dados Chain com 19000 iterações.



(b) Unified Distance Matrix resultante de um treino do modelo UbiSOM utilizando o conjunto de dados Chain, obtida no âmbito da dissertação do João Borrego [9].

Figura 5.5: Unified Distance Matrix obtida fazendo o treino do modelo UbiSOM com o conjunto de dados Chain.

Observações Gerais/Análise Após análise individual a cada mapa resultante dos treinos do modelo UbiSOM conclui-se que a versão do algoritmo utilizada nesta dissertação continua a funcionar de forma correcta. A única desvantagem tratou-se de não se ter conseguido obter Unified Distance Matrix estritamente iguais, dado que a versão do algoritmo UbiSOM utilizada nesta dissertação estava truncada às 19000, como tem sido declarado ao longo deste documento e não foi possível executar os treinos do modelo UbiSOM utilizando o GPU.

5.2.2 *Feature Selection*

De forma poder perceber quais as melhores *features* de um certo *dataset*, ou seja, aquelas que mais relevância para a obtenção de melhores resultados ao nível do erro de quantização e erro de topologia do algoritmo UbiSOM, foram feitas experiências utilizando o protocolo *Multi-Armed Bandit*. Para tal, utilizou-se uma variação do algoritmo ϵ-Greedy. Este algoritmo analisa a variação do erro de quantização aquando de uma acção sobre uma *feature*, i.e. ligar ou desligar essa mesma *feature* para uma determinada execução.

Este algoritmo irá ser testado com o *dataset* Iris, utilizando uma versão modificada do mesmo, onde colunas com ruído serão inseridas (colunas duplicadas das colunas base, onde os seus valores são multiplicados por uma percentagem, neste caso, 20%) e por colunas artificiais com valores aleatórios. Com esta experiência, pretende-se obter as *features* mais relevantes para a convergência do algoritmo e minimização do erro de quantização e erro topológico.

Os treinos do algoritmo UbiSOM para esta experiência terão os seguintes parâmetros: $\beta = 0.7$, $\sigma_i = 0.6$, $\sigma_f = 0.2$, $\eta_i = 0.1$, $\eta_f = 0.08$ e $T = 2000$. Os valores escolhidos para a parametrização são os valores ótimos para o treino do *dataset* Iris, utilizando o algoritmo UbiSOM. Para o número de iterações, foram escolhidas as 19000, novamente, havendo uma restrição do número de iterações devido à limitação imposta no módulo UbiSOM. Para obter resultados experimentalmente mais sólidos, os treinos do modelo UbiSOM serão repetidos dez vezes em série, utilizando apenas uma unidade de computação, de

forma que a exploração das *features* seja consistente e sejam consideradas todas as *features* presentes no *dataset* Iris artificial. Após a primeira execução terminar, o ficheiro *JSON* dos resultados das *features* é guardado e re-utilizado/actualizado com o decorrer das outras execuções.

Os resultados obtidos para cada uma das *features* ao longo dos dez treinos estão ilustrados na tabela 5.2. Relembrando as condições para escolher uma *feature*: uma *feature* no fim da execução de um treino do modelo UbiSOM terá de ter valor estimado igual ou acima de 0.3 e terá de ter sido escolhida para análise pelo menos 3 vezes (secção 4.4). Assim sendo, as *features* escolhidas aquelas assinaladas a negrito na tabela 5.2.

Conforme referido na secção 4.4, o mecanismo necessita da definição de *timeouts* para desta forma irem desligando/ligando *features* dentro de intervalos admissíveis, para a variação do erro médio de quantização poder ser analisada convenientemente. Como o tempo de execução de um treino do modelo UbiSOM com 19000 iterações é em média 10 segundos (utilizando a unidade de computação *localhost*), decidiu-se definir o *timeout* como 1 segundo. Para além disso, foi feita a definição do *timeout delay* inicial do *scheduler* do mecanismo de selecção de *features* como 500 milissegundos, para efectivamente fazer um compasso de espera até o treino do modelo UbiSOM entrar em convergência (por volta das 3000 iterações). Desta forma, o mapa converge e não é feita a análise da variação do erro médio de quantização precocemente.

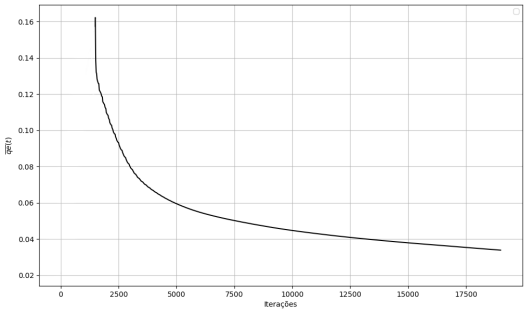
Para comparação de resultados, após os dez treinos do modelo UbiSOM com o conjunto de dados do *Iris* artificial com dez *features* com o mecanismo de selecção de *features* activo, é feito um treino apenas com as *features* vencedoras (contendo quatro *features*), de forma a analisar posteriormente a variação do erro topológico, a variação do erro de quantização médio e as Unified Distance Matrix resultantes. Estas três análises serão entre o *dataset* Iris original e o *dataset* com as *features* vencedoras. Esta comparação servirá para analisar se as *features* seleccionadas foram admissíveis ou não.

Nas figuras 5.6a e 5.6b pode ser observado a evolução do erro de quantização ao longo das iterações, para o *dataset* Iris "regular"e para a sua versão com as *features* seleccionadas pelo protocolo *Multi-Armed Bandit*. O valor final do erro médio de quantização do treino do algoritmo UbiSOM utilizando o *dataset* Iris "regular"foi 3.38% e do *dataset* com as *features* seleccionadas foi 2.2%. O valor do erro médio de um *dataset* para outro baixou, revelando que o *dataset* com as *features* que foram seleccionadas pelo mecanismo de selecção de *features* atingiu a convergência ao mesmo tempo que o *dataset* Iris original e que as melhores *features* possíveis foram seleccionadas.

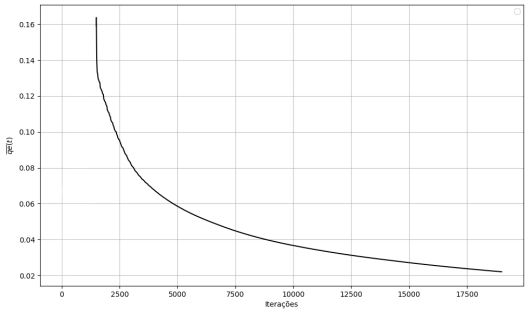
Nas figuras 5.7a e 5.7b pode ser observado a evolução do erro de topologia ao longo das iterações, para o *dataset* Iris "regular"e para a sua versão com as *features* seleccionadas pelo protocolo *Multi-Armed Bandit*. O erro de topologia para o *dataset* Iris "regular", no fim das iterações tinha o valor de 7.2% e no *dataset* Iris com *features* seleccionadas tinha o valor

Feature	Valor estimado	Acção escolhida	Recompensa
sepalLength	**0.5555555555555556**	**9**	**5**
sepalWidth	**0.5675675675675675**	**37**	**21**
petalLength	0.3333333333333333	6	2
petalWidth	**0.6071428571428571**	**28**	**17**
artificial1	0.5	6	3
artificial2	0.4444444444444444	9	4
duplicate1	0.2	5	1
duplicate2	**0.5833333333333333**	**48**	**28**
duplicate3	0.0	11	0
duplicate4	0.4	5	2

Tabela 5.2: Valores relativos à execução de dez treinos do UbiSOM com o protocolo *Multi-Armed Bandit* activo. A coluna "acção escolhida"representa o número de vezes que uma certa *feature* fora escolhida e a coluna "recompensa"representa o número de vezes que uma *feature* teve uma evolução favorável no erro de quantização entre dois instantes de tempo.

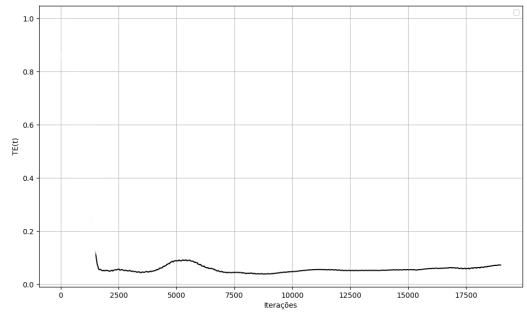


(a) Erro de quantização de um treino do algoritmo UbiSOM utilizando o *dataset* Iris "regular".

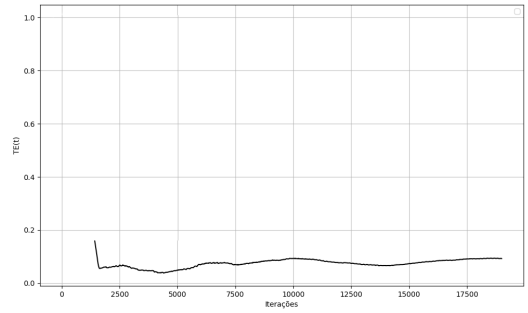


(b) Erro de quantização de um treino do algoritmo UbiSOM utilizando o *dataset* com as *features* seleccionadas pelo protocolo MAB.

Figura 5.6: Erros médios de quantização.



(a) Erro topológico de um treino do algoritmo UbiSOM utilizando o *dataset* Iris "regular".



(b) Erro topológico de um treino do algoritmo UbiSOM utilizando o *dataset* com as *features* seleccionadas pelo protocolo MAB.

Figura 5.7: Erros topológicos.

de 9.3%. O erro topológico do algoritmo UbiSOM é uma métrica que mede distorções do mapa resultante e preferencialmente, o valor do erro topológico é preferível que atinja valores mais próximos de zero. Neste caso, o erro topológico do *dataset* com as *features* seleccionadas pelo mecanismo de selecção de *features* subiu ligeiramente em relação ao erro topológico do *dataset* Iris original. A subida de valor indica maior distorção do mapa resultante treinando com o conjunto de dados com *features* seleccionadas. Isto deve-se ao facto da *feature duplicate2* estar no lote das *features* seleccionadas. Sendo uma *feature* artificial, não deveria pertencer a esse lote.

Antes de dar como terminado a descrição dos resultados da experiência afectiva ao *Feature Selection*, ainda é importante analisar os resultados das Unified Distance Matrix relativas ao *dataset* com as *features* seleccionadas pelo mecanismo Multi-Armed Bandit Protocol (figura 5.9), o *dataset* Iris original (figura 5.8) e o *dataset* utilizado para descobrir então as *features* mais relevantes (figura 5.10), de forma a avaliar a qualidade dos mapas resultantes. Como se pode observar na figura 5.9, existem bastantes semelhanças com a Unified Distance Matrix obtida pelo treino com o *dataset* Iris original, confirmando que efectivamente as melhores *features* possíveis foram seleccionadas.

Ainda assim, avaliando entre o mapa 5.10 e o mapa 5.9, observa-se uma melhor uniformização dos protótipos, tendo havido uma maior diversificação de cores no mapa, sendo relativamente mais simples identificar os vários *clusters* resultantes do treino do modelo UbiSOM.

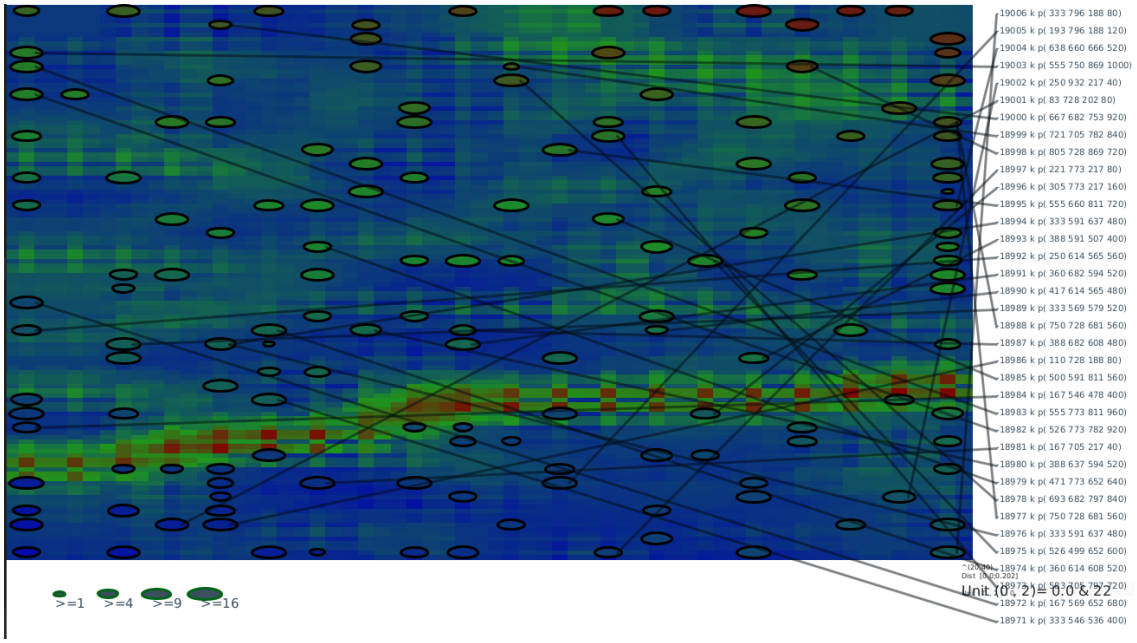


Figura 5.8: Unified Distance Matrix resultante de um treino do algoritmo UbiSOM utilizando o *dataset* Iris original.

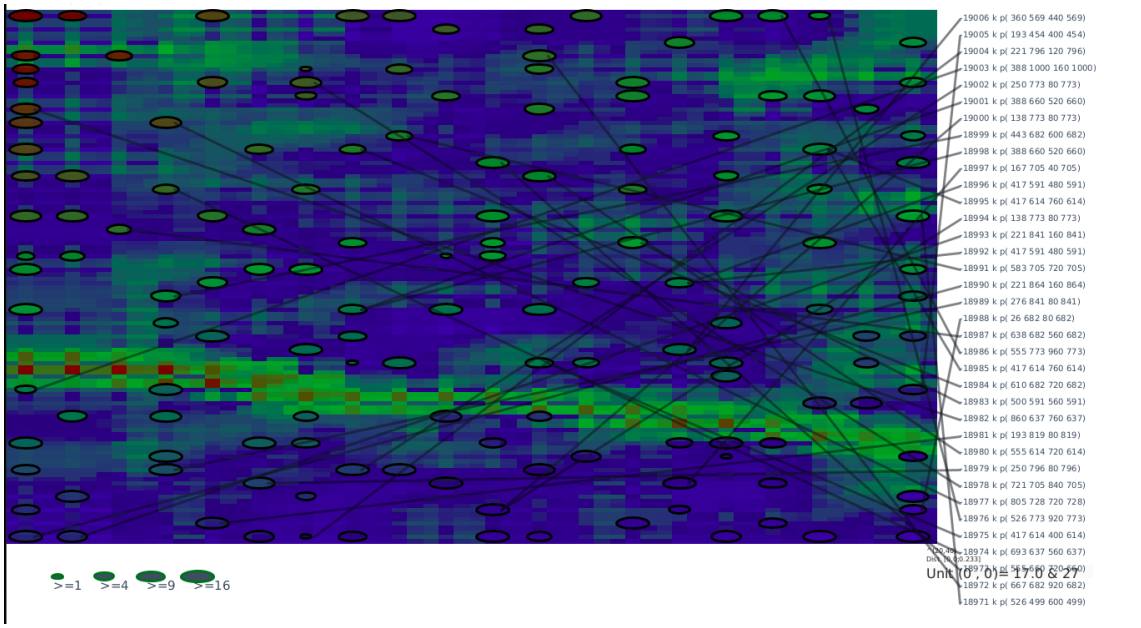


Figura 5.9: Unified Distance Matrix resultante de um treino do algoritmo UbiSOM utilizando o *dataset* com as quatro *features* seleccionadas pelo protocolo Multi-Armed Bandit Protocol.

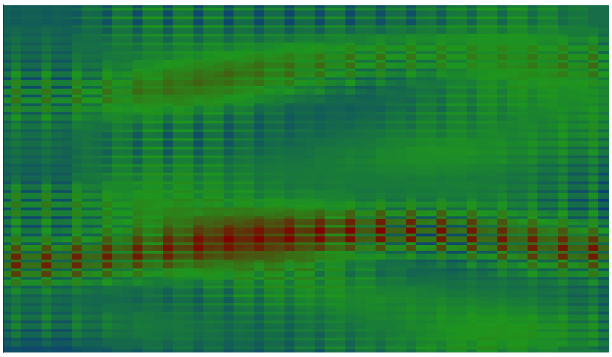


Figura 5.10: Unified Distance Matrix resultante de um treino do algoritmo UbiSOM utilizando o *dataset* Iris artificial com dez *features*, sem o mecanismo Multi-Armed Bandit Protocol ligado.

Observações Globais/Análise Em suma, os resultados aplicando o algoritmo $\epsilon - Greedy$ para um mecanismo de selecção de *features* foram favoráveis. Praticamente todas as *features* do *dataset* Iris original foram seleccionadas, exceptuando a *feature petalLength*. Em contrapartida, a *feature duplicate2* foi seleccionada, o que fez com que o erro topológico aumentasse ligeiramente. Com este aumento, revela que o mapa resultante de um treino do modelo UbiSOM com as *features* seleccionadas está um pouco mais distorcido que um treino do modelo UbiSOM com o *dataset* Iris original, mas não invalida a positividade dos resultados, sendo o lote de *features* obtido considerado favorável. Ainda assim, utilizar o algoritmo $\epsilon - Greedy$ induz alguma aleatoriedade na escolha de *features*, havendo fases em que são seleccionadas *features* que a certa altura sabe-se que não são as melhores que se podem escolher.

5.2.3 Hiper-Parametrizações

Foram feitas experiências para serem comparadas com os resultados obtidos por Bruno Silva em [72], especialmente feitos para tirar conclusões relativamente à análise de sensibilidade, ou seja, quais os melhores parâmetros para um certo de conjunto de dados. Para além dos treinos em massa que foram realizados, foi também feita uma análise aos parâmetros que Silva estabeleceu como fixos, sendo eles a taxa de aprendizagem inicial e final (η_i e η_f) e o raio da vizinhança normalizado inicial e final (σ_i e σ_f).

Inicialmente, irão ser utilizadas alguns dos resultados obtidos anteriormente na análise de performance e serão comparados os erros topológicos e de quantização, de forma a que se obtenha um consenso no facto de quais as melhores parametrizações a utilizar para um certo conjunto de dados. Nesta experiência inicial, o β e o T serão os únicos parâmetros variáveis, ainda que estejam a variar dentro do intervalo admissível ($\beta \in [0.6; 0.9]$ e $T \in [1000; 2500]$) estabelecido por Silva.

Posteriormente, os parâmetros σ e η vão sofrer a mesma variação, contudo, num intervalo de $[0; 1]$. Esta experiência centra-se em testar então se os parâmetros são mais rentáveis de deixar fixos em todos os treinos, utilizando qualquer conjunto de dados ou se a variação implica melhores resultados.

Com estas duas experiências espera-se reavaliar a parametrização estabelecida por Silva e se efectivamente, se for necessário variar alguns dos parâmetros, identificar quais são os mais rentáveis de impingir essa variação e em que conjuntos de dados deve ser aplicada.

5.2.3.1 Parâmetros η e σ variáveis

De forma a analisar se existem melhores parametrizações a nível do raio da vizinhança normalizado e da taxa de aprendizagem, foram realizadas 5 execuções para cada um dos *datasets* previamente enunciados, em que estes parâmetros tomam valores aleatórios no intervalo $[0.0; 1.0]$, treinando os modelos com cerca de 19000 iterações. Para além disso, os parâmetros β e T estão igualmente sujeitos a aleatoriedade. Anteriormente a esta experiência, testou-se o algoritmo UbiSOM com os parâmetros β e T fixos nos parâmetros admissíveis determinados por Bruno Silva, contudo, percebeu-se que utilizar estes parâmetros fixos (valores de β e T para cada um dos *datasets* referidos nas legendas das figuras 5.11, 5.13, 5.15, 5.17) ou não, gera os mesmos resultados àqueles obtidos nesta experiência.

Convém referir que a geração dos parâmetros η e σ não tem um detalhe de implementação associado que é a garantia que o valor final de η seja menor que o valor inicial para que a convergência do algoritmo seja atingida. Por este motivo, nas análises seguintes, alguns dos modelos não têm esse decrescimento nos valores de η e irão ser descartados à partida, mesmo que possuam melhores valores de $\overline{qe}(t)$ que os restantes.

Para facilidade de interpretação, o erro de quantização médio ao longo desta (e da próxima sub-secção) irá ser identificado pelo seu símbolo matemático: $\overline{qe}(t)$.

Estas experiências foram realizadas na unidade de computação *localhost*, tendo sido os treinos executados simultaneamente, pelo menos para cada conjunto de dados.

De seguida, apresentam-se então os resultados para cada um dos conjuntos de dados, com uma breve descrição dos resultados e discussão dos mesmos.

Clouds No gráfico 5.12 está representado os resultados dos treinos com os parâmetros σ e η gerados aleatoriamente, estando no gráfico 5.11 representados os resultados obtidos por Bruno Silva em [72]. O gráfico que contém os resultados tem ainda um pequeno quadro com os parâmetros de cada uma das funções descritas no mesmo, assim como a respectiva cor.

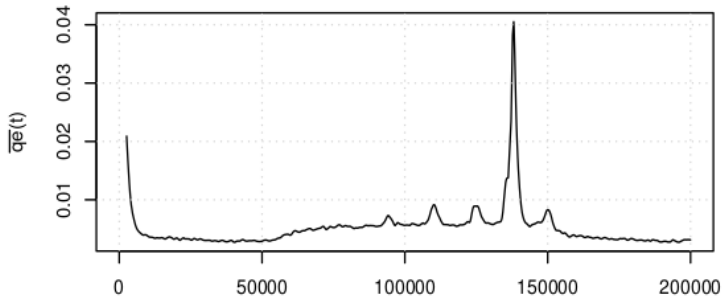


Figura 5.11: Resultados de Bruno Silva em [72] do erro médio de quantização ($\overline{qe}(t)$), utilizando a parametrização $\eta = 0.1, 0.08$ e $\sigma = 0.6, 0.2$ para o *dataset* Clouds.

Os modelos treinados ao abrigo desta experiência tiveram valores de erro de quantização médio ($\overline{qe}(t)$) compreendidos entre o valor máximo de 3% e valores menores que 1%. Existiram dois tipos de resultados face aos parâmetros utilizados: modelos de que resultaram valores de $\overline{qe}(t)$ que a partir da convergência do algoritmo UbiSOM tornaram-se constantes e modelos onde os valores de $\overline{qe}(t)$ oscilavam entre um valor máximo e um valor mínimo ao longo das iterações. Na tabela 5.3 pode ser observado os parâmetros para cada um dos treinos apresentados no gráfico 5.12, onde está apresentado o valor inicial e final para cada um dos parâmetros η e σ e a diferença entre estes.

σ_i	σ_f	η_i	η_f	Diferencial σ	Diferencial η	$\overline{qe}(t)$
0.51	0.13	0.58	0.39	-0.38	-0.19	3.09%
0.06	0.07	0.3	0.84	+0.54	+0.01	0.18%
0.71	0.57	0.33	0.25	-0.08	-0.14	0.47%
0.14	0.39	0.46	0.52	+0.25	+0.06	1.64%
0.01	0.59	0.58	0.5	+0.58	-0.08	0.44%

Tabela 5.3: Parâmetros η e σ utilizados nos treinos do modelo UbiSOM para o *dataset* Clouds, seguindo respectivamente a ordem em que aparecem no gráfico 5.12.

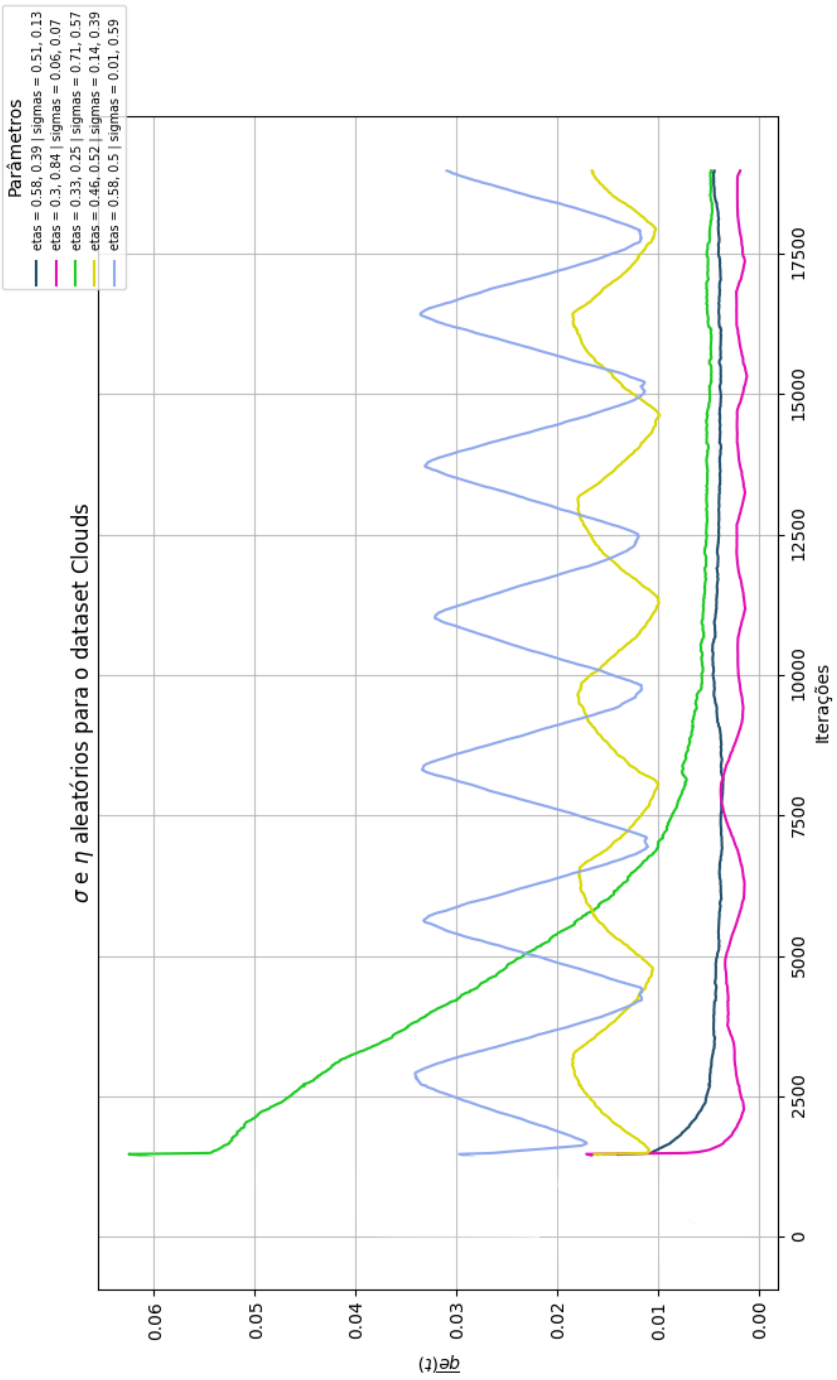


Figura 5.12: Resultados relativos a modelos UbiSOM treinados com parâmetros η e σ aleatórios utilizando o *dataset* Clouds.

Dito isto, fazendo uma análise comparativa destes resultados com o resultado de Bruno Silva [72], dos cinco modelos treinados, apenas dois tiveram um desempenho positivo. Os parâmetros ótimos presentes na tese de Bruno Silva geraram um gráfico similar àquele obtido nesses modelos de treino, comparando à evolução dos valores ao longo das iterações. Apesar no modelo representado pela cor lilás ter tido um erro médio de quantização de 0.18%, como a diferença entre os valores de η é positiva, o modelo é descartado, dado que não é garantido que o algoritmo UbiSOM atinja a convergência utilizando os parâmetros especificados na tabela.

Assim sendo, comprova-se que para o conjunto de dados *Clouds*, o η terá de ter uma diferença entre valores iniciais e finais ($0.14 \leq \eta_f - \eta_i \leq 0.19$). O σ não poderá ter uma diferença entre valores iniciais e finais maior que 0.08, dado que parâmetros acima do valor anteriormente especificado, fazem com que o $\overline{qe}(t)$ tome valores mais elevados.

Complex No gráfico 5.14 está representado os resultados dos treinos com os parâmetros σ e η gerados aleatoriamente, estando no gráfico 5.13 representados os resultados obtidos por Bruno Silva em [72]. O gráfico que contém os resultados tem ainda um pequeno quadro com os parâmetros de cada uma das funções descritas no mesmo, assim como a respectiva cor.

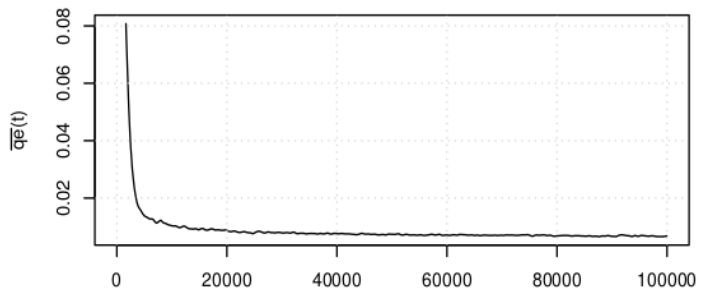


Figura 5.13: Resultados de Bruno Silva em [72] do erro médio de quantização ($\overline{qe}(t)$), utilizando a parametrização $\eta = 0.1, 0.08$ e $\sigma = 0.6, 0.2$ para o *dataset* Complex.

Os modelos treinados utilizando o conjunto de dados Complex obtiveram valores de $\overline{qe}(t)$ compreendidos entre o valor máximo de 17% e o valor mínimo de aproximadamente 1.1%. Existiram dois tipos de resultados face aos parâmetros utilizados: modelos de que resultaram valores de $\overline{qe}(t)$ que a partir da convergência do algoritmo UbiSOM tornaram-se constantes e modelos onde os valores de $\overline{qe}(t)$ oscilavam entre um valor máximo e um valor mínimo ao longo das iterações. Na tabela 5.4 pode ser observado os parâmetros para cada um dos treinos apresentados no gráfico 5.14, onde está apresentado o valor inicial e final para cada um dos parâmetros η e σ e a diferença entre estes.

σ_i	σ_f	η_i	η_f	Diferencial σ	Diferencial η	$\overline{qe}(t)$
0.53	0.3	0.65	0.7	-0.23	+0.05	1.14%
0.63	0.81	0.62	0.25	+0.18	-0.37	17.05%
0.36	0.77	0.27	0.78	+0.41	+0.51	5.7%
0.55	0.29	0.49	0.05	-0.26	-0.44	1.94%
0.09	0.69	0.67	0.24	+0.6	-0.43	9.7%

Tabela 5.4: Parâmetros η e σ utilizados nos treinos do modelo UbiSOM para o *dataset* Complex, seguindo respectivamente a ordem em que aparecem no gráfico 5.14.

Assim sendo, existem à partida três modelos que estão excluídos da análise, dado que não convergem para valores de $\overline{qe}(t)$ conforme aqueles obtidos por Bruno Silva. Esses modelos são os modelos que oscilam entre um valor máximo e um valor mínimo e que nunca chegam a estabilizar num valor aquando da convergência do algoritmo UbiSOM. Restando dois modelos, representados pelas funções de cor amarela e cor castanha no gráfico 5.14, são os modelos cuja evolução dos valores de $\overline{qe}(t)$ que mais se aproximam das melhores parametrizações obtidas por Bruno Silva. Contudo, o modelo representado pela cor amarela tem uma diferença entre valores de η iniciais e finais positiva e pelo qual, não poderá ser considerado para análise.

Restando apenas um modelo, pode-se concluir que os valores de σ terão de decrescer ao longo das iterações de um treino do modelo UbiSOM ($\sigma_f - \sigma_i \leq 0.0$) e os valores de η terão de ter imposta essa mesma descida, podendo até ser algo acentuada ($\eta_f - \eta_i \leq 0.43$), de forma a produzir resultados mais próximos daqueles quando se utilizam os parâmetros ótimos para o *dataset Complex*.

Hepta No gráfico 5.16 está representado os resultados dos treinos com os parâmetros σ e η gerados aleatoriamente, estando no gráfico 5.15 representados os resultados obtidos por Bruno Silva em [72]. O gráfico que contém os resultados tem ainda um pequeno quadro com os parâmetros de cada uma das funções descritas no mesmo, assim como a respectiva cor.

Os modelos treinados utilizando o *dataset* Hepta geraram valores de $\overline{qe}(t)$ compreendidos entre um valor mínimo de 0.03% e valores máximos de aproximadamente 18.7%. Conforme se tem vindo a verificar para outros conjuntos de dados, existem duas vertentes de resultados: um modelo que obtém valores de erro de quantização médio que após uma subida abrupta no início do treino, após a convergência, estabelecem um valor constante e assim perduram até ao final das iterações; e outro, onde existe um valor máximo e mínimo onde o $\overline{qe}(t)$ oscila até ao fim do treino do algoritmo UbiSOM. Na tabela 5.5 pode ser observado os parâmetros para cada um dos treinos apresentados no gráfico 5.16, onde está apresentado o valor inicial e final para cada um dos parâmetros η e σ e a diferença entre estes.

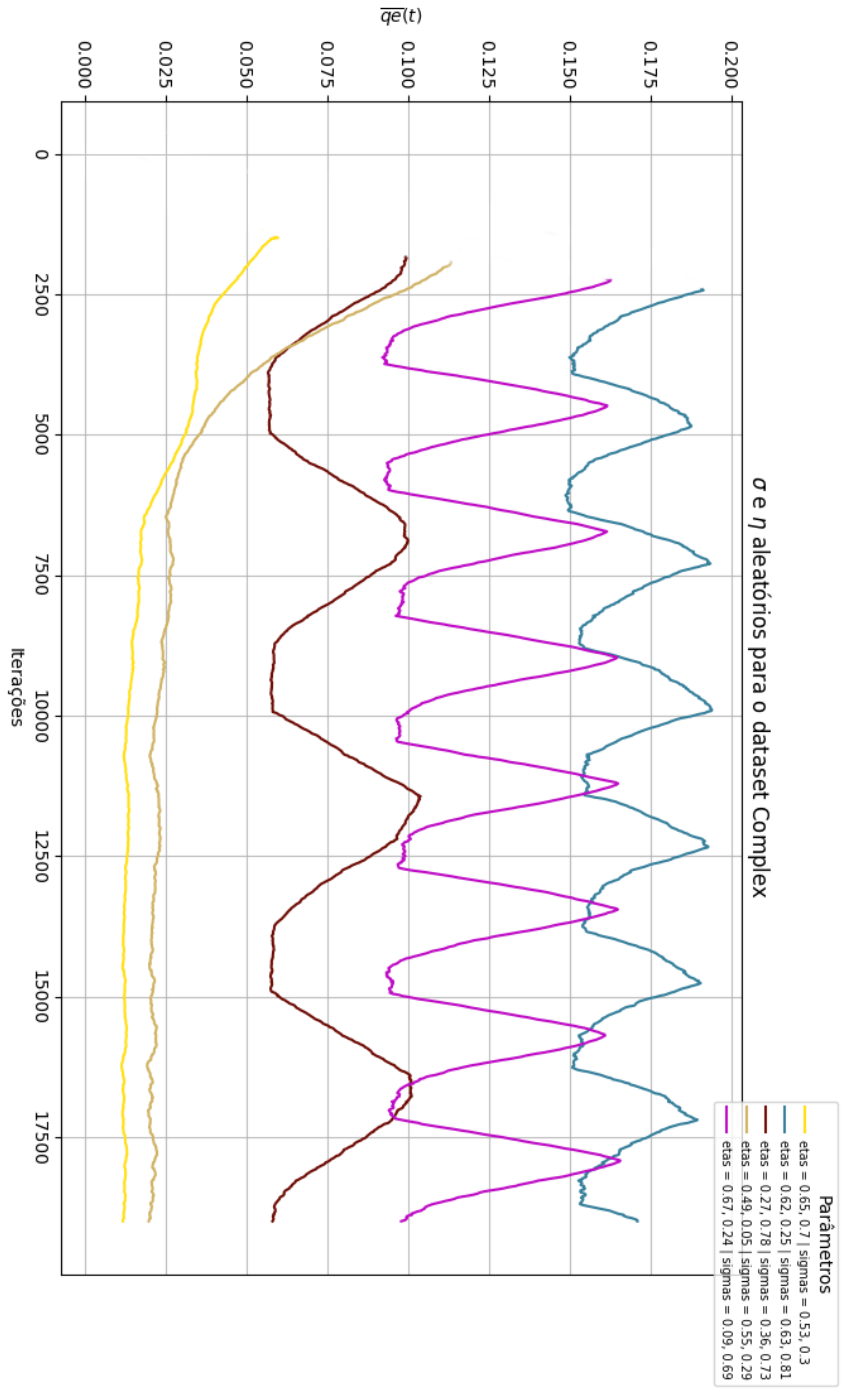


Figura 5.14: Resultados relativos a modelos UbiSOM treinados com parâmetros η e σ aleatórios utilizando o *dataset* Complex.

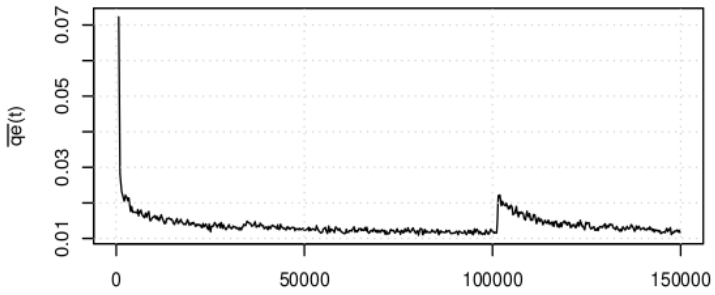


Figura 5.15: Resultados de Bruno Silva em [72] do erro médio de quantização ($\overline{qe}(t)$), utilizando a parametrização $\eta = 0.1, 0.08$ e $\sigma = 0.6, 0.2$ para o *dataset* Hepta.

σ_i	σ_f	η_i	η_f	Diferencial σ	Diferencial η	$\overline{qe}(t)$
0.51	0.14	0.98	0.87	-0.37	-0.11	1.34%
0.58	0.85	0.82	1	-0.27	+0.18	0.03%
0.98	0.2	0.95	0.03	-0.78	-0.92	2.6%
0.28	0.67	0.17	0.02	+0.39	-0.15	18.7%
0.27	0.35	0.26	0.52	+0.08	+0.26	7.5%

Tabela 5.5: Parâmetros η e σ utilizados nos treinos do modelo UbiSOM para o *dataset* Hepta, seguindo respectivamente a ordem em que aparecem no gráfico 5.16.

No caso do *dataset* Hepta, talvez derivado da sua natureza, gerou resultados um pouco contraditórios do que tinha vindo a ser a tendência. Comparando aos resultados do Bruno Silva, três modelos obtiveram resultados similares, onde o $\overline{qe}(t)$ é constante até ao fim das iterações do treino. Contudo, os valores de η e σ nestes três modelos tiveram comportamentos contraditórios. Em dois dos modelos, os valores iniciais e finais destes parâmetros decaíram de forma substancial e obtiveram bons resultados, assim como no modelo restante, subiram ao longo das iterações e obtiveram bons resultados da mesma forma. Como referido no início desta sub-secção, os modelos em que a diferença de valores de η seja positiva, serão retirados da análise automaticamente.

Desta forma, restam apenas dois modelos, em que um tem descidas acentuadas entre valores iniciais e finais de σ e η e outro que tem descidas mais ligeiras entre os tais valores iniciais e finais dos parâmetros. No modelo em que a descida é acentuada, o $\overline{qe}(t)$ obtido foi de 2.6%, claramente pior que os 1.34% produzido pelo modelo com uma ligeira diferença entre valores iniciais e finais dos parâmetros η e σ, estando mais aproximado dos resultados de Bruno Silva.

Em suma, no *dataset* Hepta o parâmetro σ terá de ter um valor inicial que não decaía mais do que 0.37 unidades para o valor final ($\sigma_f - \sigma_i \leq 0.37$) e o parâmetro η não poderá ter uma diferença maior que 0.11 unidades ($\eta_f - \eta_i \leq 0.11$).

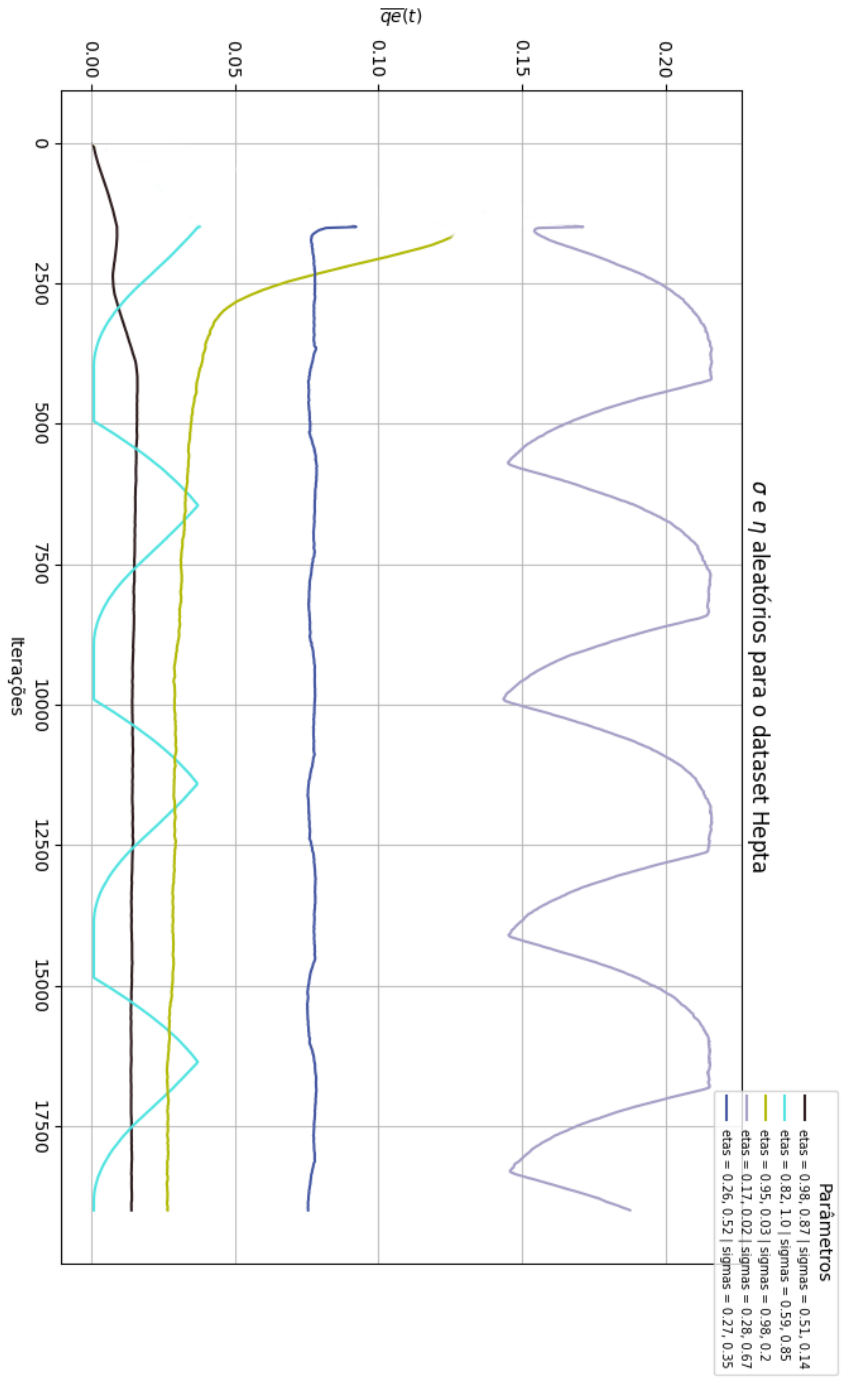


Figura 5.16: Resultados relativos a modelos UbiSOM treinados com parâmetros η e σ aleatórios utilizando o *dataset* Hepta.

Chain No gráfico 5.18 está representado os resultados dos treinos com os parâmetros σ e η gerados aleatoriamente, estando no gráfico 5.17 representados os resultados obtidos por Bruno Silva em [72]. O gráfico que contém os resultados tem ainda um pequeno quadro com os parâmetros de cada uma das funções descritas no mesmo, assim como a respectiva cor.

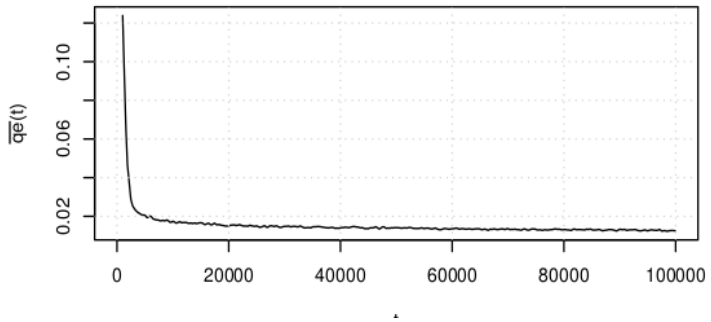


Figura 5.17: Resultados de Bruno Silva do erro médio de quantização ($\overline{qe}(t)$) em [72], utilizando a parametrização $\eta = 0.1, 0.08$ e $\sigma = 0.6, 0.2$ para o *dataset* Chain.

Os modelos treinados utilizando o *dataset* Chain geraram valores de $\overline{qe}(t)$ compreendidos entre um valor mínimo de aproximadamente 1.9% e valores máximos de aproximadamente 19.3%. Conforme se tem vindo a verificar para outros conjuntos de dados, existem duas vertentes de resultados: um modelo que obtém valores de erro médio de quantização que após uma subida abrupta no início do treino, após a convergência, estabelecem um valor constante e assim perduram até ao final das iterações; e outro, onde existe um valor máximo e mínimo onde o $\overline{qe}(t)$ oscila até ao fim do treino do algoritmo UbiSOM. Na tabela 5.6 pode ser observado os parâmetros para cada um dos treinos apresentados no gráfico 5.18, onde está apresentado o valor inicial e final para cada um dos parâmetros η e σ e a diferença entre estes.

σ_i	σ_f	η_i	η_f	Diferencial σ	Diferencial η	$\overline{qe}(t)$
0.17	0.75	0.17	0.44	+0.58	+0.27	19.3%
0.03	0.17	0.39	0.4	+0.14	+0.01	3.5%
0.61	0.68	0.73	0.54	+0.07	-0.19	13.5%
0.9	0.18	0.13	0.65	-0.72	+0.52	1.9%
0.17	0.75	0.17	0.44	+0.58	+0.27	13.3 %

Tabela 5.6: Parâmetros η e σ utilizados nos treinos do modelo UbiSOM para o *dataset* Chain, seguindo respectivamente a ordem em que aparecem no gráfico 5.18.

No caso do *dataset* Chain, apenas um modelo entra no lote dos que fazem sentido serem avaliados, que contudo, produziu um valor de $\overline{qe}(t)$ de 13.5% e desta forma não é um valor que se aproxima aqueles verificados na análise do Bruno Silva.

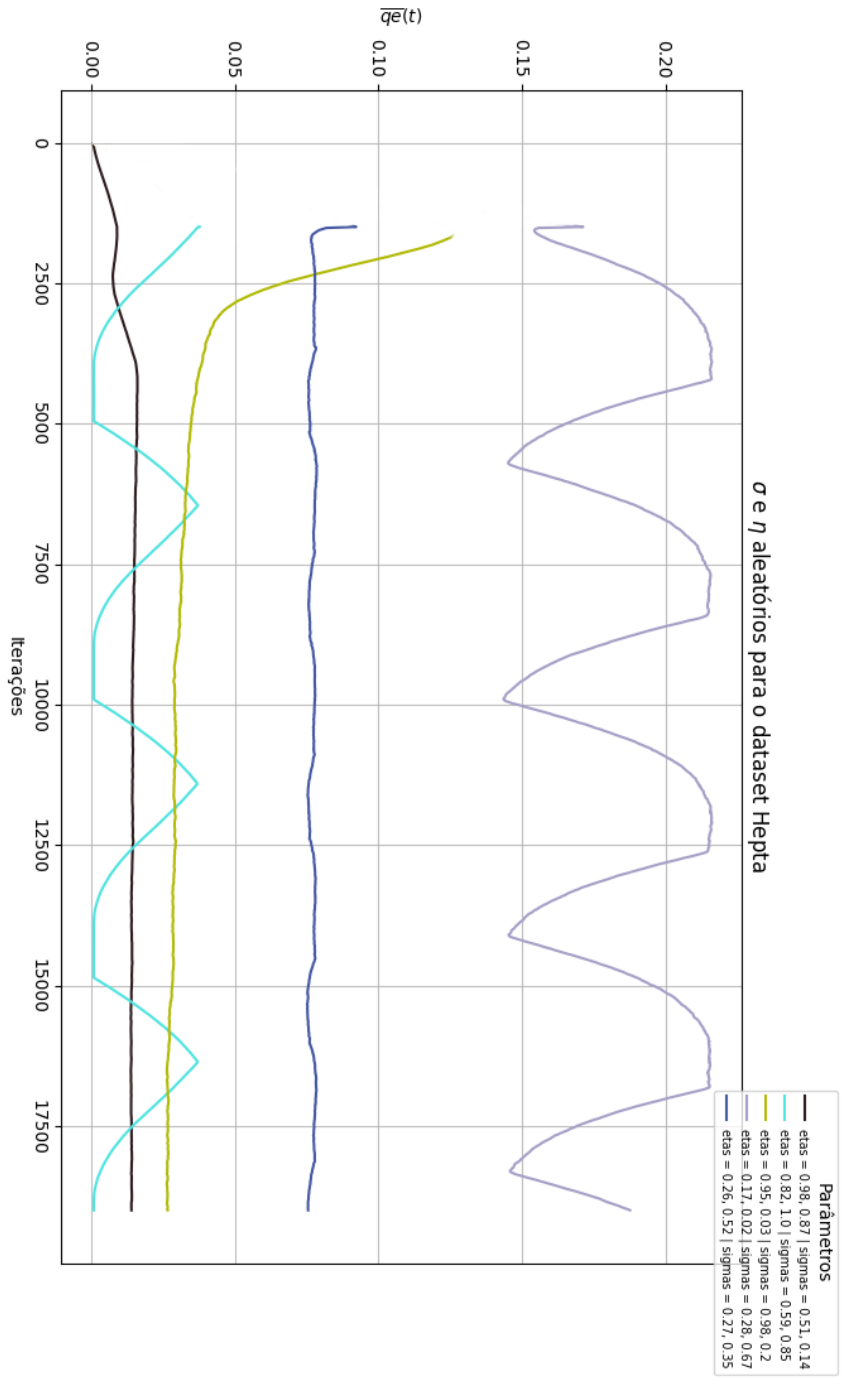


Figura 5.18: Resultados relativos a modelos UbiSOM treinados com parâmetros η e σ aleatórios utilizando o *dataset* Chain.

Resumindo, relativamente ao conjunto de dados Chain, com esta experiência não foi possível encontrar melhores valores para a parametrização do modelo UbiSOM. Portanto, aquando o treino do modelo UbiSOM utilizando este conjunto de dados, é preferível utilizar os parâmetros outrora definidos por Bruno Silva [72].

Observações Finais Efectivamente, ao longo da experiência utilizando vários *datasets*, pode-se observar que ligeiras mudanças nos parâmetros η e σ podem gerar os mesmos resultados que Bruno Silva obteve na sua análise de sensibilidade, inclusive foi possível concluir os casos em que certos parâmetros danificam os valores do erro médio de quantização e portanto entender quais os valores que se deve evitar de forma a que se tenha os melhores resultados.

No entanto, o número de iterações estar limitado é algo que pode prejudicar a conclusão geral acerca destes parâmetros, dado que certos *datasets*, como é o caso do Hepta, têm mudanças abruptas a partir das 50000 iterações. Como não foi possível treinar modelos com esse número de iterações, não foi possível verificar se utilizando os parâmetros admissíveis obtidos no decorrer desta experiência, se se essas mudanças abruptas se verificam e se a estrutura do mapa se mantém intacta.

Porém, existe crença que esta nova análise de sensibilidade (análise gráfica da variação do $\overline{qe}(t)$, nomeadamente a exclusão automática de certos modelos, como se fez ao longo da análise nesta secção) venha acrescentar novos intervalos aos parâmetros η e σ em *datasets* ainda não testados com o algoritmo UbiSOM, potenciando as melhores escolhas possíveis relativamente aos parâmetros anteriormente mencionados.

5.2.3.2 Parâmetros β e T variáveis

Nesta experiência pretende-se analisar e confirmar a parametrização ótima obtida por Bruno Silva em [72], obtendo uma parametrização mais refinada em termos dos parâmetros β e T. O β é um parâmetro inserido numa função de deriva, onde faz o equilíbrio entre o erro médio de quantização e a utilidade média de um protótipo. O T é um parâmetro que estabelece o tamanho da janela deslizante, utilizada no algoritmo UbiSOM.

Os testes realizados nesta experiência são feitos sobre os *datasets* Clouds, Complex, Hepta e Chain, não havendo resultados para serem comparados com o conjunto de dados Iris, frequentemente utilizado neste projecto. Irão ser realizados cerca de cinco treinos do modelo UbiSOM para cada um dos *datasets* com os parâmetros β e T aleatórios (estando compreendidos entre $\beta \in [0.6; 0.9]$ e $T \in [1000; 2500]$) e os resultados serão expostos num gráfico, avaliando o erro médio de quantização ao longo das iterações, sendo comparado directamente com os resultados do Bruno Silva. Já a parametrização do σ e η estão fixos nos valores que Bruno Silva determinou como mais admissíveis, sendo eles: $\sigma_i = 0.6$, $\sigma_f = 0.2$, $\eta_i = 0.1$ e $\eta_f = 0.08$. Os treinos do modelo UbiSOM serão parados às 19000 iterações.

Clouds Os resultados referentes ao *dataset* Clouds estão ilustrados no gráfico 5.19 e os resultados referentes ao treino deste conjunto de dados com os parâmetros $\beta = 0.6$ e $T = 2500$ de Bruno Silva estão representados no gráfico 5.11.

Como o *dataset Clouds* é um conjunto de dados não estacionário e não deriva dentro das primeiras 19000 iterações, não pode ser retirada qualquer conclusão sobre os melhores parâmetros β e T, até porque a parametrização obtida por Bruno Silva estava sujeita a permitir plasticidade.

Complex Os resultados referentes ao *dataset* Complex estão ilustrados no gráfico 5.20 e os resultados referentes ao treino deste conjunto de dados com os parâmetros $\beta = 0.8$ e $T = 1500$ de Bruno Silva estão representados no gráfico 5.13.

Todos os treinos com a parametrização aleatória de β e T tiveram um desempenho similar àquele representado no gráfico 5.13. Às aproximadamente 5000 iterações já teriam um valor de $\overline{qe}(t)$ igual àquele obtido por Bruno Silva e foram constantes até ao fim da iterações do treino. De referir que a aleatoriedade dos valores de β e T estão compreendidos nos valores apresentados no início desta sub-secção.

Existiu pelo menos um modelo de parâmetros $\beta = 0.74$ e $T = 1130$ que começou a tender para melhores valores de $\overline{qe}(t)$ mais cedo que os outros treinos, aproximadamente às 2000 iterações. Todos os outros modelos têm valores de T maiores que neste caso, sendo o β mais variável, mas podemos concluir que conjugando o β e T para ter parâmetros ótimos, terão de estar compreendidos abaixo de 0.74 para o caso do β ($0.6 \leq \beta \leq 0.74$) e abaixo de 1130, para o caso de T ($1000 \leq T \leq 1130$).

Hepta Os resultados referentes ao *dataset* Hepta estão ilustrados no gráfico 5.21 e os resultados referentes ao treino deste conjunto de dados com os parâmetros $\beta = 0.6$ e $T = 500$ de Bruno Silva estão representados no gráfico 5.15.

Como o *dataset Hepta* é um conjunto de dados não estacionário e não deriva dentro das primeiras 19000 iterações, não pode ser retirada qualquer conclusão sobre os melhores parâmetros β e T, até porque a parametrização obtida por Bruno Silva estava sujeita a permitir plasticidade.

Chain Os resultados referentes ao *dataset* Chain estão ilustrados no gráfico 5.22 e os resultados referentes ao treino deste conjunto de dados com os parâmetros $\beta = 0.6$ e $T = 1000$ de Bruno Silva estão representados no gráfico 5.17.

Todos os treinos com a parametrização aleatória de β e T tiveram um desempenho similar àquele representado no gráfico 5.17. Às aproximadamente 10000 iterações já teriam um valor de $\overline{qe}(t)$ igual àquele obtido por Bruno Silva e foram constantes até ao fim da iterações do treino.

Existiu pelo menos um modelo de parâmetros $\beta = 0.9$ e $T = 1199$ que começou a tender para melhores valores de $\overline{qe}(t)$ mais cedo que os outros treinos, aproximadamente às 3000 iterações. Podemos admitir que o β para o *dataset* Chain pode variar dentro dos

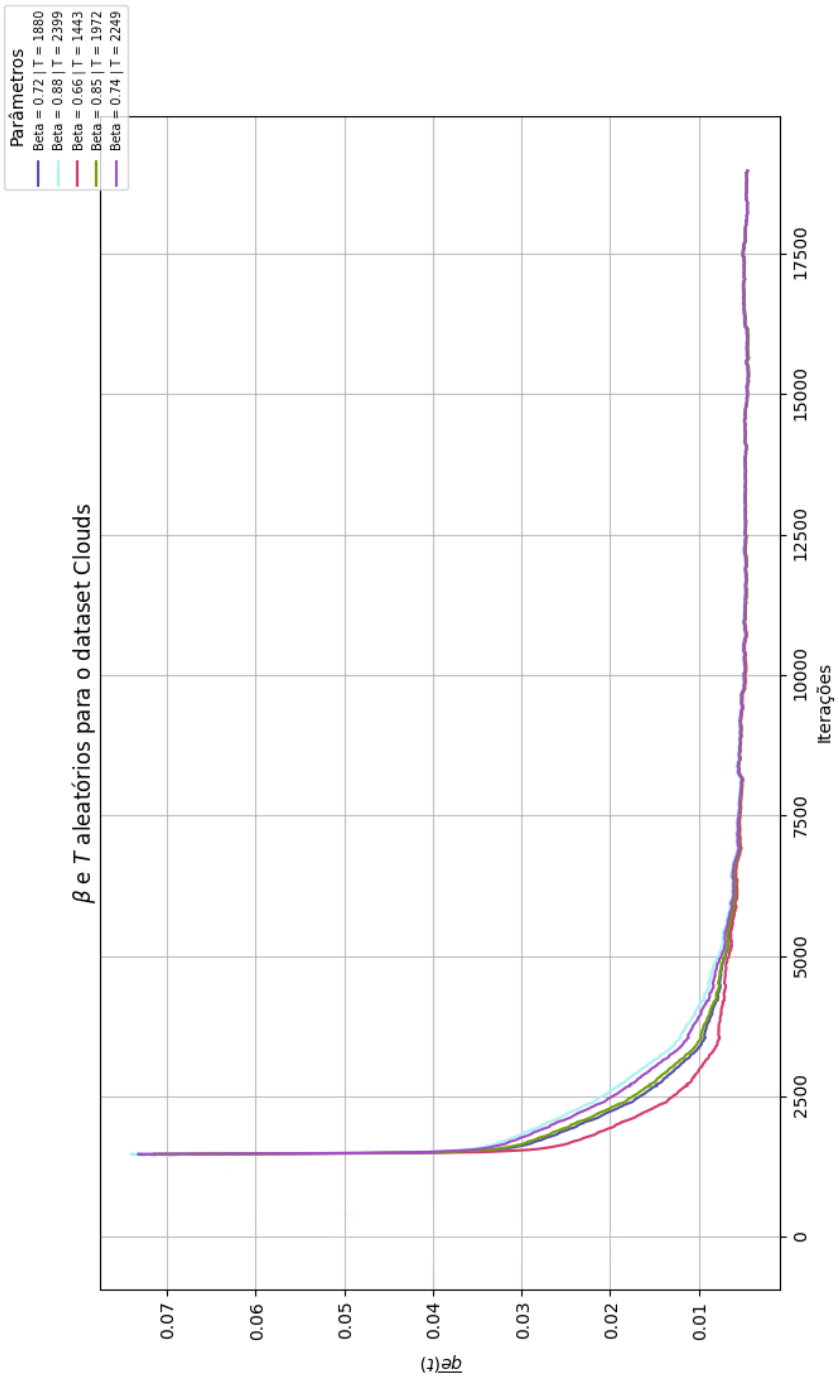


Figura 5.19: Resultados relativos a modelos UbiSOM treinados com parâmetros β e T aleatórios utilizando o *dataset* Clouds.

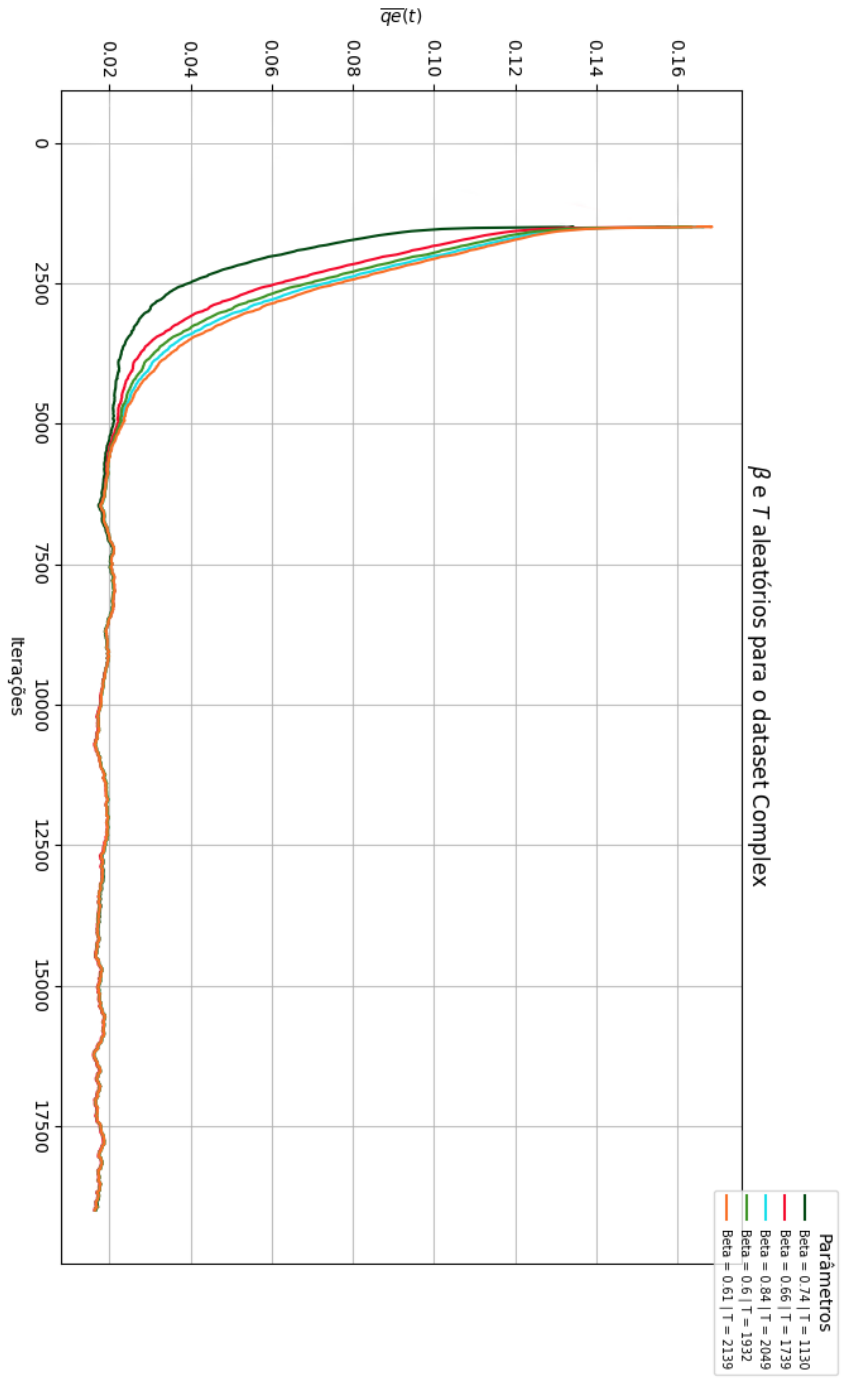


Figura 5.20: Resultados relativos a modelos UbiSOM treinados com parâmetros β e T aleatórios utilizando o *dataset* Complex.

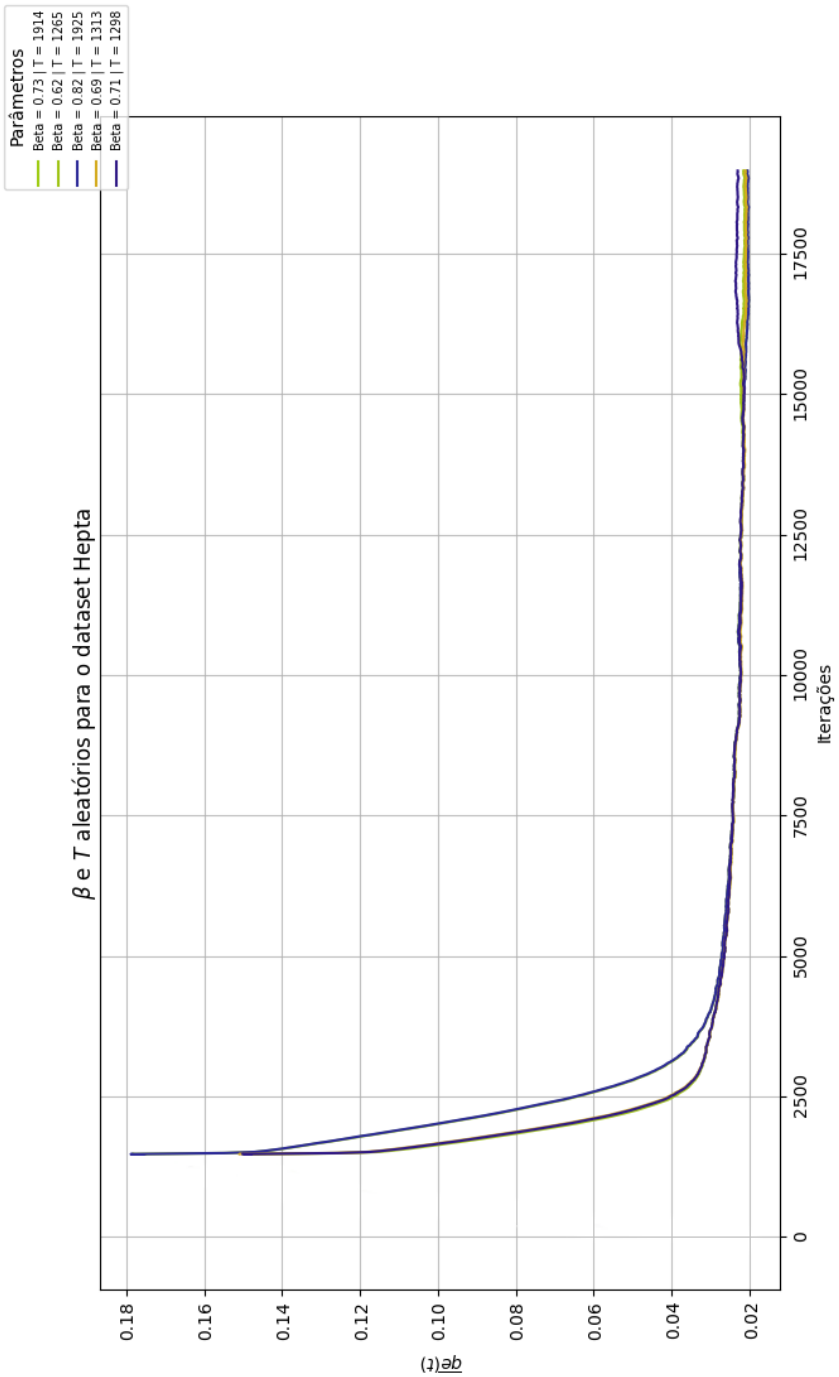


Figura 5.21: Resultados relativos a modelos UbiSOM treinados com parâmetros β e T aleatórios utilizando o *dataset* Hepta.

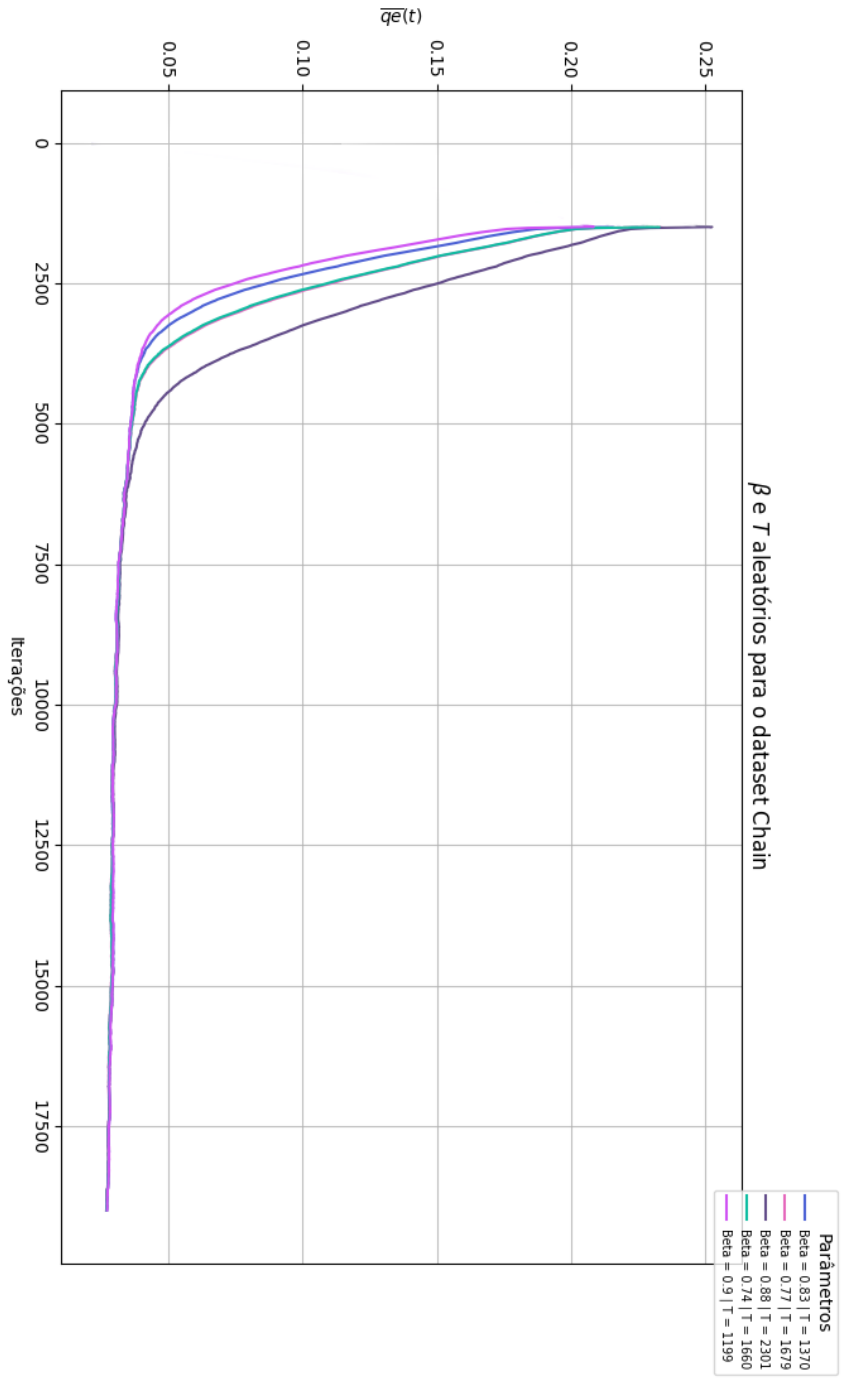


Figura 5.22: Resultados relativos a modelos UbiSOM treinados com parâmetros β e T aleatórios utilizando o *dataset* Chain.

parâmetros admissíveis apresentados no início da secção ($0.6 \leq \beta \leq 0.9$), enquanto que o parâmetro T é mais adequado estar compreendido entre 1000 e 1199 ($1000 \leq T \leq 1199$)

Observações Finais Conforme foi referido no início da sub-secção de análise aos hiperparâmetros do UbiSOM, estes testes não têm o mesmo número de iterações que aqueles feitos por Bruno Silva, mas que ainda assim dão para retirar conclusões (naqueles *datasets* que são estacionários), dado que o comportamento do $\overline{qe}(t)$ é determinante assim que o algoritmo UbiSOM entre na fase de convergência. Contudo, não dá para analisar certas fases de certos *datasets* como o caso do Clouds e do Hepta, onde a partir das 50000 iterações sofre uma mudança abrupta e faz disparar os valores do erro médio de quantização.

Ainda assim deu para comprovar a viabilidade dos parâmetros obtidos por Bruno Silva e em alguns casos, deu para refinar os seus valores e obter novos intervalos de admissibilidade, podendo servir como ponto de partida para novos *datasets* que se queiram treinar com o algoritmo UbiSOM e desta forma obter os resultados mais próximos do caso ótimo.

5.3 Performance

Para testar e validar a questão da performance do sistema, foram feitas experiências relativamente aos tempos que o algoritmo demora a executar os vários treinos em várias unidades de computação. Fundamentalmente, as questões que visam ser respondidas por estas experiências são as seguintes:

- Um maior número de iterações compensa em relação a treinar vários mapas separadamente?
- Compensa utilizar apenas uma unidade de computação com vários treinos a serem executados simultaneamente ou compensa utilizar mais unidades de computação?
- O mecanismo de selecção de *features* induz uma melhor performance geral no sistema?

Esta secção irá conter então as experiências efectuadas, com uma breve descrição das mesmas, inclusão de gráficos e/ou tabelas para ilustração dos resultados e uma pequena análise dos mesmos, visando como objectivo final responder às perguntas apresentadas acima.

Os testes nesta secção foram realizados com múltiplas unidades de computação, tendo sido os treinos distribuídos ou não, estando devidamente identificados quando cada um dos casos fora utilizado. Para executar de forma paralela, os treinos foram coordenados com o *tmux*, abreviatura de *terminal multiplexer*, um *software* que permite que um certo número de terminais seja criado, acedido e controlado a partir de um único ecrã [37]. Isto permite que várias instâncias *ssh* (ou instâncias locais) sejam controladas de forma

síncrona e assim, os treinos do algoritmo UbiSOM são invocados e executados de forma paralela, em várias unidades de computação.

5.3.1 Maior número de iterações *versus* vários mapas a treinar separadamente

Com esta experiência pretende-se determinar qual o cenário mais vantajoso em relação a treinar um modelo UbiSOM com um número elevado de iterações, em comparação com vários treinos a executar na mesma unidade de computação. Tenciona-se desta forma analisar se o conceito de treinar vários mapas em simultâneo influencia uma minimização dos valores de tempos de execução dos treinos do modelo UbiSOM, avaliando se a possibilidade de testar mapas em simultâneo é eficiente para este sistema.

Foram efectuados treinos com os *datasets* descritos na sub-secção 5.1.2 e foram efetuados nos seguintes moldes:

- Um treino do modelo UbiSOM com **12000** iterações.
- Um treino do modelo UbiSOM com **18000** iterações.

Após os treinos serem executados com a metodologia apresentada acima, são realizados os treinos em que uma unidade de computação acarreta vários a treinar simultaneamente. O objectivo é executar tantos mapas a 6000 iterações numa unidade de computação que se aproximem dos valores de iterações acima descritos.

Para os testes com vários mapas a treinar separadamente, segue-se então uma descrição destes para desambiguação, invocando o seguinte exemplo:

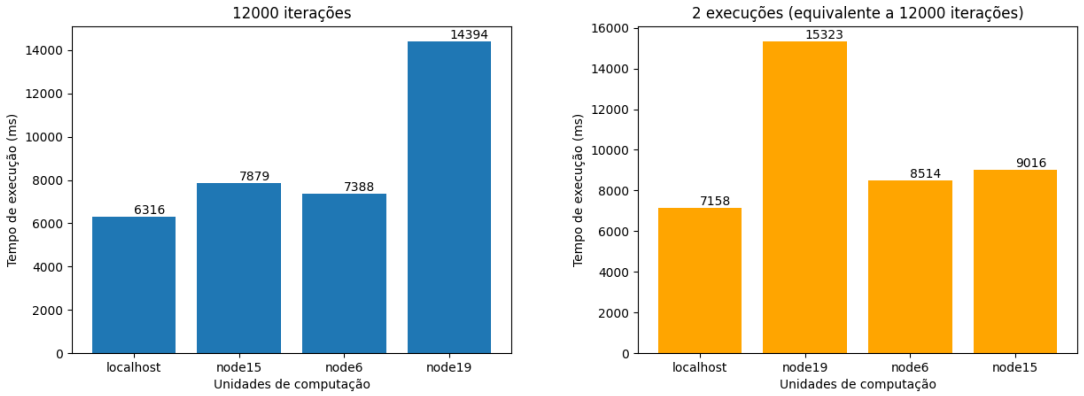
- Um mapa é treinado com 12000 iterações. Para equivaler a este número, treina-se dois mapas em várias unidades de computação, em que cada um destes treinos possuí 6000 iterações. Desta forma, pode ser feita uma comparação com o cenário de várias unidade de computação a executar um treino único com 12000 iterações.
- Um mapa é treinado com 18000 iterações. Para equivaler a este número, treina-se três mapas em várias unidades de computação, em que cada um destes treinos possuí 6000 iterações. Desta forma, pode ser feita uma comparação com o cenário de várias unidade de computação a executar um treino único com 18000 iterações.

Os resultados desta experiência foram transversais a todos os conjuntos de dados utilizados nos treinos aqui presentes, pelo que só irá ser feita uma análise a um *dataset* (Clouds), para simplificação dessa mesma análise. A escolha do *dataset* apresentado não envolveu nenhuma predisposição, apenas se decidiu escolher um dos conjuntos de dados para ilustrar os resultados desta experiência. Os resultados dos treinos dos restantes *datasets* estão presentes no anexo VII.

Os resultados serão apresentados em formato de gráfico de barras, onde o eixo das ordenadas representa o tempo de execução (em milissegundos) e no eixo das abcissas estão representados cada um dos nós de computação onde estes treinos foram realizados.

Resultados relativos ao *dataset* Clouds Nos gráficos 5.23a e 5.23b estão representados os resultados relativos a treinos com 12000 iterações, seja numa única unidade de computação, seja distribuído por duas máquinas (cada um dos treinos tem 6000 iterações), utilizando o *dataset* Clouds.

Conforme se observa nos dois gráficos, a unidade de computação *localhost* é aquela que leva menos tempo a treinar os modelos nos dois casos e a unidade de computação *node19* é aquela que toma mais tempo. Efectivamente, os treinos que melhoram a performance geral do sistema são aqueles que utilizam um maior número de iterações invés de um maior número de *threads*. Veja-se o caso do *node6*, onde o treino único com 12000 iterações demorou 7388 milissegundos a ser concluído, enquanto que lançar duas *threads* com 6000 iterações cada, demorou mais 1126 milissegundos do que o primeiro caso.

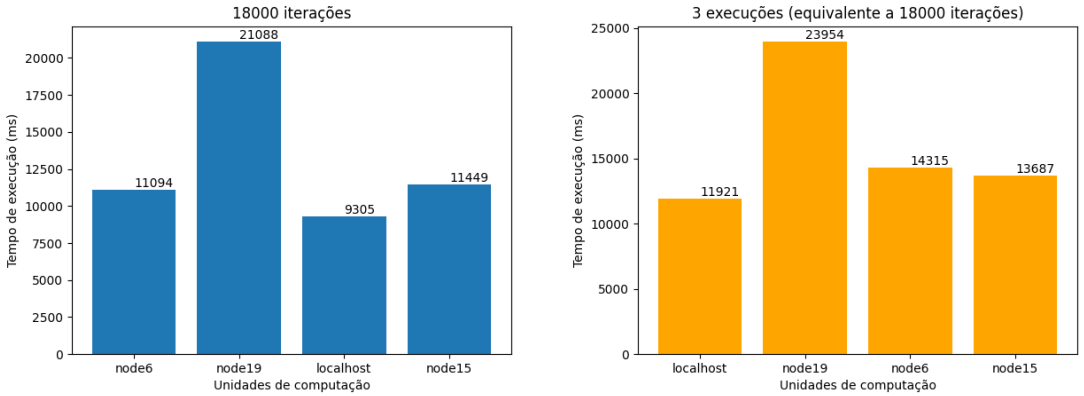


(a) Resultados relativos ao tempo de execução do treino do algoritmo UbiSOM com 12000 iterações, usando uma unidade de computação, para o *dataset* Clouds.

(b) Resultados relativos ao tempo de execução de dois treinos de 6000 iterações cada, utilizando várias unidades de computação, para o *dataset* Clouds.

Figura 5.23: Cenário de treino com 12000 iterações *vs.* dois treinos em cada unidade de computação cada um com 6000 iterações.

Essa tendência é também observável no outro caso testado, utilizando 18000 iterações num treino em uma unidade de computação e fazendo três treinos em várias unidades de computação, perfazendo 18000 iterações. Os gráficos 5.24a e 5.24b são referentes ao treino único com um elevado número de iterações (18000) e a computação de três treinos em várias unidades de computação, respectivamente.



(a) Resultados relativos ao tempo de execução do treino do algoritmo UbiSOM com 18000 iterações, usando uma unidade de computação, para o *dataset* Clouds.

(b) Resultados relativos ao tempo de execução de três treinos de 6000 iterações cada, utilizando várias unidades de computação, para o *dataset* Clouds.

Figura 5.24: Cenário de treino com 18000 iterações *vs.* três treinos em cada unidade de computação cada um com 6000 iterações.

Como se pode observar, a melhor unidade de computação fora o *localhost* e a pior em termos de tempos de execução foi o *node19*, assim como no teste anterior. Igualmente ao teste anterior, o método mais eficiente para o *dataset* Clouds é utilizar uma unidade de computação com mais iterações, conforme se pode observar nos resultados do *node6*. A execução de um treino com 18000 iterações no *node6* é cerca de 11094 milissegundos, enquanto que executando três treinos (perfazendo 18000 iterações) é 14315 milissegundos, existindo uma diferença de 3221 milissegundos, 1.3x vezes mais demorado que no caso de um treino único.

Efectivamente, o melhor caso de uso do sistema é computar um treino único com um elevado número de iterações, invés de utilizar vários treinos numa unidade de computação com poucas iterações, de forma a perfazer o tal elevado número de iterações.

Observações Globais/Análise No âmbito geral, os resultados obtidos foram esperados. Computacionalmente, é muito mais moroso lançar várias *threads* que em simultâneo usam os recurso computacionais de uma dada unidade de processamento e que podem em certas alturas saturar a mesma, gerado um certa latência em obter os cálculos dos vários processos que compõem o algoritmo UbiSOM. Este algoritmo e este sistema portanto beneficiam a utilização de uma unidade de processamento com o número máximo de iterações possível.

Contudo, não foi de todo esperado que por exemplo a unidade de computação *node19* fosse substancialmente mais lenta que todas as outras, em todos os testes, em todos os *datasets*, dado que não é uma unidade de computação com recursos inferiores a por exemplo o *localhost*. No entanto, o *localhost* obtém sempre os melhores resultados porque não está a ser acedido de forma remota e como os outros nós de computação estão a ser acedidos dessa forma, então pode ser induzido um pouco de *overhead*, dado que o

alocamento de recursos do *cluster* inicialmente falado está feito de forma diferente do que utilizar uma unidade de computação localmente. No entanto, a justificação para o *node19* pode ser que para além do alocamento de recursos ter um processo diferente e isso poder induzir latência nas execuções, a unidade de computação em questão possuí um valor inferior de memória RAM em relação a outras unidades de computação utilizadas para esta experiência, atrasando portanto os treinos nesta unidade de computação de uma forma geral.

Noutro prisma, a velocidade do *clock* do *localhost* pode também ter contribuído para resultados tão favoráveis, apesar da unidade de processamento ter à sua disposição menores recursos ao nível de *cores*, *threads* e memória RAM, em relação a outras unidades de processamento utilizadas nesta experiência.

5.3.2 1 unidade de computação *versus* múltiplas unidades de computação

O objectivo desta experiência é comprovar qual o modelo de execução mais eficiente para executar mais treinos, na medida em que se deve utilizar mais ou menos unidades de computação para atingir o objectivo deste sistema: determinar as melhores parametrizações para um certo conjunto de dados. Essencialmente, trata-se em testar o sistema e compreender qual a forma mais eficiente de executar mais mapas e de que forma as execuções devem ser distribuídas por cada uma das unidades de computação disponíveis em determinado momento.

Convém referir que as várias *threads* de treino não comunicam entre si, ou seja, não existe *overhead* (que se considerou desnecessário implementar) em comunicação intra-execuções, dado que não seria necessário acrescentar mais uma camada de complexidade num algoritmo que se for lançado numa *thread* é auto-suficiente para gerar resultados. Para além disso, todas as execuções vão ter parâmetros diferentes e seria um cenário despropositado estar a trocar mensagens/resultados com outros treinos que não pertenceriam ao mesmo escopo de parâmetros.

O sistema permite executar vários mapas de duas formas: executar um certo número de *threads* em que cada *thread* faz um treino de um mapa e a outra forma sendo a distribuição de forma equalitária de vários treinos pelas unidades de computação, algo que na secção 3.2 foi referido como sendo uma espécie de balanceamento de carga.

Para testar estes cenários, foram utilizados os *datasets* descritos na sub-secção 5.1.2 e estes foram treinados nos seguintes moldes:

- Em cada uma das unidades de computação do conjunto destas, são executados 1, 2, 4, 8, 10 e 20 treinos do modelo UbiSOM.

- Duas unidades de computação são escolhidas (neste caso, *localhost* e *node17*) e são-lhes atribuídas 2, 4, 8, 10, 20 treinos em simultâneo, distribuídos por cada uma das unidades de computação.

- Quatro unidades de computação são escolhidas (neste caso, *localhost*, *node15*, *node17* e *node19*) e são-lhes atribuído 4, 8, 16 e 20 treinos em simultâneo, distribuídos por cada uma das unidades de computação.

O número de execuções no caso do balanceamento de carga são escolhidos de acordo com o número de execuções escolhidas no caso de existir uma unidade de computação única de forma a poder comparar o cenário de utilizar uma unidade de computação única e um cenário onde exista distribuição de carga por várias unidades de computação. Ou seja, uma execução do algoritmo UbiSOM em cada uma de quatro máquinas, equivale ao caso de ter uma única unidade de computação a executar quatro treinos. Convém referir que todos os treinos do modelo UbiSOM foram realizados com 6000 iterações, servindo como um bom número de iterações, dado que o algoritmo UbiSOM atinge a convergência a partir das 3000 iterações.

De forma a concluir os tempos de execução em unidades de computação únicas, são analisados todos os tempos de execução de cada um dos treinos, sendo escolhido para representação o tempo máximo de execução, dado que todas as execuções começam ao mesmo tempo. Isto é, se existirem dois mapas a executar numa unidade de computação e os dois demoram dois e quatro segundos, o tempo de execução escolhido para a análise é quatro segundos.

Análise aos resultados utilizando os *datasets* Hepta, Chain, Complex, Clouds Os resultados dos treinos para os *datasets* Hepta, Chain, Complex e Clouds, os resultados encontram-se ilustrados no anexo VIII. Para os *datasets* enunciados anteriormente, segue-se então a análise geral aos resultados:

- Os tempos de computação de 20 mapas têm melhores valores no caso da distribuição equalitária de mapas pelas unidades de computação. Contudo, em relação em utilizar outro número de mapas no caso da distribuição equalitária, os tempos de execução são constantes, havendo pouca variância entre eles, mas para o caso de utilizar 20 mapas em duas máquinas, o tempo de execução sobe ligeiramente.

- Os tempos de computação de 10 mapas têm tempos de execução menores para unidades de computação individuais do que a distribuição equalitária por 2 máquinas, excepto nas unidades de computação *localhost* e *node19*.

- Os tempos de computação de 4 treinos são maiores num modelo de distribuição equalitária de mapas por unidades de computação, utilizando 2 e 4 máquinas, excepto quando se computam 4 mapas nas unidades de computação *node6*, *node17* e *node19*.

Em suma, são retiradas as seguintes conclusões:

- O modelo de execuções com distribuição equalitária de execuções por unidades de computação tem tempos de execução que evoluem de forma constante, com

pouca variação. A excepção confirma-se no caso dos 20 mapas, quando os treinos são distribuídos por duas unidades de computação, onde o tempo de execução tem uma considerável subida em relação a utilizar outro número de mapas.

- Com mais mapas a treinar numa unidade de computação, quando a quantidade de mapas ultrapassa as *threads* em cada uma dessas unidades de computação, a certo momento os tempos de execução aumentam consideravelmente, excepto na unidade de computação *node6* e *node17*.

- Se o número mapas que se pretenderem computar forem inferiores ao número de *threads* que uma unidade de computação possuí, é preferível utilizar apenas uma unidade de computação. Se esse número for superior ao número de *threads* das unidades de computação disponíveis a certo momento, é preferível distribuir de forma equalitária os mapas por muitas unidades de computação.

Contudo, para o *dataset* Iris, houve uma pequena divergência nos resultados, sendo necessário uma análise individual e profunda aos resultados dos treinos envolvendo este conjunto de dados.

Análise aos resultados dos treinos utilizando o *dataset* Iris Os resultados para o *dataset* Iris estão representados nas figuras 5.26 e 5.25, sendo respectivamente os tempos de execução utilizando uma unidade de computação e os tempos de execução utilizando uma distribuição igualitária de execuções por várias unidades de computação. Os resultados foram algo variáveis, mas existem conclusões que se podem retirar à partida:

- O modelo de execuções com distribuição equalitária de execuções por unidades de computação tem tempos de execução constantes.

- Com mais mapas a treinar numa unidade de computação, quando a quantidade de mapas ultrapassa as *threads* em cada uma dessas unidades de computação, a certo momento os tempos de execução aumentam quase de forma exponencial.

Contudo, existem excepções nas conclusões retiradas anteriormente. Nos pontos seguintes apresentam-se essas excepções, para cada unidade de computação:

- No nó *localhost* e no nó *node19*, para uma execução com 20 treinos em simultâneo, o tempo de execução é sensivelmente 2.5x maior relativamente a utilizar quatro máquinas com treinos distribuídos e 3x maior no caso de utilizar duas máquinas.

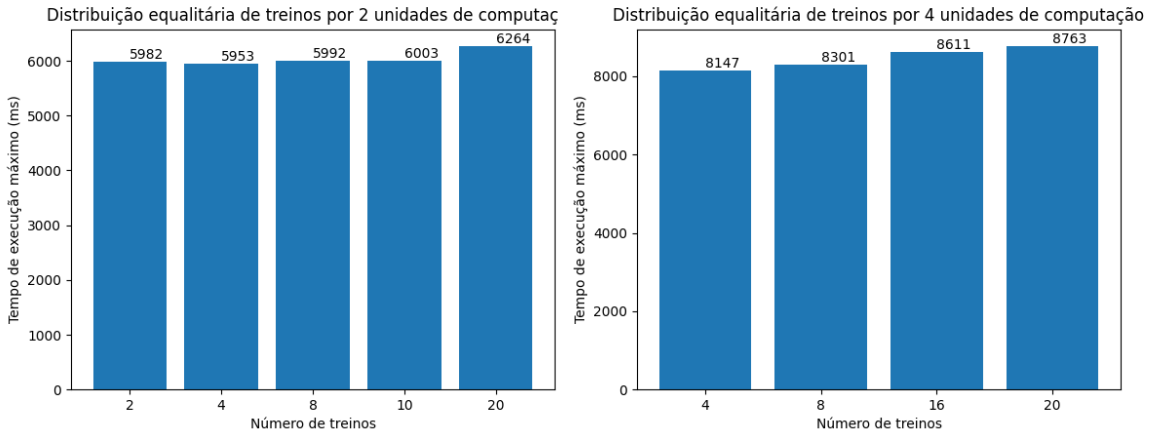
- No caso de 4, 8 e 10 treinos nos dois modelos de execução, as máquinas *node6* e *node15* com esse número de treinos, têm tempos de execução mais reduzidos que utilizar duas e quatro máquinas com treinos distribuídos de forma equalitária.

- No caso de 20 treinos nos dois modelos de execução, as máquinas *node6* e *node17* têm tempos inferiores àqueles resultantes de distribuir treinos em duas e quatro máquinas de forma equalitária.

Apesar dos resultados serem quase similares aos resultados dos outros *datasets*, relativamente ao tempo de execução relativamente à distribuição equalitária de mapas por várias unidades de computação, este é sempre constante. Nos outros *datasets*, no caso de se utilizar duas máquinas, o tempo de computar 20 treinos do modelo UbiSOM sobe em relação a outro número de treinos executados. Uma das justificações que pode ser elaborada é que o *dataset* Iris é constituído por menos observações e portanto, demora teoricamente menos tempo no processo de *parsing*, podendo conduzir a um tempo reduzido de execução geral.

Outra das justificações que poderia ser válida é o facto do *dataset* Iris é o único do grupo de conjunto de dados utilizados para este projecto que é constituído à partida por valores inteiros, mas como este sistema tem uma fase de normalização de valores dos *datasets*, facilmente se excluiu essa possibilidade. A validade deste argumento assentava-se no facto de cálculos feitos com valores de vírgula flutuante serem mais complexos de calcular nas operações de divisão e multiplicação do algoritmo UbiSOM, mas dado que os dados neste sistema são normalizados (para valores de natureza inteira) numa certa fase do pré-treino, então o argumento não progrediu como sendo a razão pela qual o *dataset* Iris teve uma vantagem no caso de distribuir os mapas de forma equalitária por várias unidades de computação em relação a outros *datasets*.

A última justificação e pensa-se que tenha sido a mais válida é que tenha havido um lapso na recolha dos dados dos tempos de execuções relativamente a este *dataset* e portanto tenha influenciado de forma positiva os tempos de execução.



(a) Tempo de execução resultante de treinos distribuídos de forma equalitária por duas unidades de computação.

(b) Tempo de execução resultante de treinos distribuídos de forma equalitária por quatro unidades de computação.

Figura 5.25: Resultados referentes às experiências com distribuição equalitária de treinos por várias unidades de computação, utilizando o *dataset* Iris.

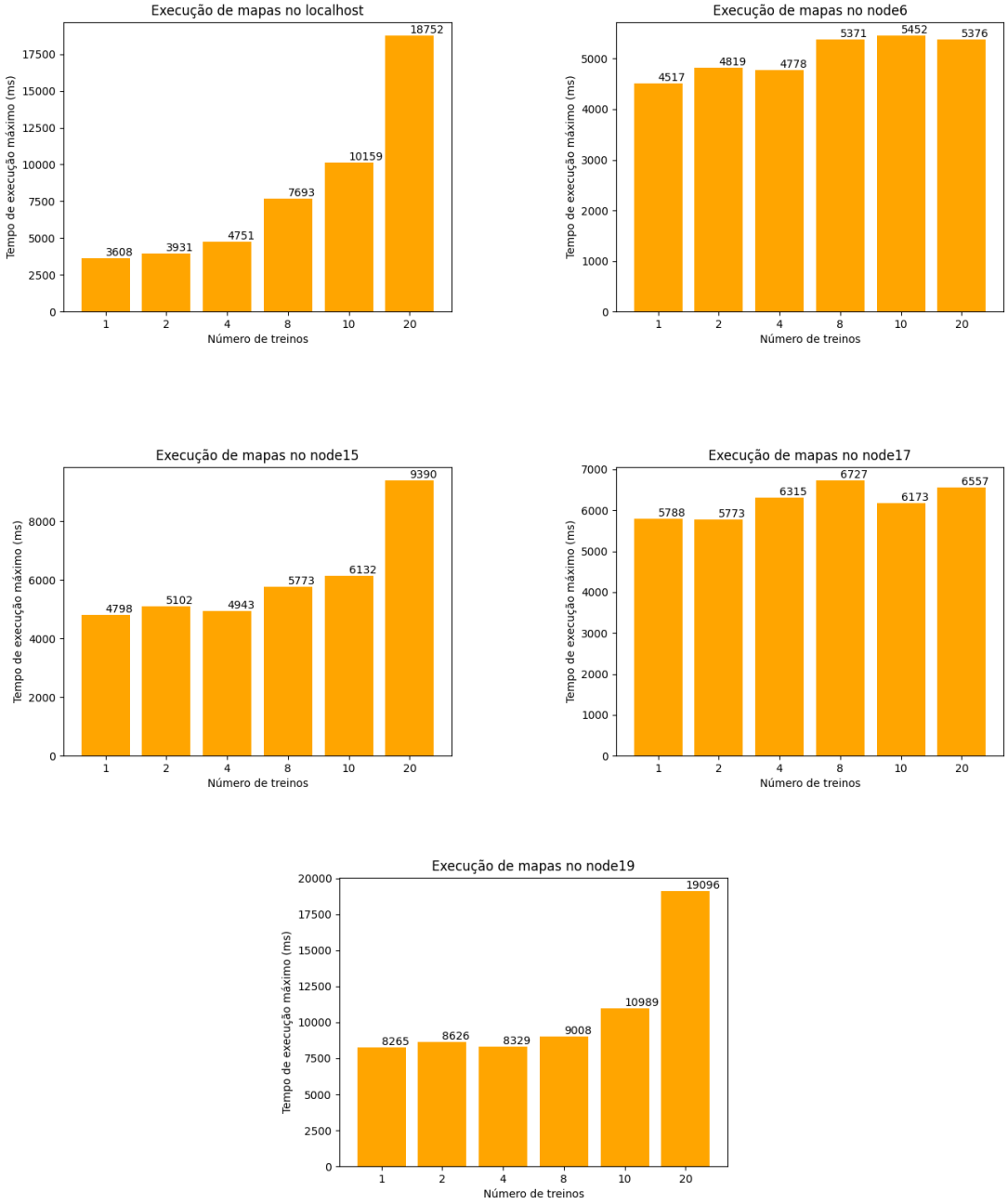


Figura 5.26: Tempos de execução resultantes de um número de treinos variáveis numa unidade de computação única utilizando o *dataset* Iris.

Observações Globais/Análise Num âmbito geral, as conclusões que se retiram é que é preferível utilizar uma única unidade de computação que suporte vários treinos simultaneamente. Contudo, a partir de um certo valor de mapas, que seja correspondente ao valor de *threads* que uma certa unidade de computação suporta, é preferível utilizar um modelo de execução que envolva mais unidades de computação com treinos distribuídos em igual número pelas unidades de computação disponíveis a certo momento. Ainda assim, dado que nos resultados geralmente este modelo de execução anteriormente falado tem tempos de execução que evoluem de forma constante, em suma, pensa-se que seja preferível utilizar sempre este método de execução, dado que com este sistema pretende-se calcular sempre o maior número de mapas possível e de forma simultânea. Assim, aproveita-se de forma eficiente os recursos disponíveis, não saturando uma unidade de computação com muitos treinos do algoritmo UbiSOM e gerando os resultados em tempo útil.

Ainda assim, nota-se que o nó de computação *node19* e o *localhost* a partir de um certo número de treinos, demoraram muito mais tempo que outras unidades de computação, dado o seu pequeno valor de *threads* comparativamente a outros nós. De certa forma, conclui-se que seja preferível sempre utilizar processadores que ultrapassem as 16 *threads*, mesmo para o caso que se faça uma distribuição de carga por várias unidades de computação. Não obstante, não foi esperado que o nó *node19* demorasse tanto tempo em executar os mapas, dado que tem recursos aproximados àqueles presentes no *node15*, que teve um comportamento favorável ao longo desta experiência.

Um detalhe que é transversal aos treinos dos *datasets* Complex e Clouds é que nas execuções de 16 treinos distribuindo-os de uma forma equalitária por quatro unidades de computação, o tempo de execução foi mais elevado comparativamente à execução de 20 treinos na mesma modalidade. Este detalhe confirma então um lapso da recolha dos resultados, que pode ter sido o mesmo tipo de lapso a originar os tempos constantes na modalidade de distribuição de treinos de forma equalitária no conjunto de dados Iris. Ainda assim, este diferencial nos resultados do Complex e Clouds pode também ter sido originado pelo facto de ter sido utilizado o *node19*, que já havia ter sido confirmado como sendo uma unidade de computação que gera os resultados mais demorados relativamente a treinos do modelo UbiSOM. Pode-se concluir assim que utilizando várias unidades de computação e distribuindo os treinos de forma equalitária compensa sempre em relação a saturar uma unidade de computação com muitos treinos, excepto quando é utilizado uma unidade de computação com um processador inferior ou uma quantidade de memória RAM inferior a outras unidades de computação dentro dessa *pool* de unidades de computação, como é o caso do *node19*.

Contudo, apesar destes (possíveis) lapsos na avaliação dos resultados, isto não afecta a conclusão relativamente a esta experiência, dado que houve resultados suficientes para serem avaliados os dois cenários e ser confirmado a melhor hipótese, aquela representada no início desta secção.

5.3.3 *Feature Switching*

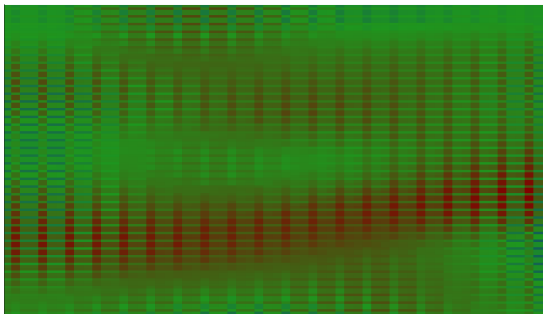
O objectivo desta experiência é analisar o impacto do mecanismo de *feature switching* na performance do sistema. Este mecanismo trata de ligar ou desligar *features* durante a execução de um treino do algoritmo UbiSOM. Pretende-se portanto analisar se este mecanismo, em conjunto com *datasets* que possuam um elevado número de *features*, reduz o tempo de execução de um treino.

Para testar a performance deste mecanismo, utilizou-se o *dataset* Iris como base, acrescentando mais *features* de valor aleatório compreendido entre 1 e 100. Estes *datasets* artificiais estão descritos na sub-secção 5.1.2 e vão ser utilizados as versões de 20, 40 e 80 *features*.

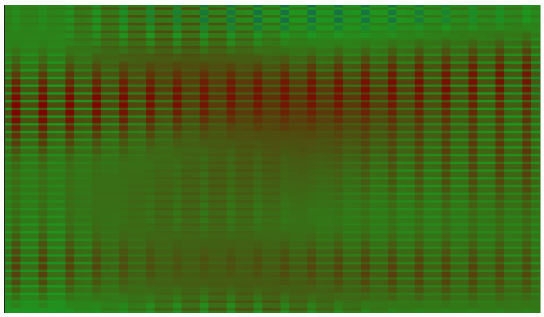
De forma a comparar a qualidade dos *datasets* artificiais do Iris com o *dataset* Iris "normal", estão ilustradas nas figuras 5.27a, 5.27b, 5.27c, 5.27d e 5.27e as Unified Distance Matrix relativas a cada um destes conjuntos de dados, tendo sido computadas sem recurso ao mecanismo de *feature switching* activo. Conforme se pode observar, à medida que se adicionam mais *features* ao mesmo *dataset*, a qualidade dos resultados vai deteriorando, sendo cada vez mais difícil de determinar as diferentes classes, que são possíveis de distinguir utilizando o *dataset* regular do Iris, conforme mostrado em 5.27d. Um detalhe referente a estas UMat é que à medida que existem mais *features*, desaparece uma das classes em comparação ao resultado do *dataset* Iris "regular"e parecendo que se adicionar mais do que 80 *features*, eventualmente a cor vermelha (indicativo de grande distância entre *clusters*) apodera-se totalmente do mapa, não havendo basicamente distinção entre *clusters* resultantes do algoritmo UbiSOM.

Cada *dataset* vai ser sujeito a 6000, 10000 e 19000 iterações do algoritmo UbiSOM e vão ser treinados num modelo sem *feature switching* e num modelo com este mecanismo activo. Como já foi prorferido nas sub-secções anteriores, utilizar um número máximo de iterações como 19000 justifica-se pelo facto do módulo do treino UbiSOM utilizado neste sistema estar limitado a este número de iterações. Os restantes valores utilizados para as iterações regem-se pelo facto de se querer o meio termo (aproximadamente 10000 iterações) e porque as 6000 iterações são exactamente 3000 iterações depois de se verificar a convergência do algoritmo, normalmente. A unidade de computação utilizada para esta experiência foi o *localhost*.

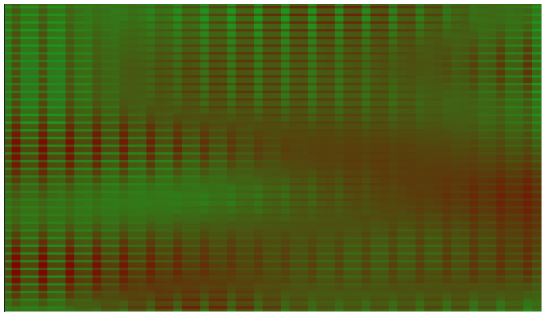
Os resultados estão expostos nos gráficos 5.28, 5.29 e 5.30, respectivamente, o *dataset* Iris com 20 *features*, *dataset* Iris com 40 *features* e o *dataset* Iris com 80 *features*.



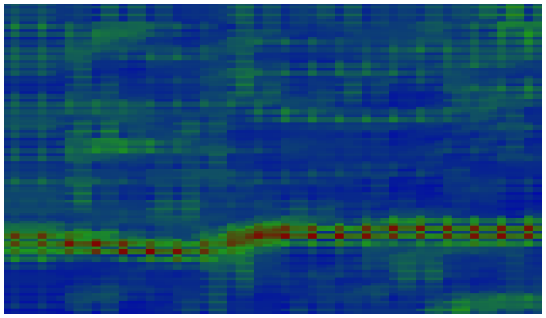
(a) Unified Distance Matrix referente a um treino do modelo UbiSOM utilizando o *dataset* Iris artificial com 20 *features*.



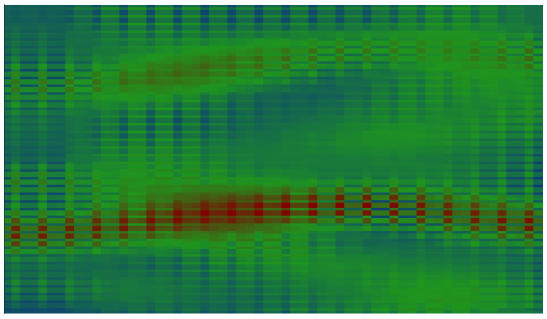
(b) Unified Distance Matrix referente a um treino do modelo UbiSOM utilizando o *dataset* Iris artificial com 40 *features*.



(c) Unified Distance Matrix referente a um treino do modelo UbiSOM utilizando o *dataset* Iris artificial com 80 *features*.



(d) Unified Distance Matrix referente a um treino do modelo UbiSOM utilizando o *dataset* Iris "regular".



(e) Unified Distance Matrix referente a um treino do modelo UbiSOM utilizando o *dataset* Iris artificial com 10 *features*.

Figura 5.27: Unified Distance Matrix das diferentes variações do conjunto de dados Iris. Treinos realizados com os parâmetros $\beta = 0.7$, $T = 2000$, $\sigma_i = 0.6$, $\sigma_f = 0.2$, $\eta_i = 0.1$, $\eta_f = 0.08$.

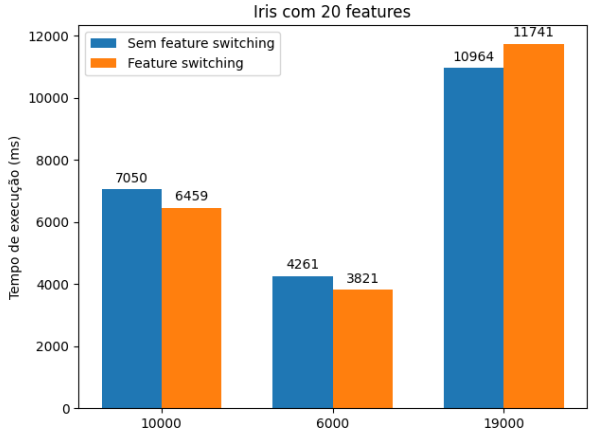


Figura 5.28: Tempos de execução dos treinos com *feature switching* e com este mecanismo desligado, para o *dataset* Iris com 20 *features*.

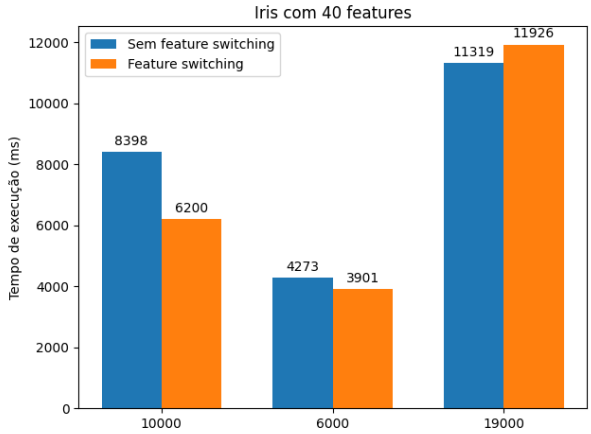


Figura 5.29: Tempos de execução dos treinos com *feature switching* e com este mecanismo desligado, para o *dataset* Iris com 40 *features*.

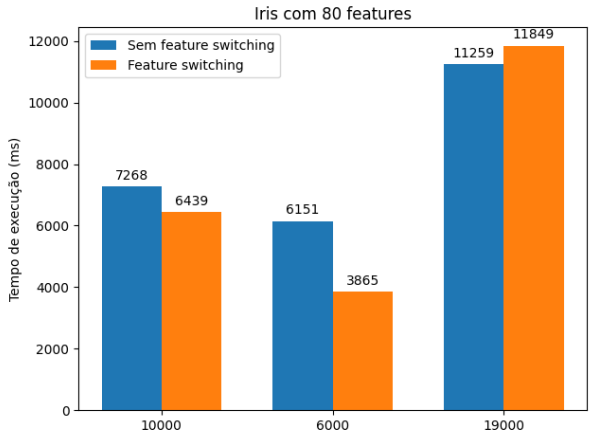


Figura 5.30: Tempos de execução dos treinos com *feature switching* e com este mecanismo desligado, para o *dataset* Iris com 80 *features*.

Os resultados foram semelhantes para todos os *datasets*, não havendo muita discrepância entre tempos de execução, nem entre os tempos relativos ao mecanismo de *feature switching* relativamente ao algoritmo UbiSOM "normal".

No caso das 6000 e 10000 iterações, os tempos de execução foram mais eficientes no caso do mecanismo de *feature switching* estar ligado e o caso contrário foi verificado nos treinos com 19000 iterações. Utilizando 6000 iterações no treino com o *dataset* de 80 *features*, o tempo de execução utilizando o mecanismo de *feature switching* foi 1.5x menor que no caso da não utilização deste. Uma descida acentuada fora também verificada no *dataset* com 40 *features*, utilizando 10000 iterações.

Efectivamente, verifica-se que para um número de iterações até 10000, a utilização do mecanismo de *feature switching* com *datasets* com um elevado número de *features*, reduz o tempo de execução face a um treino sem este mecanismo activo. Contudo, valores acima desse número de iterações prejudica utilizar este mecanismo, apesar da diferença entre tempos de execução não ser substancial, o tempo de execução é maior se o mecanismo estiver activo.

Os resultados verificados para as 19000 iterações podem ter sido gerados pelo facto de ter sido utilizado uma unidade de computação com menos recursos para lidar com tantas iterações e com um treino de um *dataset* com um elevado número de *features*. Estes tempos de execução elevados também podem ter sido gerados pelo facto de algumas *features* após serem desligadas, voltarem a entrar no treino e juntando o facto de estar a ser utilizado um elevado número de iterações, o algoritmo correspondente ao mecanismo de *feature switching* pode estar sempre a escolher as mesmas *features*, fazendo com que o tempo de execução aumente. Provavelmente, utilizando uma unidade de computação com mais recursos ou mesmo utilizando um GPU, o tempo de exeução utilizando o mecanismo de *feature switching* baixe consideravelmente o tempo levado a fazer um treino com estes condicionalismos.

5.4 Qualidade de *Software*

5.4.1 Facilidade de Utilização

Neste contexto, a aplicação MultiSOM implementada para esta dissertação é relativamente fácil de utilizar. Os parâmetros de lançamento do programa estão bem definidos e bem estruturados, com nomes que se assemelham bem à funcionalidade prevista por esses parâmetros. O lançamento do MultiSOM é feito a partir do ficheiro *JAR* gerado pelo IDE onde se fez as alterações à implementação da mesma aplicação.

Listagem 5.1: Exemplo de lançamento da aplicação MultiSOM.

```
1    java -jar multisom-cluster-v03.jar --execsReq 2 --filename iris.
        csv --randomParams --iteratorMode sequential --timerEnd 80000
        --iterations 6000 --member node6 --featSwi
```

Fazendo uma breve descrição dos parâmetros:

- *execsReq* é o número de treinos do modelo UbiSOM que se pretendem executar.
- *filename* é o *dataset* inicial que se pretende treinar.
- *randomParams* é o parâmetro responsável por gerar os parâmetros afectos ao modelo UbiSOM aleatoriamente. Em contrapartida, pode-se utilizar um ficheiro com as parametrizações (lido com o parâmetro *parametersFilename*). Para lançar o MultiSOM com o processo de treino visual, é necessário utilizar este ficheiro, com a estrutura apresentada em 3.2.
- *iteratorMode* é o parâmetro que determina se pretende fazer uma iteração sequencial ou aleatória do conjunto de dados inicial.
- *timerEnd* é o parâmetro que determina quando a aplicação pára (em milissegundos). Normalmente, é dado um valor que represente o final dos treinos definidos em *execsReq*.
- *iterations* é o parâmetro que define quantas iterações do modelo UbiSOM se pretendem executar.
- *member* é o parâmetro que guarda a *String* fornecida, para servir como uma etiqueta representante da unidade de computação utilizada para executar os treinos do modelo UbiSOM.
- *featSwi* é o parâmetro que determina se o mecanismo de seleção de *features* está activo. A sua ausência faz com que os treinos do modelo UbiSOM não executem com este mecanismo activo.

Em suma, a utilização desta aplicação é relativamente fácil, contudo, é necessário ter em conta a existência dos ficheiros de *script* e parametrizações. Como o processo de treino do modelo UbiSOM visual se tornou semi-automático com a inclusão desses ficheiros, é importante que o utilizador saiba o que o *script* está a fazer. Espera-se que com a documentação anexa [47] e auxílio desta dissertação, essas dúvidas possam ficar esclarecidas.

5.4.2 Complexidade da Implementação

Para efectuar a contagem do total de linhas de código utilizadas para a implementação utilizou-se um *plugin* do IntelliJ IDEA, IDE utilizado para realizar a implementação da aplicação apresentada nesta dissertação. Esse *plugin* é o *Statistic* e trata de contabilizar as linhas de código, as linhas de comentários, entre outras.

Esta aplicação contém cerca de 2545 linhas de código, constituídas por 187 métodos, 138 comentários dentro dos métodos e 22 comentários em bloco. O número de comentários é suficiente para descrever as funcionalidades que aparentam estar mais ambíguas.

Em suma, a documentação está bem efectuada. O número de linhas de código poderia de facto ser mais reduzido, mas para as funcionalidades que o sistema possui, pensa-se que é um valor satisfatório. Assim como o número de métodos: apesar de existirem 187 métodos, estão bem modularizados, efectuam uma tarefa apenas e são relativos a uma faixa de métodos referentes a cálculos do UbiSOM, *parsing* de parâmetros, entre outros. Ou seja, acredita-se que seja igualmente um número satisfatório de métodos.

Assim sendo, como o *software* está implementado, tem particularidades que influenciarão implementações futuras, facilitando o trabalho dessas mesmas implementações.

5.4.3 Testes à *Framework*

Para testar as novas componentes da aplicação MultiSOM de forma a que sejam permitidos múltiplos treinos em paralelo com múltiplas parametrizações, foi adoptada uma técnica de implementação chamada de *Test-Driven Development*. O conceito básico desta técnica é que todo o código que seja para ambiente de produção (ou versão final de uma aplicação) está implementado para resposta a um caso de teste. Efectivamente, comprova que para qualquer caso de uso de uma aplicação, a implementação funciona como esperado [31].

Foram implementados testes unitários para as seguintes componentes do sistema:

- Parametrizações geradas aleatoriamente
- Verificação ao processo de *feature switching*
- Teste à leitura/*parsing* dos *datasets* CSV
- Teste à instanciação de treinos UbiSOM
- Verificação do iterador sequencial e aleatório

Parametrizações geradas aleatoriamente Este teste tem o propósito de provar que as parametrizações geradas aleatoriamente regem-se pelos valores determinados na análise de sensibilidade de Bruno Silva [72]. A sua implementação está presente no anexo IV, listagem IV.1. Neste teste utiliza-se a biblioteca *Random* do Java e é-lhe atribuída uma *seed* de forma a tornar o processo aleatório determinístico. Após isso, é gerado o valor do parâmetro β e do parâmetro T do UbiSOM e é feita uma asserção no final se os valores estão dentro do intervalo estipulado. A geração dos parâmetros do modelo UbiSOM que está a ser testada está incluída na classe *ModelsLauncher*, aquando da execução do MultiSOM com geração de parâmetros aleatórios (secção 4.2).

Verificação ao processo de *feature switching* Neste teste é feita uma verificação ao processo de acções sobre *features*, nomeadamente testar se o método que realiza esta acção, liga/desliga a *feature* pretendida e realiza as tarefas pretendidas sobre as estruturas de dados que sustentam este mecanismo. A sua implementação encontra-se no anexo IV,

listagem IV.2. Este teste serve para verificar o comportamento do método *temporarilyExcludeFeature()* da classe *WrapperUbiSOM*, especificamente se desliga a *feature* correcta (secção 4.4).

Teste à leitura/*parsing* dos *datasets* CSV Este teste é um teste relativamente simples onde são invocados os métodos responsáveis pela leitura/*parsing* de um ficheiro CSV. É feita a leitura do *dataset* Iris base e os dados recolhidos por esta componente são depois comparados com as dimensões já conhecidas desse *dataset*. A implementação deste teste encontra-se no anexo IV, na listagem IV.3.

Essencialmente, neste teste verifica-se se o número de *features* está a ser correctamente lido pelos métodos *createCSV()* e *startCSVTable()*, inicialmente implementados para o MultiSOM e adaptados para a classe *MultiSOMNoDraw* (secção 4.3, primeira fase da figura 4.7).

Teste à instanciação de treinos UbiSOM De todos os testes presentes nesta secção, este teste é o que tem uma complexidade ligeiramente acrescida, dado que é feita uma comparação com o *output* do módulo implementado por Bruno Silva em [72], e como o módulo é privado então teve de ser feita uma implementação extra de uma classe que lê os *outputs* da consola de um determinado programa. A par disto, o teste é um teste simples de instanciação de modelos, onde é comprovado que o sistema ordena que o número correcto de modelos do UbiSOM seja instanciado. A implementação pode ser vista em IV, estando representado o código do teste na listagem IV.4 e o código da classe auxiliar que permite ler o *output* de um programa a partir da consola na listagem IV.5.

Essencialmente, procura-se entender se quando se parametriza na inicialização do MultiSOM um certo número de execuções (*–execsReq*), se o programa inicializa esse número de módulos UbiSOM para serem treinados (4.4).

Verificação do iterador sequencial e aleatório Este teste é o teste responsável por verificar se o iterador sequencial possuí os valores correctos para uma iteração sequencial de *inputs* de um *dataset* e se o iterador aleatório possuí valores contrários àqueles apresentados pelo iterador sequencial. O teste é muito simples, onde é repetido a forma como é calculado a posição da matriz do *dataset* que o iterador tem de recolher, já presente na implementação do MultiSOM. Os dois iteradores são testados pela mesma classe. O seu código está presente no anexo IV, listagem IV.6, estando este devidamente comentado para realçar a diferença entre os dois iteradores.

O teste é feito sobre a implementação do *parsing* dos conjuntos de dados, onde é feita a inicialização destes iteradores, nomeadamente o método *startCSVTable()*, presente na listagem VI.1, anexo VI.

5.4.4 Observações Finais e Trabalho Futuro

A implementação realizada para esta dissertação é extensível, podendo facilmente ser adaptadas novas funcionalidades ao sistema. O sistema encontra-se bem modularizado, com todas as componentes bem definidas, o que facilitará a implementação e manutenção futuras.

Os testes unitários comprovaram que a implementação está bem efectuada e que os métodos utilizados para executar os treinos do modelo UbiSOM (todas as funcionalidades envolvidas como o *parsing* dos conjuntos de dados, etc.) funcionam da forma esperada.

O tamanho da implementação em termos do número de linhas de código pode ser reduzida utilizando herança de classes em alguns contextos (e.g. *MultiSOM* e *MultiSOM-NoDraw*). Existe de igual forma uma repetição de código no *scheduler* do mecanismo de selecção de *features*, o qual poderá influenciar a organização da implementação.

A implementação encontra-se bem documentada, servindo igualmente para implementações futuras.

6

Conclusões e Trabalho Futuro

Um dos objectivos principal deste projecto era entender quais as melhores parametrizações para o modelo UbiSOM para certos conjuntos de dados e pode-se declarar que esse objectivo foi cumprido parcialmente. Através do treino de múltiplos mapas com múltiplas parametrizações em simultâneo, foi possível entender quais os melhores valores para os hiper-parâmetros do modelo UbiSOM (β, T, σ, η), tendo sido efectuado com sucesso um refinamento da análise de sensibilidade anteriormente determinada noutros trabalhos do género, conforme observado na secção 5.2.3. Os parâmetros β e T foram avaliados e apenas num dos *datasets* é que não foi possível encontrar melhores parametrizações, dentro do intervalo ótimo estabelecido por trabalhos anteriores. Quanto aos parâmetros σ e η, foram efectivamente encontrados as melhores combinações e variações destes parâmetros. Contudo, foram apenas testados alguns conjuntos de dados e efectivamente, não foram encontradas as melhores combinações num âmbito geral. Ainda assim, pensa-se ter encontrado variações que possam ser utilizadas em utilizações futuras do algoritmo UbiSOM, criando assim uma alternativa aos parâmetros admissíveis analisados por Bruno Silva em [72].

Outro dos objectivos seria avaliar a performance do algoritmo UbiSOM a executar em CPU, validando assim este algoritmo para conjunto de dados com um número de *features* muito elevado e para mapas de dimensão ainda maiores daqueles estudados ao longo deste projecto (as dimensões dos mapas nas várias experiências estava limitado a $20x40$). Para extrair de forma mais eficiente performance em questão dos treinos a executar em cada CPU, é preferível utilizar uma unidade de computação com bastantes *cores*/*threads*, invés de utilizar várias em simultâneo. Contudo, a partir de um certo elevado número de treinos, é necessário aliar mais unidades de computação para abater o *overhead* imposto por esse número elevado de treinos e portanto, está implícito que é preciso uma forma de distribuir os vários treinos de uma forma eficiente pelas unidades de computação disponíveis a certo momento.

Ainda assim, é favorável aliar a este facto, apenas executar treinos com um elevado número de iterações, de forma a utilizar os recursos das unidades computacionais da maneira mais eficiente, que no entanto pode ser contra-produtivo, dado que o sistema

está pronto para encontrar as melhores parametrizações para certos *datasets* e se não executamos o número suficiente de treinos, podemos não estar a avançar nesse contexto de encontrar as melhores parametrizações, mesmo com a análise de sensibilidade dos hiper-parâmetros concluída neste projecto. De facto, a migração destes treinos do modelo UbiSOM para unidades de computação GPU podem vir a resolver os dois problemas anteriormente mencionados, onde se pode executar um grande número de treinos em simultâneo com um grande número de iterações cada um.

No campo da selecção de *features*, o objectivo proposto seria encontrar as melhores *features* dentro de um conjunto de dados que possuísse um elevado número destas e encontrar aquelas que melhores resultados produziam, i.e. o conjunto de *features* que ficassem activas, geravam valores de erro de quantização/topológicos menores e assim reduzindo o conjunto de *features*, para maior facilidade na análise de resultados. Na secção 5.3.3 estão presentes os resultados afectos a esta experiência. Pode-se declarar que os resultados foram melhores do que o esperado, tenho o mecanismo encontrado praticamente todas as *features* do *dataset* Iris original. O único factor negativo foi o aumento do erro topológico, revelando que o lote de *features* encontradas podia ter sido ainda melhor. Contudo, a subida do erro não foi substancial, revelando que o algoritmo simples utilizado no mecanismo de selecção de *features* foi exemplar, contudo, são necessários mais experiências para determinar se o algoritmo utilizado no protocolo Multi-Armed Bandit foi o mais correcto e se igualmente se adapta a outros *datasets*.

Convém ainda referir que a performance do *feature switching* não teve impacto ao nível do aumento da performance do sistema, dado que a complexidade temporal no cálculo das distâncias e da fase de actualização do algoritmo UbiSOM não fora alterada com este mecanismo.

Relativamente ao estudo e alterações propostas feitas no algoritmo UbiSOM a executar em GPU, não foi possível acarretar tais experiências por falta de tempo. Infelizmente, com toda a dimensão já imposta por outras experiências propostas, não foi possível dedicar o tempo suficiente para propor alterações no algoritmo UbiSOM aquando da sua execução em GPU, tão pouco testar as alterações já feitas para que o modelo de execução contivesse estrutura para poder realizar o treinos simultâneos com múltiplas parametrizações. Naturalmente, com treinos a executar numa unidade de computação como o GPU, que possuí mais capacidade em termos de *cores* e velocidade de processamento (torna execuções mais rápidas por dividir o trabalho por vários *cores* e produzindo resultados mais rapidamente), iria reduzir os tempos de execução obtidos ao longo da secção 5.3, gerando *speedup* e podendo retirar mais conclusões tanto acerca da hiper-parametrização, tanto da selecção de *features*.

Não existiram grandes dificuldades na concepção/implementação do sistema aqui discutido nem no processo de engenharia reversa para poder entender o sistema, efectivamente comprovado pelos resultados esperados na parte da performance. No entanto, apesar dos resultados afectos à performance do sistema, o algoritmo UbiSOM aqui utilizado e como o sistema está definido pode ainda sofrer mais mudanças, especialmente

na parte referente à distribuição de treinos por várias unidades de computação, de forma a que exista uma divisão mais coerente de treinos por unidades de computação e nas partes paralelizáveis do algoritmo, algo que não se encontrava implementado neste sistema. Efectivamente, irá sempre haver uma redução dos tempos de execução de treinos do modelo UbiSOM executando os mesmos em GPU. Para isso se verificar, terá de ser feita uma reforma igualmente no CPU, dado que é a unidade de computação que delega os treinos para o GPU, influenciando os tempos de execução globais.

Trabalho Futuro Feitas as asserções sobre o trabalho realizado, existem algumas propostas quanto ao trabalho que futuramente se pode vir a verificar no projecto aqui desenvolvido:

- ***Frameworks* de computação distribuída:** Apesar do GPU servir como uma boa alternativa em termos de computação de alta performance, inicialmente idealizou-se o uso de uma *framework* que distribuísse os treinos do modelo UbiSOM por várias unidades de computação, utilizando algoritmos de *load balancing* refinados e preparados para o efeito de lidar com inúmeras unidades de computação e inúmeros treinos do modelo UbiSOM. Prevê-se que os ganhos não sejam muito substanciais, mas poderá ser um auxílio em futuras versões de aplicações que contém o algoritmo UbiSOM seja a executar em CPU ou em GPU, mesmo num contexto que o CPU apenas tenha de servir como delegado das tarefas por GPUs. Uma *framework* que poderá ser utilizada é a *Hazelcast*, que replica o sistema de Java Virtual Machine, responsável pela execução de programas Java, de forma a que vários grupos de execuções possam usar várias Java Virtual Machine, inclusive trocar mensagens entre eles.

- ***Framework* de paralelização em CPU:** Dado que a programação para ambientes de execução GPU ainda seja algo complexa e difícil de implementar e manter, uma das melhorias propostas é a adição de *frameworks* de paralelização na aplicação MultiSOM, especialmente nos cálculos referentes à distância euclidiana, a pesquisa da BMU, entre outros. Conforme declarado na secção 2.3.1, a raíz quadrada é um cálculo que não é tão eficiente para o GPU e portanto, vem confirmar que apesar de termos ganhos a nível temporal no GPU, estas pequenas mudanças podem surtir efeito na execução em CPU, podendo equiparar os resultados obtidos em cada unidade de computação.

- **Selecção de *features*:** Apesar dos resultados obtidos em 5.2.2 terem sido positivos e maioritariamente o mecanismo de selecção de *features* ter obtido o lote de *features* presente no *dataset* original utilizado para a experiência, pensa-se que ainda poderá ser refinado em termos da exploração de *features*. O algoritmo utilizado ($\epsilon - Greedy$) tem inerente uma chance de revisitar *features* que já haviam sido comprovadas que não maximizavam o valor de recompensa. A aleatoriedade do algoritmo faz com

que qualquer *feature* possa ser revisitada, o que não é o melhor cenário, piorando a execução do mecanismo de selecção de *features*. Uma possível melhoria seria utilizar o algoritmo *Upper-Confidence Bounds*, apesar de induzir um pouco de complexidade, dado que terá de se repetir um treino com as mesmas parametrizações várias vezes para ser analisado e para poder ser retirado ilações sobre o mesmo, de forma a utilizar os melhores intervalos para o algoritmo UCB (algoritmo que favorece a potencialidade de acções sobre *features*, invés de priorizar a maximização do valor estimado).

Outra implementação que poderá ser feita acerca do mecanismo de selecção de *features* é a paralelização do Multi-Armed Bandit Protocol. Se existirem múltiplos treinos do modelo UbiSOM a ocorrer em simultâneo em várias unidades de computação, poderá ser eficaz utilizar as várias instâncias para tirar conclusões acerca do *dataset* que esteja a ser treinado ao nível das melhores *features*. Efectivamente, trata-se das várias execuções comunicarem entre si os resultados obtidos, optimizando o mecanismo na medida em que poderá ser evitado explorar certas *features* à partida. Apesar de estar ilustrado uma arquitetura nesses moldes na figura 3.4, essa funcionalidade não foi testada (foi decidido utilizar uma versão sequencial utilizando apenas uma unidade de computação), dado que se acrescenta uma camada de complexidade se existirem várias unidades de computação a executar treinos em simultâneo, sendo necessário a existência de uma *framework* de troca de mensagens para a coordenação dos resultados.

- ***Caching* de resultados intermédios de cálculos:** Com um acréscimo do número de treinos a executar em simultâneo, cresce uma necessidade por *caching* no contexto de cálculos referentes ao algoritmo UbiSOM. Como os treinos a executar em certo momento partilham o mesmo *dataset*, e mesmo com parametrizações diferentes, alguns dos cálculos podem ser guardados em *cache*, de forma a que unidades de computação que necessitem de um resultado, não o tenham de calcular novamente, podendo ser usando por exemplo no cálculo da distância euclidiana. Isto encaixa na ideia geral de uma arquitetura distribuída do sistema que foi proposto neste projecto, especialmente na adição de uma *framework* que faça a tal distribuição de treinos por várias unidades de computação. Ainda assim, a mudança pode não ser directa, dado que possuir uma interdependência entre execução de treinos pode acrescentar uma camada de complexidade que pode não trazer benefícios gerais aos tempos de execução.

- **Aplicação do modelo UbiSOM em contexto financeiro:** Conforme referido em 2.4.2, o modelo UbiSOM foi comprovado como sendo uma ferramenta viável para avaliação financeira. Contudo, ainda existe inovações que possam ser realizadas nomeadamente a adaptação do modelo UbiSOM a conseguir detectar *crashes* financeiros, apesar de actualmente já ser capaz de analisar relatórios financeiros a

longo prazo. Para tal, prevê-se que seja necessário uma análise de sensibilidade aos parâmetros do modelo UbiSOM, adaptada a um contexto onde se utilize *datasets* financeiros, especialmente os conjuntos de dados relativos à análise fundamental.

- **Execução de *pipelines* de treino utilizando o modelo UbiSOM em GPU:** Conforme mencionado, os planos de trabalho desta dissertação incluíam inicialmente execuções paralelas com múltiplas parametrizações do modelo UbiSOM em Unidades de Processamento Gráfico. Infelizmente, não foi possível testar a arquitetura proposta em 3.3 e portanto, considera-se trabalho futuro em aplicar o modelo UbiSOM para execução nesse tipo de unidade de computação. Um dos factores que motivam a utilização desta metodologia de execução é o *speed-up* gerado pelos treinos do modelo UbiSOM serem executados em GPU. Esse factor poderia acelerar a análise de sensibilidade das parametrizações, gerando muitos mais mapas em simultâneo e assim, podendo excluir/aceitar à partida mais combinações de parâmetros. Outro factor é a computação de mapas de grande dimensão. Apesar da junção de várias unidades de computação (CPUs) ter sido comprovado como viável, a execução de mapas de grande dimensão possivelmente ascendiam a valores de tempos de execução muito elevados.

- **Trocar de visualização na aplicação MultiSOM:** Uma das ideias iniciais era a possibilidade da aplicação poder trocar a visualização do mapa resultante em qualquer momento dessa execução, seja por vontade do utilizador, seja porque um modelo UbiSOM que esteja a treinar em paralelo represente melhor uma distribuição subjacente a um certo conjunto de dados. Apesar de não estar implementado, a aplicação é extensível o suficiente para esta funcionalidade ser implementada em futuras versões da aplicação.

Bibliografia

[1] Adobe. *Photoshop graphics processor (GPU) card FAQ*. Acesso em Julho 2020. URL: https://helpx.adobe.com/photoshop/kb/photoshop-cc-gpu-card-faq.html (ver p. 19).

[2] M. O. Afolabi e O. Olude. "Predicting Stock Prices Using a Hybrid Kohonen Self Organizing Map (SOM)". Em: *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)* (2007), pp. 48–48 (ver p. 11).

[3] AMD. *Introduction to OpenCL Programming*. May, 2010 (ver pp. 19–21).

[4] *An Easy Introduction to CUDA C and C++*. Acesso em Junho 2020. URL: https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/ (ver pp. 18, 19).

[5] U. Aydonat et al. "An opencl™ deep learning accelerator on arria 10". Em: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 55–64. DOI: https://doi.org/10.1145/3020078.3021738 (ver p. 19).

[6] R. G. C. B. C. Hewitson. "Self-organizing maps: applications to synoptic climatology". Em: (2002), pp. 1–14. DOI: https://doi.org/10.3354/cr022013 (ver p. 11).

[7] E. Berglund e J. Sitte. "The parameterless self-organizing map algorithm". Em: *IEEE Transactions on Neural Networks* 17.2 (2006), pp. 305–316. DOI: http://doi.org/10.1109/TNN.2006.871720 (ver p. 11).

[8] J. Bhimani, M. Leeser e N. Mi. "Accelerating K-Means clustering with parallel implementations and GPU computing". Em: *2015 IEEE High Performance Extreme Computing Conference (HPEC)* (2015), pp. 1–6 (ver p. 7).

[9] J. Borrego. "Mapas Auto-Organizados Ubíquos em Unidades de Processamento Gráfico". Em: Rel. téc. Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologias, 2018 (ver pp. xix, 10, 12–17, 23, 25, 26, 46, 67–72).

[10] J. A. Bullinaria. "Self Organizing Maps: Fundamentals". Em: *Introduction to Neural Networks: Lecture 16*. 2004, pp. 1–15 (ver p. 8).

[11] H. M. N. Carrega Miguel Santos. "Progress in Artificial Intelligence". Em: (2021), pp. 609–621. DOI: 10.1007/978-3-030-86230-5 (ver pp. 2, 30).

[12] M. Carrega. "Analysis of Fundamental Investment Indicators Using SOMs". Em: Rel. téc. Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologias, 2020 (ver pp. 11, 42).

[13] B. Caulfield. *What's the Difference Between a CPU and a GPU?* 2009. URL: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/ (ver p. 18).

[14] T. Cavazos. "Using Self-Organizing Maps to Investigate Extreme Climate Events: An Application to Wintertime Precipitation in the Balkans". Em: *J. Climate* (2000), pp. 1718–1732. DOI: https://doi.org/10.1175/1520-0442(2000)013<1718:USOMTI>2.0.CO;2 (ver p. 11).

[15] cgorman. *tensorflow-som*. Acesso em Julho 2020. URL: https://github.com/cgorman/tensorflow-som (ver p. 23).

[16] D. M. Chitty. "A Data Parallel Approach to Genetic Programming Using Programmable Graphics Hardware". Em: (2007) (ver p. 18).

[17] G. Deboeck. "Financial applications of self-organizing maps". Em: *Neural Network World* (1998), pp. 213–241. DOI: https://doi.org/10.1109/HICSS.2007.441 (ver p. 11).

[18] G. Deboeck e T. Kohonen. *Visual Explorations in Finance with Self-Organizing Maps*. First. Springer Finance, 1998, pp. xxix–xlv. ISBN: 978-1-84996-999-4 (ver pp. 2, 7, 28).

[19] D. Demidov. *VexCL*. Acesso em Julho 2020. URL: https://github.com/ddemidov/vexcl (ver p. 19).

[20] D. Exterman. *CUDA vs OpenCL: Which to Use for GPU Programming*. URL: https://www.incredibuild.com/blog/cuda-vs-opencl-which-to-use-for-gpu-programming (ver p. 20).

[21] B. Fry. *Visualizing Data: Exploring and Explaining Data with the Processing Environment*. First. O'Reilly, 2008. ISBN: 0596514557 (ver p. 35).

[22] V. Garcia, E. Debreuve e M. Barlaud. "Fast k nearest neighbor search using GPU". Em: *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. IEEE. 2008, pp. 1–6. DOI: http://doi.org/10.1109/CVPRW.2008.4563100 (ver p. 18).

[23] Google. *Introduction to TensorFlow*. Acesso em Julho 2020. URL: https://developers.google.com/machine-learning/crash-course/first-steps-with-tensorflow/toolkit (ver p. 23).

[24] T. Google. *GPU support*. Acesso em Julho 2020. URL: https://www.tensorflow.org/install/gpu (ver p. 23).

[25] K. Group. *OpenCL Overview*. Acesso em 29 Junho 2020. URL: https://www.khronos.org/opencl/ (ver p. 19).

[26] G. Hamerly e C. Elkan. "Alternatives to the k-means algorithm that find better clusterings". Em: *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management* (2002), pp. 600–607. DOI: https://doi.org/10.1145/584792.584890 (ver pp. 5, 6).

[27] J. A. Hartigan e M. A. Wong. "A K-Means Clustering Algorithm". Em: (1975) (ver pp. 5, 6).

[28] S. Haykin. *Neural Networks and Learning Machines*. Third. Pearson Education, 2000, pp. 425–466. ISBN: 0-13-147139-2 (ver pp. 7–9).

[29] B. Hong-tao et al. "K-Means on Commodity GPUs with CUDA". Em: *2009 WRI World Congress on Computer Science and Information Engineering* (2009), pp. 651–655. DOI: https://doi.org/10.1109/CSIE.2009.491 (ver p. 7).

[30] A Hsu et al. "Visualising cluster separation with dynamic som tree". Em: *Proceedings of 6th International Conference on Soft Computing, Iizuka, Fukuoka, Japan*. 2000 (ver p. 11).

[31] IBM. *Test-Driven Development*. URL: https://www.ibm.com/garage/method/practices/code/practice_test_driven_development/ (ver p. 108).

[32] *Information Theory, Inference and Learning Algorithms*. First. University of Cambridge, 2003. Cap. An Example Inference Task: Clustering, pp. 284–292. ISBN: 9780521642989 (ver p. 6).

[33] Intel. *Dependency Flow Graph Example*. URL: https://software.intel.com/content/www/us/en/develop/documentation/tbb-documentation/top/intel-threading-building-blocks-developer-reference/flow-graph/overview/dependency-flow-graph-example.html (ver p. 47).

[34] P. P. Ippolito. *Machine Learning Visualization*. URL: https://towardsdatascience.com/machine-learning-visualization-fcc39a1e376a (ver p. 35).

[35] Y. Jia et al. "Caffe: Convolutional architecture for fast feature embedding". Em: *Proceedings of the 22nd ACM international conference on Multimedia*. 2014, pp. 675–678. DOI: https://doi.org/10.1145/2647868.2654889 (ver p. 23).

[36] J. Joseph e I. Indratmo. "Visualizing stock market data with self-organizing map". Em: *The Twenty-Sixth International FLAIRS Conference*. 2013 (ver pp. 2, 28).

[37] M. Kerrisk. *tmux(1) - Linux manual page*. URL: https://man7.org/linux/man-pages/man1/tmux.1.html (ver p. 93).

[38] L. Khacef, V. Gripon e B. Miramond. "GPU-Based Self-Organizing Maps for Post-labeled Few-Shot Unsupervised Learning". Em: *International Conference on Neural Information Processing*. Springer. 2020, pp. 404–416 (ver pp. 23, 24).

[39] K. Kiviluoto e P. Bergius. "Two-level self-organizing maps for analysis of financial statements". Em: *1998 IEEE International Joint Conference on Neural Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No. 98CH36227)*. Vol. 1. IEEE. 1998, pp. 189–192. DOI: https://doi.org/10.1109/ijcnn.1998.682260 (ver pp. 28–30).

[40] T. Kohonen. "Self-Organizing Maps". Em: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480. DOI: https://doi.org/10.1109/5.58325 (ver pp. 8–10).

[41] T. Kohonen. "Self-Organizing Maps". Em: (2000), pp. 70–75, 106–127 (ver pp. 8, 10).

[42] L. Krippahl. "Feature Extraction". Em: *Machine Learning: Chapter 16*. Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologias, 2019, pp. 142–143 (ver p. 7).

[43] Y. Li et al. "Speeding up k-Means algorithm by GPUs". Em: *Journal of Computer and System Sciences 79* 2 (2013), pp. 216–229. DOI: https://doi.org/10.1016/j.jcss.2012.05.004 (ver p. 7).

[44] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (ver p. ii).

[45] S. A. Manavski. "CUDA Compatible GPU As An Efficient Hardware Accelerator For AES Cryptograhpy". Em: *2007 IEEE International Conference on Signal Processing and Communications (ICSPC 2007)* (2007), pp. 65–68. DOI: ? (ver pp. 18, 19).

[46] S. A. Manavski e G. Valle. "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment". Em: *BMC bioinformatics* 9.S2 (2008), S10. DOI: http://doi.org/10.1186/1471-2105-9-S2-S10 (ver p. 18).

[47] N. Marques e R. Pinheiro. *Implementação do MultiSOM*. URL: https://drive.google.com/file/d/1q-gE93fFuorSBVkrHW0-EtKezkHKJOYY/view?usp=sharing (ver p. 107).

[48] N. C. Marques, B. Silva e H. Santos. "An interactive interface for multi-dimensional data stream analysis". Em: *2016 20th International Conference Information Visualisation (IV)*. IEEE. 2016, pp. 223–229 (ver pp. 42, 45, 67).

[49] R. Marques et al. "Algorithmic Skeleton Framework for the Orchestration of GPU Computations". Em: ago. de 2013. DOI: 10.1007/978-3-642-40047-6_86 (ver p. 23).

[50] S. Mathew e P. Joy. "Ultra fast SOM using CUDA". Em: *NeST-NVIDIA Center for GPU computing, hpc@ nestgroup. net* (2010) (ver pp. 24, 25).

[51] P. M. Mele e D. E. Crowley. "Application of self-organizing maps for assessing soil biological quality". Em: *Agriculture, Ecosystems Environment* 126 (2008), pp. 139–152. DOI: https://doi.org/10.1016/j.agee.2007.12.008 (ver p. 11).

[52] J. Meyer. *Programming 101: The How and Why of Programming Revelead Using the Processing Programming Language*. Apress, 2018. ISBN: 9781484236963 (ver p. 35).

[53] F. C. Moraes et al. "Parallel high dimensional self organizing maps using CUDA". Em: *2012 Brazilian Robotics Symposium and Latin American Robotics Symposium*. IEEE. 2012, pp. 302–306. DOI: https://doi.org/10.1109/sbr-lars.2012.56 (ver pp. 24, 25).

[54] J. I. Mwasiagi. *Self Organizing Maps: Applications and Novel Algorithm Design*. 2011 (ver p. 10).

[55] J. Nickolls e W. J. Dally. "The GPU Computing Era". Em: (2010) (ver p. 19).

[56] Nvidia. *CUDA In Action*. Acesso em Junho 2020. URL: https://developer.nvidia.com/cuda-action-research-apps (ver p. 19).

[57] Nvidia. *CUDA Zone | Nvidia Developer*. Acesso em Maio 2020. URL: https://developer.nvidia.com/cuda-zone (ver p. 18).

[58] J. M. D. P. O. A. Postolache P. M. B. S. Girao e H. M. G. Ramos. "Self-organizing maps application in a remote water quality monitoring system". Em: *IEEE Transactions on Instrumentation and Measurement* 54 (2005), pp. 322–329 (ver p. 11).

[59] Octopus. *Octopus | Main Page*. Acesso em Julho 2020. URL: https://octopus-code.org/wiki/Main_Page (ver p. 19).

[60] *OpenCL Programming Guide*. First. Addison-Wesley Professional, 2011. Cap. Conceptual Foundations of OpenCL, pp. 3–29. ISBN: 9780321749642 (ver pp. 19–22).

[61] Oracle. *Timer (Java Platform SE 7)*. Acesso a 18 Setembro 2021. URL: https://docs.oracle.com/javase/7/docs/api/java/util/Timer.html (ver p. 60).

[62] G. C. Panosso. "Análise do Mercado Financeiro baseada em Análise Técnica com Self-Organizing Maps". Em: Rel. téc. Universidade Nova de Lisboa, Instituto Superior de Estatística e Gestão de Informação, 2011 (ver p. 30).

[63] P. Patil. *What is Exploratory Data Analysis?* URL: https://towardsdatascience.com/exploratory-data-analysis-8fc1cb20fd15 (ver p. 5).

[64] D. J. Peddie. "Is it Time to Rename the GPU?" Em: (2012) (ver p. 18).

[65] E. Peters e A. Savakis. "SVM with OpenCL: High performance implementation of support vector machines on heterogeneous systems". Em: *2015 IEEE International Conference on Image Processing (ICIP)*. 2015, pp. 4322–4326. DOI: http://doi.org/10.1109/ICIP.2015.7351622 (ver p. 19).

[66] Processing. *Welcome to Processing!* URL: https://processing.org/ (ver p. 35).

[67] S. Raschka. *What is the main difference between TensorFlow and scikit-learn?* Acesso em Julho 2020. URL: https://sebastianraschka.com/faq/docs/tensorflow-vs-scikitlearn.html (ver p. 23).

[68] A. Richardson e F. A. Shillington. "Using self-organizing maps to identify patterns in satellite imagery". Em: *Progress in Oceanography* 59 (2003), pp. 223–239. DOI: https://doi.org/10.1016/j.pocean.2003.07.006 (ver p. 11).

[69] D. Russo et al. "A tutorial on thompson sampling". Em: *arXiv preprint arXiv:1707.02038* (2017) (ver p. 35).

[70] *OpenCL Basics*. Faculty Of Eng. And Math., Bielefeld University of Applied Sciences, pp. 1–29 (ver pp. 19, 22).

[71] N. I. of Science e Techonology. *What is EDA?* URL: https://www.itl.nist.gov/div898/handbook/eda/section1/eda11.htm (ver p. 5).

[72] B. Silva. "Exploratory Cluster Analysis from Ubiquitous Data Streams using Self-Organizing Maps". Em: Tese de Doutoramento, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologias, 2016 (ver pp. xix, 2, 9, 12–17, 27, 39, 49, 54, 66–71, 77, 78, 80, 81, 83, 85, 87, 108, 109, 111).

[73] B. Silva e N. C. Marques. "The ubiquitous self-organizing map for non-stationary data streams". Em: *Journal of Big Data* 2.1 (2015), p. 27. DOI: https://doi.org/10.1186/s40537-015-0033-0 (ver pp. 12, 13).

[74] J. Sirotković, H.Dujmić e V. Papić. "K-means image segmentation on massively parallel GPU architecture". Em: *2012 Proceedings of the 35th International Convention MIPRO* (2012), pp. 489–494 (ver p. 7).

[75] M. Steuwer, P. Kegel e S. Gorlatch. "Skelcl-a portable skeleton library for high-level gpu programming". Em: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE. 2011, pp. 1176–1182. DOI: http://doi.org/10.1109/IPDPS.2011.269 (ver p. 23).

[76] A. Trevino. *Introduction to K-means Clustering*. 2016. URL: https://blogs.oracle.com/datascience/introduction-to-k-means-clustering (ver p. 6).

[77] J. W. Tukey. *The Future of Data Analysis*. Princeton University e Bell Telehpone Laboratories, 1967 (ver p. 5).

[78] C. University. *Introduction to GPGPU and CUDA Programming: Memory Coalescing*. Acesso em Julho 2020. URL: https://cvw.cac.cornell.edu/GPU/coalesced (ver p. 25).

[79] L. Weng. "The Multi-Armed Bandit Problem and Its Solutions". Em: *lilianweng.github.io/lil-log* (2018). URL: http://lilianweng.github.io/lil-log/2018/01/23/the-multi-armed-bandit-problem-and-its-solutions.html (ver pp. 2, 32, 34, 35).

[80] B. Wilson. *The Machine Learning Dictionary*. Acesso em Novembro 2019. URL: https://www.cse.unsw.edu.au/~billw/mldict.html#lrate (ver p. 9).

[81] P. Wittek e S. Darányi. "A GPU-Accelerated Algorithm for Self-Organizing Maps in a Distributed Environment." Em: *20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges, Belgium.* 2012 (ver pp. 24, 25).

I

Anexo 1: Instanciação de Modelos

Neste anexo está presente a implementação referente à instanciação dos múltiplos treinos em paralelo no MultiSOM.

Listagem I.1: Método responsável pela instanciação dos vários modelos UbiSOM na aplicação MultiSOM

```
1  package multiSOM;
2
3  import commandLine.ParameterController;
4  import multiSOM.MultiSOMNoDraw;
5  import processing.core.PApplet;
6  import ubiSOM.Model;
7  import ubiSOM.ModelCoordinator;
8
9  import java.nio.charset.Charset;
10 import java.nio.charset.StandardCharsets;
11 import java.util.ArrayList;
12 import java.util.List;
13 import java.util.Random;
14 import java.util.concurrent.ExecutorService;
15 import java.util.concurrent.Executors;
16
17 public class ModelsLauncher extends PApplet {
18     public ParameterController pc;
19     private List<Integer> itsToSwitch; // Iterations in which the MABP
           is activated
20     private List<Integer> featuresToSwitch; // List containing the
          status of each feature
21     private boolean featureSwitching;
22
23     public static void main(String[] args) {
24         // Temp hardcoded indexes
25         String[] modelIndexes = {"0", "1", "2", "3"};
```

```
26          String[] featureSwitching = {"true", "true", "false", "false"
                };
27
28          List<MultiSOM> multiSOMList = new ArrayList<>();
29          List<MultiSOMNoDraw> noDrawList = new ArrayList<>();
30
31          String[] appletArgs = new String[] { "multiSOM.MultiSOM" };
32          System.out.println("MultiSOM." +
33                  "\n(c) nmm@fct.unl.pt " +
34                  "Please check Readme and Licence for information on
                        citation and usage.");
35
36          ParameterController pc = new ParameterController();
37          pc.readUbiSOMParameters();
38          ModelCoordinator modelCoordinator = new ModelCoordinator(pc);
39
40          // Create all multiSOMs
41          for(int index = 0; index < pc.getNoOfExecutions(); index++) {
42              // The models which don't have no MS UI are created after
                    the MS UI ones
43              MultiSOM ms = new MultiSOM(); // Create a MultiSOM UI
44
45              ms.width = 1360;
46              ms.height = 768;
47
48              // Adding to both lists (coordinator and multi som window)
49              multiSOMList.add(ms);
50              modelCoordinator.addToMSList(ms);
51          }
52
53          for(int index = 0; index < pc.getNumOfModelsNoGUI(); index++)
                {
54              // Random name generator
55              byte[] array = new byte[7]; // length is bounded by 7
56              new Random().nextBytes(array);
57              String nameForModel = new String(array, StandardCharsets.
                    UTF_8);
58              MultiSOMNoDraw ms = new MultiSOMNoDraw(nameForModel,
                    modelCoordinator, pc, index);
59              noDrawList.add(ms);
60              modelCoordinator.addToModelNoGUIList(ms);
61          }
62
63          modelCoordinator.setupMultipleExecs(); // Populating
                executions list
64
```

```
65          // Start all the MultiSOM executions
66          int auxIndex = 0;
67          for(MultiSOM ms : multiSOMList) {
68              ms.setup_my_model_coordinator(modelCoordinator);
69              ms.setParameterController(pc);
70
71              if(auxIndex == 1) {
72                  ms.X_MONITOR_POSITION = 0;
73                  ms.Y_MONITOR_POSITION = 0;
74              }
75
76              ms.currModelIndex = auxIndex++;
77              ms.setFeatureSwitchingValue(false);
78              PApplet.runSketch(appletArgs, ms);
79          }
80
81          if(noDrawList.size() > 0) {
82              ExecutorService executorService = Executors.
                    newFixedThreadPool(noDrawList.size());
83              for (int i = 0; i < noDrawList.size(); i++) {
84                  final int count = i;
85                  executorService.submit(() -> {
86                      noDrawList.get(count).getCSV("exemplos/chain.csv")
                            ;
87                      noDrawList.get(count).startTraining();
88                  });
89              }
90          }
91      }
92  }
```

Listagem I.2: Script de inicialização da aplicação MultiSOM.

```
1  # multiSOM : comandos iniciais
2
3  print Configurar CSV:
4
5  set CSV exemplos/iris.csv
6
7  colordim 0
8  print CSV lido, valores:
9
10 get CSV
11
12 print Configurar visualizacao treino
13
14 #transformx 10000 737 2556 7
```

```
#transformy 0 5170 1678 10000
transformx 1653 577 85 27
transformy 0 619 4334 10000

print Ligar grelha SOM
set SOMGrid on

print Treinar 3000 iteracoes vel inicial.
step 19000
set framerate 60

print Gravar resultado apos treino.

writePrototypes

print Visualizar

set zoomIn 0

version

print Ativar modo de pausa (P - Continuar)
set pause on
step 1
```

II

Anexo 2: Recolha e análise da parametrização

Neste anexo estão descritos os métodos referentes à recepção e *parsing* dos parâmetros para cada um dos treinos requisitados.

Listagem II.1: Método para o *parsing* dos parâmetros para os vários treinos a partir de um ficheiro.

```
1  public void readUbiSOMParameters() {
2          try {
3              File parameters = new File("parameters_data_aux.data");
4              Scanner in = new Scanner(parameters);
5              Pattern pn = Pattern.compile(" ");
6              String typeExec = "CPU_Simple",
7                      etas, sigmas, betas,
8                      slidingWindowValues, featuresSwitchValues,
                           iterationsToSwitch;
9              String noOfExecsAux = " ";
10             int noOfExecs = 0, noOfExecsNoGUI = 0; // 1 is default
                   value
11             List<Float> listEtas = new ArrayList<>(),
12                     listSigmas = new ArrayList<>(),
13                     listBetas = new ArrayList<>();
14             List<Integer> listSlidingWindow = new ArrayList<>(),
15                           featureSwitchList = new ArrayList<>(),
16                           itsToSwitchList = new ArrayList<>();
17             int countAux = 0;
18             while(in.hasNextLine()) {
19                 typeExec = in.nextLine(); // Type of exec
20
21                 noOfExecsAux = in.nextLine(); // Number of executions
                       with GUI
22                 Scanner line = new Scanner(noOfExecsAux);
23                 line.useDelimiter("/");
```

```
24                  while(line.hasNext()) {
25                      if(countAux == 0) {
26                          noOfExecs = Integer.parseInt(line.next());
27                          countAux++;
28                      } else {
29                          String aux = line.next();
30                          noOfExecsNoGUI = Integer.parseInt(aux);
31                      }
32                  }
33
34                  etas = in.nextLine(); // List of eta values
35                  sigmas = in.nextLine(); // List of sigma values
36                  betas = in.nextLine(); // List of beta values
37                  slidingWindowValues = in.nextLine(); // List of
                        sliding window values
38                  featureSwitchingMode = Boolean.parseBoolean(in.
                        nextLine()); // Flag to activate/deactivate feature
                         switching
39                  featuresSwitchValues = in.nextLine(); // Features to
                        turn off
40                  iterationsToSwitch = in.nextLine(); // Iterations to
                        switch off
41
42                  if(etas.isEmpty() || sigmas.isEmpty() || betas.isEmpty
                        () || slidingWindowValues.isEmpty()
43                          || typeExec.isEmpty() || featuresSwitchValues.
                              isEmpty() || iterationsToSwitch.isEmpty())
                              {
44                      System.err.println("Not all values are set. Write
                            them, please." +
45                              "For help. check parameters.help file on
                                  the main directory.");
46                      throw new NullPointerException();
47                  }
48
49                  listEtas = pn.splitAsStream(etas)
50                          .map(Float::valueOf)
51                          .collect(Collectors.toList());
52
53                  listSigmas = pn.splitAsStream(sigmas)
54                          .map(Float::valueOf)
55                          .collect(Collectors.toList());
56
57                  listBetas = pn.splitAsStream(betas)
58                          .map(Float::valueOf)
59                          .collect(Collectors.toList());
```

```
60
61                  listSlidingWindow = pn.splitAsStream(
                        slidingWindowValues)
62                          .map(Integer::valueOf)
63                          .collect(Collectors.toList());
64
65                  featureSwitchList = pn.splitAsStream(
                        featuresSwitchValues)
66                          .map(Integer::valueOf)
67                          .collect(Collectors.toList());
68
69                  itsToSwitchList = pn.splitAsStream(iterationsToSwitch)
70                          .map(Integer::valueOf)
71                          .collect(Collectors.toList());
72              }
73              in.close();
74
75              // Fill the parameter controller values.
76              setTypeOfExec(typeExec);
77              setNumberOfExecutions(noOfExecs);
78              setNumberOfModelsNoGUI(noOfExecsNoGUI);
79              setupEtasList(listEtas);
80              setupSigmasList(listSigmas);
81              setupBetasList(listBetas);
82              setupSliWinList(listSlidingWindow);
83              setupFeaturesList(featureSwitchList);
84              setupItsToSwitch(itsToSwitchList);
85
86          } catch(FileNotFoundException f) {
87              f.printStackTrace();
88          }
89      }
```

Listagem II.2: Método para análise dos parâmetros e correspondente interligação a cada um dos treinos que se pretendem inicializar.

```
1  public void setupMultipleExecs() {
2      System.out.println("[INFO] Setting up " + pc.getNoOfExecutions() +
           " UbiSOM executions."); // pc is the instance of parameter
          controller
3      etas = pc.getEtasList();
4      sigmas = pc.getSigmasList();
5      betas = pc.getBetasList();
6      sliWinSizes = pc.getSlidingWindowSzList();
7
8      // Loop control variables.
9      int index = 0;
```

```
10      int indexBetas = 0;
11
12      float etaI, sigmaI, etaF, sigmaF, beta;
13      int windowSize = 0;
14      int controlCycle = 0;
15      int countModelsWoDraw = 0;
16
17      for(int a = 0; a < pc.getNoOfExecutions()+pc.getNumOfModelsNoGUI()
            ; a++) {
18
19        etaI = etas.get(index);
20        sigmaI = sigmas.get(index);
21
22        etaF = etas.get(++index);
23        sigmaF = sigmas.get(index);
24
25        index++;
26
27        beta = betas.get(controlCycle);
28        windowSize = sliWinSizes.get(controlCycle);
29        controlCycle++;
30
31        ubiSOM.WrapperUbiSOM ubisomInst;
32        MultiSOM aux = null;
33        MultiSOMNoDraw aux_;
34
35        if(a < pc.getNoOfExecutions()) {
36          try {
37            aux = multiSOMList.get(a);
38          } catch (NullPointerException e) {
39            e.printStackTrace();
40          }
41        }
42
43        if(aux != null) {
44          ubisomInst = new ubiSOM.WrapperUbiSOM(aux,
45              etaI, etaF,
46              sigmaI, sigmaF,
47              beta, windowSize);
48          executionsList.add(ubisomInst);
49        } else {
50          aux_ = noDrawList.get(countModelsWoDraw);
51
52          ubisomInst = new ubiSOM.WrapperUbiSOM(
53              aux_,
54              etaI, etaF,
```

```
55              sigmaI, sigmaF,
56              beta, windowSize);
57
58          execsNoGUI.add(ubisomInst);
59
60          countModelsWoDraw++;
61        }
62      }
63    }
```

III

Anexo 3: *Scheduling* para o protocolo *Multi-Armed Bandit*

Implementação relativa à temporização do protocolo *Multi-Armed Bandit*, integrado no mecanismo de selecção de *features*.

Listagem III.1: Classe que alberga a tarefa referente ao *switching* das *features*

```
1  package multiSOM;
2
3  import ubiSOM.Feature;
4
5  import java.io.Serializable;
6  import java.util.*;
7  import java.util.concurrent.ThreadLocalRandom;
8
9  public class FeatureSwitchingTask extends TimerTask implements
       Serializable {
10     private static final long serialVersionUID = 1085817105782061832L;
11
12     public static int NEGATIVE_ONE = -1;
13
14     private MultiSOM ms;
15     private MultiSOMNoDraw noDraw;
16
17     private double secondInstance;
18     private int axis;
19     private int currModelIndex;
20
21     private int exploreCounter;
22     private int exploitCounter;
23
24     public List<Feature> listFeatures;
25
26     private List<Feature> baseMABP;
```

```
27
28      int indexFeature;
29      private Random random = new Random();
30
31      int actions = 0;
32
33      public static final float EPSILON_VALUE = 0.1f;
34      public static final double INCERTAINTY_QUANTIZATION = 1.2d;
35
36      public static final int MAP_EVOLUTION_TIMEOUT = 100;
37
38      public FeatureSwitchingTask(MultiSOM ms, int axis, int
            currModelIndex) {
39          this.ms = ms;
40          this.axis = axis;
41          this.currModelIndex = currModelIndex;
42      }
43
44      public FeatureSwitchingTask(MultiSOMNoDraw ms) {
45          this.noDraw = ms;
46
47          if(ms.getJSONMABPFile() == null) {
48              this.listFeatures = new ArrayList<>(ms.getNumberFeatures()
                    );
49
50              try {
51              List<String> tmp = new ArrayList<>(ms.getCSVHeader());
52              for(String a : tmp) {
53                  Feature feature = new Feature(a);
54                  listFeatures.add(feature);
55              }
56              } catch(Exception e) {
57                  e.printStackTrace();
58              }
59          } else {
60              this.listFeatures = new ArrayList<>(ms.getJSONMABPFile().
                    size());
61              for(Feature f : ms.getJSONMABPFile()) {
62                  try {
63                      this.listFeatures.add(f.clone());
64                  } catch(CloneNotSupportedException e) {
65                      e.getStackTrace();
66                  }
67              }
68          }
69      }
```

```
70
71      public void saveErrorReport(MultiSOMNoDraw noDraw) {
72          noDraw.errorReport.add("############### Model @ "
73                  + Arrays.toString(noDraw.getModel().getParameters()) +
                        " Error Report ###############");
74          noDraw.errorReport.add("Feature turned off: " + noDraw.
                getColumnByAxis(indexFeature));
75          noDraw.errorReport.add("Step: " + noDraw.getModelIteration());
76          noDraw.errorReport.add("Topological Error: " + noDraw.getModel
                ().getMeanErrors(noDraw.getModel().getCurrBMU())[0] * 100);
77          noDraw.errorReport.add("Quantization Error: " + noDraw.
                getModel().getMeanErrors(noDraw.getModel().getCurrBMU())[1]
                 * 100);
78          noDraw.errorReport.add("###############

                ###############\n");
79      }
80
81      public void saveErrorVariation(MultiSOMNoDraw noDraw, Feature f,
            double delta, double firstInstance, double secondInstance) {
82          noDraw.errorReport.add("############### Second measure and
                variation @ "
83                  + f.getNameFeature() + " ###############");
84          noDraw.errorReport.add("Feature chosen: " + f.getNameFeature()
                );
85          noDraw.errorReport.add("Step: " + noDraw.getModelIteration());
86          noDraw.errorReport.add("Topological Error: " + noDraw.getModel
                ().getMeanErrors(noDraw.getModel().getCurrBMU())[0] * 100);
87          noDraw.errorReport.add("First mesaure QE: " + firstInstance);
88          noDraw.errorReport.add("Second mesaure QE: " + secondInstance)
                ;
89          noDraw.errorReport.add("Gain/Loss: " + delta);
90          noDraw.errorReport.add("###############

                ###############\n");
91      }
92
93      public float randomInRange(float min, float max) {
94          float range = max - min;
95          float scaled = random.nextFloat() * range;
96          float shifted = scaled + min;
97          return shifted;
98      }
99
100     public List<Feature> getListFeatures() {
101         return listFeatures;
```

```
102     }
103
104     public int randomInRangeInteger(int min, int max) throws
            NoSuchElementException {
105         Random random = new Random();
106         return random.ints(min, max)
107                 .findFirst()
108                 .getAsInt();
109     }
110
111     @Override
112     public void run() {
113         try {
114             Thread.sleep(10); // Esperar por medi  es dos erros das
                    outras threads
115         } catch(InterruptedException e) {
116             e.printStackTrace();
117         }
118         if(ms != null) {
119             ms.temporarilyExcludeFeature(axis, currModelIndex);
120         } else if(noDraw != null) {
121             saveErrorReport(noDraw);
122             actions++;
123             double epsilonValue;
124             double prob = randomInRange(0.0f, 1.0f); // Generate
                    probability
125             double currentQuantizationValue =
126                     noDraw.getModel().getMeanErrors(noDraw.getModel().
                            getCurrBMU())[1] * 100;
127             Feature chosenFeature;
128             double estValue;
129             int binaryIndicator = -1;
130
131             epsilonValue = 1.0f / actions; // Deca mento do valor de
                    epsilon ao longo das invoca  es deste algoritmo
132
133             if(prob < epsilonValue) {
134                 System.err.println("EXPLORE");
135                 exploreCounter++;
136
137                 indexFeature = randomInRangeInteger(0, noDraw.getModel
                        ().getNumberFeatures());
138
139                 chosenFeature = listFeatures.get(indexFeature);
140                 noDraw.getModel().temporarilyExcludeFeature(
                        indexFeature);
```

```
141
142                 binaryIndicator = calculateReward(
                        currentQuantizationValue, chosenFeature);
143             } else {
144                 System.err.println("EXPLOIT");
145                 exploitCounter++;
146
147                 chosenFeature = Collections.max(listFeatures, new
                        Comparator<>() {
148                     @Override
149                     public int compare(Feature t1, Feature t2) {
150                         return Double.compare(t1.getEstimatedValue(),
                                t2.getEstimatedValue()); // Retornar a
                                feature com valor estimado m ximo
151                     }
152                 });
153
154                 if(noDraw.getJSONMABPFile() != null) {
155                     for(Feature f : listFeatures) {
156                         if(f.getNameFeature().equals(chosenFeature.
                                getNameFeature())) {
157                             indexFeature = listFeatures.indexOf(f);
158                             break;
159                         }
160                     }
161                 } else
162                     indexFeature = listFeatures.indexOf(chosenFeature)
                            ;
163
164                 System.out.println("Index of feature to turn off: " +
                        indexFeature);
165                 noDraw.getModel().temporarilyExcludeFeature(
                        indexFeature);
166                 binaryIndicator = calculateReward(
                        currentQuantizationValue, chosenFeature);
167             }
168
169             listFeatures.get(indexFeature).incrementChosenValue(); //
                    Incrementa  o do valor das vezes que uma certa
                    feature foi escolhida
170             listFeatures.get(indexFeature).setCurrBinaryIndicator(
                    binaryIndicator);
171             int freq = listFeatures.get(indexFeature).getChosen();
172
173             if(binaryIndicator == 1) {
174                 if(actions > 1)
```

```
175                     listFeatures.get(indexFeature).
                            setLastEstimatedValue(listFeatures.get(
                            indexFeature).getEstimatedValue());
176                 estValue = (1.D / freq) * listFeatures.get(
                        indexFeature).getRewardCounter();
177             } else {
178                 estValue = (1.D / freq) * listFeatures.get(
                        indexFeature).getRewardCounter();
179                 if(estValue == 0.0f || listFeatures.get(indexFeature).
                        getChosen() < 3)
180                     estValue = 1.0f; // Se a feature foi escolhida
                            menos de 3 vezes e nunca gerou recompensa,
                            deixar o seu valor estimado com o valor por
                            omiss o
181             }
182
183             listFeatures.get(indexFeature).setEstimatedValue(estValue)
                    ;
184         }
185     }
186
187     public int calculateReward(double currentQuantizationValue,
            Feature f) {
188         double delta;
189         int binaryIndicator;
190
191         // Esperar algum tempo para o mapa evoluir e analisar a
                varia  o do erro de quantiza  o
192         try {
193             Thread.sleep(MAP_EVOLUTION_TIMEOUT);
194             secondInstance = noDraw.getModel().getMeanErrors(noDraw.
                    getModel().getCurrBMU())[1] * 100;
195         } catch (InterruptedException e) {
196             e.printStackTrace();
197         }
198
199         delta = secondInstance - currentQuantizationValue;
200
201         saveErrorVariation(noDraw, f, delta, currentQuantizationValue,
                secondInstance);
202
203         delta *= NEGATIVE_ONE;
204
205         if(delta > 0.d) {
206             binaryIndicator = 1;
207             f.incrementRewardCounter();
```

```
208         } else {
209             binaryIndicator = 0;
210         }
211 
212         return binaryIndicator;
213     }
214 
215     public int getExploitCounter() {
216         return exploitCounter;
217     }
218 
219     public int getExploreCounter() {
220         return exploreCounter;
221     }
222 
223     public List<Feature> getBaseMABP() {
224         return baseMABP;
225     }
226 }
```

Listagem III.2: Classe responsável pela redução dos valores estimados encontrados pelo protocolo Multi-Armed Bandit

```
1  package multiSOM;
2 
3  import com.google.gson.Gson;
4  import com.google.gson.GsonBuilder;
5  import ubiSOM.Feature;
6 
7  import java.io.File;
8  import java.io.FileWriter;
9  import java.io.IOException;
10 import java.io.Writer;
11 import java.sql.Timestamp;
12 import java.util.*;
13 import java.util.stream.Collectors;
14 
15 import org.apache.commons.io.FilenameUtils;
16 
17 import javax.swing.filechooser.FileNameExtensionFilter;
18 
19 import static java.util.stream.Collectors.*;
20 
21 public class MultiArmedBanditGeneralCalculations {
22     private List<Feature> gatherFeatures;
23     private List<Feature> resultFeatureList;
24     private List<Feature> baseMabpFile;
```

```
25
26      private List<MultiSOMNoDraw> execsList;
27
28      private String csvFilename;
29      private String member;
30      private int execsRequired;
31
32      public MultiArmedBanditGeneralCalculations(List<MultiSOMNoDraw>
            execsList) {
33          this.execsList = execsList;
34          this.csvFilename = execsList.get(0).getCSVFilename();
35          this.member = execsList.get(0).runningMember;
36          this.execsRequired = execsList.get(0).execsRequired;
37          this.gatherFeatures = new ArrayList<>();
38          for(MultiSOMNoDraw ms : execsList) {
39              List<Feature> aux = new ArrayList<>(ms.listFeatures); //
                    final result list features
40              gatherFeatures.addAll(aux);
41          }
42          this.resultFeatureList = new ArrayList<>();
43      }
44
45      public MultiArmedBanditGeneralCalculations(List<MultiSOMNoDraw>
            execsList,
46                                                 List<Feature> baseFile)
                                                       {
47          this.execsList = execsList;
48          this.csvFilename = execsList.get(0).getCSVFilename();
49          this.execsRequired = execsList.get(0).execsRequired;
50          this.gatherFeatures = new ArrayList<>();
51
52          if(baseFile != null) {
53              this.baseMabpFile = new ArrayList<>(baseFile);
54          }
55
56          if(baseMabpFile != null) {
57              for (MultiSOMNoDraw ms : execsList) {
58                  List<Feature> aux = new ArrayList<>();
59                  for(Feature f : ms.finalAdjustments) {
60                      try {
61                          aux.add(f.clone());
62                      } catch(CloneNotSupportedException e) {
63                          e.getStackTrace();
64                      }
65                  }
66                  gatherFeatures.addAll(aux);
```

```
67              }
68          } else {
69              for (MultiSOMNoDraw ms : execsList) {
70                  List<Feature> aux = new ArrayList<>();
71                  for(Feature f : ms.listFeatures) {
72                      try {
73                          aux.add(f.clone());
74                      } catch(CloneNotSupportedException e) {
75                          e.getStackTrace();
76                      }
77                  }
78                  gatherFeatures.addAll(aux);
79              }
80          }
81
82          if(baseMabpFile != null) {
83              gatherFeatures.addAll(baseMabpFile);
84          }
85
86          this.resultFeatureList = new ArrayList<>();
87      }
88
89      public void calculations() {
90          Map<String, List<Feature>> aux_ = gatherFeatures.stream()
91                  .collect(Collectors.groupingBy(Feature::getNameFeature
                        ));
92
93          double accEstimatedValue = 0.0;
94          int chosenTimesFinal = 0;
95          double finalAverage = 0.0;
96          int sizeOfFeatureList = 0;
97          int rewardFinal = 0;
98
99          for(Map.Entry<String, List<Feature>> entry : aux_.entrySet())
                {
100             String key = entry.getKey();
101             accEstimatedValue = 0; // per key it resets the attributes
102             chosenTimesFinal = 0;
103             rewardFinal = 0;
104             for(Feature f : entry.getValue()) {
105                 chosenTimesFinal += f.getChosen();
106                 rewardFinal += f.getRewardCounter();
107             }
108
109             if(chosenTimesFinal == 0 && rewardFinal == 0) {
110                 finalAverage = 1.D;
```

```
111             } else {
112                 finalAverage = (1.D/chosenTimesFinal) * rewardFinal;
113             }
114 
115             Feature f = new Feature(key, chosenTimesFinal,
                    finalAverage, rewardFinal);
116 
117             resultFeatureList.add(f);
118         }
119 
120         System.out.println("################################");
121         System.out.println("Features selected: ");
122         for(Feature a : resultFeatureList) {
123                 if(a.getEstimatedValue() >= 0.3D && a.getChosen() >=
                        3) {
124                     System.out.println("> " + a.getNameFeature());
125                 }
126         }
127         System.out.println("################################");
128 
129     }
130 
131 
132     public List<Feature> getResultList() {
133         return resultFeatureList;
134     }
135 
136     public void saveResults() throws IOException {
137         Timestamp timestamp = new Timestamp(System.currentTimeMillis()
                );
138 
139         String json = "";
140 
141         String currentDirectory = System.getProperty("user.dir");
142         currentDirectory += "/data/jsonMABP/";
143 
144         File tmp = new File(csvFilename);
145         String auxFilename = tmp.getName();
146         String finalName = FilenameUtils.removeExtension(auxFilename);
147 
148         String pathOutFile = currentDirectory + finalName + ".json";
149 
150 
151         GsonBuilder builder = new GsonBuilder().
                serializeSpecialFloatingPointValues().serializeNulls();
152 
```

```
153         try(Writer writer = new FileWriter(pathOutFile)) {
154             Gson gson = builder.create();
155             gson.toJson(resultFeatureList, writer);
156             json = gson.toJson(resultFeatureList);
157         }
158     }
159 
160     public List<Feature> loadResults() {
161         return null;
162     }
163 }
```

IV

Anexo 4: Testes unitários

Listagem IV.1: Teste unitário do processo de gerar parâmetros de forma aleatória para o parâmetro β e para o parâmetro T do UbiSOM.

```
1  import org.junit.Assert;
2  import org.junit.Test;
3
4  import java.util.ArrayList;
5  import java.util.List;
6  import java.util.Random;
7  import java.util.concurrent.ThreadLocalRandom;
8
9  public class TestGenerateParameters {
10
11     Random random = new Random(123456789); // seed to make the RNG
           deterministic
12
13     public static float B_LOWERBOUND = 0.6f;
14     public static float B_UPPERBOUND = 0.9f;
15
16     public static int T_LOWERBOUND = 1000;
17     public static int T_UPPERBOUND = 2500;
18
19     public List<Float> betas = new ArrayList<>(5);
20
21     @Test
22     public void testGenerateParameters() {
23         for(int i = 0; i < 5; i++) {
24             float beta = randomInRange(B_LOWERBOUND, B_UPPERBOUND);
25             betas.add(beta);
26         }
27
28         int slidingWindowSize =
29                 ThreadLocalRandom.current().nextInt(T_LOWERBOUND,
                       T_UPPERBOUND + 1);
```

```
30
31          for(float f : betas) {
32              Assert.assertTrue(f >= B_LOWERBOUND
33                      && f <= B_UPPERBOUND);
34          }
35
36          Assert.assertTrue(slidingWindowSize >= T_LOWERBOUND
37                  && slidingWindowSize <= T_UPPERBOUND);
38      }
39
40      public float randomInRange(float lowerbound, float upperbound) {
41          float range = upperbound - lowerbound;
42          float scaled = random.nextFloat() * range;
43          float shifted = scaled + lowerbound;
44          return shifted;
45      }
46  }
```

Listagem IV.2: Teste unitário para verificar se a *feature* requistada é efectivamente desligada/ligada.

```
1   import CSV.CsvTable;
2   import multiSOM.BlankPApplet;
3   import multiSOM.MultiSOMNoDraw;
4   import org.junit.Assert;
5   import org.junit.Test;
6   import processing.core.PApplet;
7   import ubiSOM.ModelCoordinator;
8
9   import java.util.concurrent.ExecutorService;
10  import java.util.concurrent.Executors;
11
12  public class TestVerifyFeatureSwitch {
13      @Test
14      public void testVerifyFeatureSwitch() {
15          float etaInitial = 0.1f;
16          float etaFinal = 0.08f;
17          float sigmaInitial = 0.6f;
18          float sigmaFinal = 0.2f;
19
20          float beta = 0.8f;
21          int slidingWindow = 1500;
22
23          String rsltSwitch = "";
24
25          String currentDirectory = System.getProperty("user.dir");
26          currentDirectory += "/exemplos/iris.csv";
```

```
27
28          BlankPApplet a = new BlankPApplet();
29          String[] args = { "multiSOM.BlankPApplet" };
30          ModelCoordinator mc = new ModelCoordinator();
31          MultiSOMNoDraw msNoDraw = new MultiSOMNoDraw(mc, 0,
32                  3000, "localhost", "sequential");
33          mc.setupOneExecNoVisuals(beta, etaInitial, etaFinal,
               sigmaInitial, sigmaFinal, slidingWindow, msNoDraw);
34          PApplet.runSketch(args, a);
35          CsvTable csv = new CsvTable(a, currentDirectory, 0, -1);
36          int[][] dataSet = csv.getData();
37          msNoDraw.getModel().setFeatures(dataSet[0].length);
38          rsltSwitch = msNoDraw.getModel().temporarilyExcludeFeature(0);
39          Assert.assertTrue(rsltSwitch.contains("Removed Feature: 0"));
40      }
41  }
```

Listagem IV.3: Teste unitário de verificação à leitura de um *dataset* CSV.

```
1   import CSV.CsvTable;
2   import multiSOM.BlankPApplet;
3   import org.junit.Assert;
4   import org.junit.Test;
5   import processing.core.PApplet;
6   import ubiSOM.Feature;
7
8   import java.util.ArrayList;
9   import java.util.List;
10  import java.util.logging.Level;
11  import java.util.logging.Logger;
12
13  public class TestCSVParsing {
14      public static int NO_OF_LINES_CSV_IRIS = 151;
15      public static int NO_OF_COLUMNS_CSV_IRIS = 4;
16
17      @Test
18      public void testLengthFeatures() {
19          List<Feature> listFeatures = new ArrayList<Feature>();
20
21          String currentDirectory = System.getProperty("user.dir");
22          currentDirectory += "/exemplos/iris.csv";
23
24          BlankPApplet a = new BlankPApplet();
25          String[] args = { "multiSOM.BlankPApplet" };
26          PApplet.runSketch(args, a);
27          CsvTable csv = new CsvTable(a, currentDirectory, 0, -1);
28          int[][] dataSet = csv.getData();
```

```
29
30          Assert.assertEquals(dataSet.length, NO_OF_LINES_CSV_IRIS);
31          Assert.assertEquals(dataSet[0].length, NO_OF_COLUMNS_CSV_IRIS)
                ;
32      }
33  }
```

Listagem IV.4: Teste unitário para verificar se as instâncias requistadas são criadas de forma coerente.

```
1   import multiSOM.MultiSOMNoDraw;
2   import org.junit.Assert;
3   import org.junit.Test;
4   import ubiSOM.ModelCoordinator;
5
6   import java.io.IOException;
7   import java.io.PipedInputStream;
8   import java.io.PipedOutputStream;
9   import java.io.PrintStream;
10  import java.rmi.ServerError;
11  import java.util.ArrayList;
12  import java.util.List;
13  import java.util.Scanner;
14
15  public class TestInstantiateModels {
16
17      @Test
18      public void testInstantionModels() throws IOException {
19          ModelCoordinator mc = new ModelCoordinator();
20
21          // define parameters
22          float etaInitial = 0.1f;
23          float etaFinal = 0.08f;
24          float sigmaInitial = 0.6f;
25          float sigmaFinal = 0.2f;
26
27          float beta = 0.8f;
28          int slidingWindow = 1500;
29
30          int mockWantedExecs = 5; // Valor ficticio para no de treinos
                requisitados
31
32          LoggedPrintStream lpsErr = LoggedPrintStream.create(System.err
                );
33          System.setErr(lpsErr);
34
35          for(int i = 0; i < mockWantedExecs; i++) {
```

```
36              MultiSOMNoDraw msNoDraw = new MultiSOMNoDraw(mc, i,
37                      1000, "localhost", "sequential");
38              mc.setupOneExecNoVisuals(beta, etaInitial, etaFinal,
                    sigmaInitial, sigmaFinal, slidingWindow, msNoDraw); //
                    Instanciar um treino
39              slidingWindow++;
40          }
41
42          System.err.flush();
43          System.setErr(lpsErr.underlying);
44
45          String tmp = lpsErr.buf.toString();
46
47          int lastIndex = 0;
48          int count = 0;
49          String findStr = "UbiSOM instanciation";
50          while(lastIndex != -1) {
51              lastIndex = tmp.indexOf(findStr, lastIndex);
52
53              if(lastIndex != -1) {
54                  count++;
55                  lastIndex += findStr.length();
56              }
57          }
58
59          Assert.assertEquals(count, mockWantedIterations); // Verificar
                se o numero de treinos equivale ao numero de treinos
               requisitados
60      }
61  }
```

Listagem IV.5: Classe auxiliar para ler os *outputs* criados por uma aplicação Java a partir do terminal.

```
1  import java.io.FilterOutputStream;
2  import java.io.IOException;
3  import java.io.OutputStream;
4  import java.io.PrintStream;
5  import java.lang.reflect.Field;
6
7  public class LoggedPrintStream extends PrintStream {
8
9      final StringBuilder buf;
10     final PrintStream underlying;
11
12     LoggedPrintStream(StringBuilder sb, OutputStream os, PrintStream
           ul) {
```

```
13          super(os);
14          this.buf = sb;
15          this.underlying = ul;
16      }
17
18      public static LoggedPrintStream create(PrintStream toLog) {
19          try {
20              final StringBuilder sb = new StringBuilder();
21              Field f = FilterOutputStream.class.getDeclaredField("out")
                    ;
22              f.setAccessible(true);
23              OutputStream psout = (OutputStream) f.get(toLog);
24              return new LoggedPrintStream(sb, new FilterOutputStream(
                    psout) {
25                  public void write(int b) throws IOException {
26                      super.write(b);
27                      sb.append((char) b);
28                  }
29              }, toLog);
30          } catch (NoSuchFieldException shouldNotHappen) {
31          } catch (IllegalArgumentException shouldNotHappen) {
32          } catch (IllegalAccessException shouldNotHappen) {
33          }
34          return null;
35      }
36  }
```

Listagem IV.6: Teste aos valores retornados por cada um dos métodos utilizados para gerar uma iteração sequencial ou uma iteração aleatória.

```
1   import org.junit.Assert;
2   import org.junit.Test;
3
4   import java.util.Collections;
5   import java.util.List;
6   import java.util.stream.Collectors;
7   import java.util.stream.IntStream;
8
9   import static org.junit.Assert.assertFalse;
10  import static org.junit.Assert.assertTrue;
11
12  public class TestIteratorRandomSequential {
13      public int iteratorSequential = -1;
14      public int iteratorRandom = -1;
15      public int artificialDatasetLength = 10000; // size of mock
            dataset size
16      public int iterator = 0;
```

```
17
18      @Test
19      public void testIteratorRandomSequential() {
20          List<Integer> iteratorSequential = generateIntegerList(
                artificialDatasetLength);
21          List<Integer> iteratorRandom = generateIntegerList(
                artificialDatasetLength);
22
23          // random case
24          Collections.shuffle(iteratorRandom);
25          int valueRand = nextRandom();
26          valueRand = iteratorRandom.get(valueRand);
27
28          // sequential case
29          int valueSeq = nextSequential();
30
31          Assert.assertEquals(valueSeq, 0);
32          Assert.assertNotEquals(valueRand, 0);
33      }
34
35      public List<Integer> generateIntegerList(int
            artificialDatasetLength) {
36          List<Integer> iterations = IntStream.rangeClosed(0,
                artificialDatasetLength - 1)
37                  .boxed().collect(Collectors.toList());
38
39          return iterations;
40      }
41
42      public int nextSequential() {
43          iterator = (++iteratorSequential) % (artificialDatasetLength -
                1);
44          return iterator;
45      }
46
47      public int nextRandom() {
48          iterator = (++iteratorRandom) % (artificialDatasetLength - 1);
49          return iterator;
50      }
51  }
```

V

Anexo 5: *Logs* resultantes dos treinos

Este anexo contém um exemplo relativo a um exemplo de um *log* que é criado aquando a execução de um treino do modelo UbiSOM na aplicação MultiSOM.

listings xcolor

Neste anexo estão presentes exemplos relativos aos resultados produzidos pelo sistema aquando da execução de um treino e da execução do mecanismo de *feature switching*.

Listagem V.1: Exemplo de um resultado relativo a um treino do algoritmo UbiSOM.

```
1  Execution Unit: localhost
2  Dataset: artificial/iris_10features.csv
3  ############### Model @ [0.1, 0.08, 0.6, 0.2, 0.7, 2000] Error Report
       ###############
4  Step: 10
5  Topological Error: 100.0
6  Quantization Error: 2.137865
7  ###############
       ###############
8
9  ############### Model @ [0.1, 0.08, 0.6, 0.2, 0.7, 2000] Error Report
       ###############
10 Step: 20
11 Topological Error: 100.0
12 Quantization Error: 2.9349756
13 ###############
       ###############
14
15 ############### Model @ [0.1, 0.08, 0.6, 0.2, 0.7, 2000] Error Report
       ###############
16 Step: 30
17 Topological Error: 99.933334
18 Quantization Error: 3.283025
19 ###############
       ###############
20
```

```
21 ############### Model @ [0.1, 0.08, 0.6, 0.2, 0.7, 2000] Error Report
       ###############
22 Step: 40
23 Topological Error: 99.6
24 Quantization Error: 3.5265884
25 ###############
       ###############
26
27 ############### Model @ [0.1, 0.08, 0.6, 0.2, 0.7, 2000] Error Report
       ###############
28 Step: 50
29 Topological Error: 99.53333
30 Quantization Error: 3.7942696
31 ###############
       ###############
32
33 ############### Model @ [0.1, 0.08, 0.6, 0.2, 0.7, 2000] Error Report
       ###############
34 Step: 60
35 Topological Error: 99.2
36 Quantization Error: 4.2174125
37 ###############
       ###############
38
39 ############### Model @ [0.1, 0.08, 0.6, 0.2, 0.7, 2000] Error Report
       ###############
40 Step: 70
41 Topological Error: 99.0
42 Quantization Error: 4.500982
43 ###############
       ###############
44
45 (...)
46
47 Time elapsed in training @ [0.1, 0.08, 0.6, 0.2, 0.7, 2000]: 13764 ms.
```

Listagem V.2: Ficheiro JSON resultante de um treino com o mecanismo de *feature switching* activo. O treino foi feito com o *dataset* Iris.

```
1 [
2    {
3       "actionChosenTotal":9,
4       "estimatedValue":0.1111111111111111,
5       "currBinaryIndicator":0,
6       "nameFeature":"petalLength",
7       "rewardCounter":1,
```

```
        "lastEstimatedValue":0.0
    },
    {
        "actionChosenTotal":9,
        "estimatedValue":0.1111111111111111,
        "currBinaryIndicator":0,
        "nameFeature":"petalWidth",
        "rewardCounter":1,
        "lastEstimatedValue":0.0
    },
    {
        "actionChosenTotal":5,
        "estimatedValue":0.4,
        "currBinaryIndicator":0,
        "nameFeature":"duplicate4",
        "rewardCounter":2,
        "lastEstimatedValue":0.0
    },
    {
        "actionChosenTotal":9,
        "estimatedValue":0.5555555555555556,
        "currBinaryIndicator":0,
        "nameFeature":"artificial1",
        "rewardCounter":5,
        "lastEstimatedValue":0.0
    },
    {
        "actionChosenTotal":1,
        "estimatedValue":0.0,
        "currBinaryIndicator":0,
        "nameFeature":"duplicate3",
        "rewardCounter":0,
        "lastEstimatedValue":0.0
    },
    {
        "actionChosenTotal":8,
        "estimatedValue":0.125,
        "currBinaryIndicator":0,
        "nameFeature":"sepalLength",
        "rewardCounter":1,
```

```
        "lastEstimatedValue":0.0
    },
    {
        "actionChosenTotal":147,
        "estimatedValue":0.6258503401360545,
        "currBinaryIndicator":0,
        "nameFeature":"sepalWidth",
        "rewardCounter":92,
        "lastEstimatedValue":0.0
    },
    {
        "actionChosenTotal":20,
        "estimatedValue":0.55,
        "currBinaryIndicator":0,
        "nameFeature":"artificial2",
        "rewardCounter":11,
        "lastEstimatedValue":0.0
    },
    {
        "actionChosenTotal":37,
        "estimatedValue":0.5675675675675675,
        "currBinaryIndicator":0,
        "nameFeature":"duplicate2",
        "rewardCounter":21,
        "lastEstimatedValue":0.0
    },
    {
        "actionChosenTotal":12,
        "estimatedValue":0.08333333333333333,
        "currBinaryIndicator":0,
        "nameFeature":"duplicate1",
        "rewardCounter":1,
        "lastEstimatedValue":0.0
    }
]
```

VI

Anexo 6: Treino com algoritmo UbiSOM

Neste anexo contém a implementação referente aos métodos que compõem o processo de treino do modelo UbiSOM, implementado na aplicação MultiSOM, de forma a que seja permitido múltiplos treinos em paralelo com múltiplas parametrizações.

Listagem VI.1: Métodos para leitura e *parsing* de um conjunto de dados.

```
1       /**
2       * Import and parse the dataset CSV file.
3       * @param filename - dataset CSV file location.
4       * @param a - instance of PApplet to be passed as argument to
            CsvTable class.
5       */
6      public void createCSV(String filename, BlankPApplet a) {
7          currentDirectory = System.getProperty("user.dir");
8          currentDirectory += "/exemplos/" + filename;
9
10         csv = new CsvTable(a, currentDirectory, 0, -1);
11         csvFilename = currentDirectory;
12         dataSet = csv.getData();
13         this.multiSOM = a;
14
15         errorReport.add("Dataset: " + csvFilename);
16
17         this.iterationThreshold = getWantedIterations() -
               THRESHOLD_FACTOR;
18
19         if(iteratorMode.equals(ITERATOR_MODE_RANDOM)) {
20             this.iterationNumHist = IntStream.rangeClosed(0, dataSet.
                   length - 1).boxed().collect(Collectors.toList());
21             Collections.shuffle(iterationNumHist);
22         }
23     }
```

```
24
25      (...)
26
27      public void startCsvTable(String fn, int firstCol, int size) {
28      try {
29        this.firstCol=firstCol; this.totCol=size;
30        Table table = parent.loadTable(fn, "header");
31        System.out.println("Fn:"+fn);
32
33        nFeatures = table.getColumnCount()-firstCol;
34        titles = table.getColumnTitles();
35        nLines = table.getRowCount();
36
37        if( totCol == -1) {
38          totCol=nFeatures;
39        } if(nFeatures > totCol ) {
40          System.out.println("Ignoring last " + (nFeatures-totCol) + "
              cols.");
41          nFeatures=totCol;
42        }
43
44        System.out.println(firstCol+"+"+nLines + " total rows in table")
            ;
45        System.out.println("Features: " + titles.toString() + " # " +
            nFeatures+" "+firstCol+" of "+totCol);
46
47        data = new int [nLines+1][nFeatures];
48        keys = new String[nLines+1];
49
50        if(!lockMinMax || min==null||max==null) {
51          min = new float[nFeatures];
52          max = new float[nFeatures];
53
54          calcMinMax(table);
55
56          System.out.print("min/max:");
57          for(int i=0; i<nFeatures; i++)
58            System.out.print("["+i+"]:"+min[i]+"/"+max[i]);
59          System.out.println("");
60
61        }
62        loadNorm(table);
63      } catch(Exception e) {
64        System.err.println("Exception in CsvTable.startCsvTable:" + e.
            getMessage());
65        System.err.println("DataStream.CSV_DATA_FILE(?)/fn="+ fn);
```

```
66        if(parent != null)
67          parent.exit();
68        //throw(e);
69      }
70    }
```

Listagem VI.2: Método que instancia o processo de aprendizagem do algoritmo UbiSOM.

```
1     public void startTrainingWithCSVBase() {
2         long start = System.currentTimeMillis();
3         csvHeaderBase = csv.getTitles();
4
5         System.out.println("[INFO] Model without visuals created @ " +
              Arrays.toString(currModel.getParameters()));
6
7         int numberFeatures = csv.getnFeatures();
8         currModel.setFeatures(numberFeatures);
9         this.numberFeatures = csv.getnFeatures();
10        startFeaturesScheduler();
11        while((iterationModel < wantedIterations)) {
12            int[] point;
13            point = next();
14
15            MultiPoint p = patternPointsLite.addMPoint(point);
16            currModel.SOMLearn(p);
17
18            if(iterationModel % ITERATIONS_INTERVAL_PRINT_ERROR == 0)
19                printErrorReport(); // Returns quantization and
                      topology error in a 10 iteration interval
20
21            if(iterationModel == iterationThreshold)
22                time.cancel(); // this condition is to stop the thread
                      from choking one last message
23        }
24
25    (... feature switching part ...)
26
27        System.out.println("\n\n\n[MODEL " + Arrays.toString(currModel
              .getParameters()) + "]"
28                + " Training over.\n\n\n");
29
30        long end = System.currentTimeMillis();
31        long elapsedTime = end - start;
32        System.out.println("Time elapsed in training @ " + Arrays.
              toString(currModel.getParameters())
33                + ": " + elapsedTime + " ms.");
```

```
34          errorReport.add("Time elapsed in training @ " + Arrays.
                toString(currModel.getParameters())
35                  + ": " + elapsedTime + " ms. \n");
36
37          saveMetricsLogToFile();
38          saveMultiArmedBanditLogToFile();
39          currModel.writeModelCSV(this);
40          saveModel();
41      }
```

Listagem VI.3: Método que normaliza uma observação

```
1   private double[] normalize(double[] v_in) {
2           double [] v = parseDataSet(v_in);
3
4           for (int i = 0; i < MAX_NUMBER_FEATURES; i++) {
5               v[i] = (v[i] - minValues[i]) / (maxValues[i] - minValues[i
                    ]);
6           }
7
8           return v;
9       }
10
11      private double[] parseDataSet(double v[]) {
12          for(int i=0; i< nFeatures ; i++)
13              if(availableFeatures[i]) {
14                  v[i] = avgFeature[i];
15              }
16          return v;
17      }
```

Listagem VI.4: Método que pesquisa a BMU de um certo protótipo.

```
1       private void find_bmu(int idx, double[] p) {
2           double d;
3           getDist()[idx] = Double.MAX_VALUE;
4           double second_best = getDist()[idx];
5           int bmu_x2 = -2, bmu_y2 = -2;
6
7           for (int k = 0; k < ubiSomExec.getHeight(); k++) {
8               for (int i = 0; i < ubiSomExec.getWidth(); i++) {
9                   d = distanceSOM(p, ubiSomExec.get(i, k).
                        getDoubleVector());
10                  if(multiSOM != null) {
11                      if (multiSOM.dataLabels.distLabel > 0) {
12                          multiSOM.dataLabels.markUnits(i, k, d);
13                      }
```

```
14                  }
15                  if (d < getDist()[idx]) {
16                      if (second_best > getDist()[idx]) {
17                          bmu_x2 = getBMU_X()[idx];
18                          bmu_y2 = getBMU_Y()[idx];
19                          second_best = getDist()[idx];
20                      }
21                      getDist()[idx] = d;
22                      getBMU_X()[idx] = i;
23                      getBMU_Y()[idx] = k;
24                  } else {
25                      if (second_best > d) {
26                          second_best = d;
27                          bmu_x2 = i;
28                          bmu_y2 = k;
29                      }
30                  }
31              }
32          }
33
34          if ((bmu_x2 == getBMU_X()[idx] && PApplet.abs(bmu_y2 -
                getBMU_Y()[idx]) == 1)
35                  || (bmu_y2 == getBMU_Y()[idx] && PApplet.abs(bmu_x2 -
                        getBMU_X()[idx]) == 1))
36              top_err[idx] = 0;
37          else
38              top_err[idx] = 1;
39
40          if(multiSOM != null) {
41              if (multiSOM.dataLabels.distLabel == 0) {
42                  multiSOM.dataLabels.markUnits(getBMU_X()[idx],
                        getBMU_Y()[idx], getDist()[idx]);
43              }
44          }
45      }
```

Listagem VI.5: Métodos relacionados com a instanciação dos dois tipos de iteradores do *dataset* inicial: sequencial e aleatório.

```
1       public void createCSV( ) {
2         // (... codigo relacionado com o parsing do CSV inicial ...)
3         if(iteratorMode.equals(ITERATOR_MODE_RANDOM)) {
4               this.iterationNumHist = IntStream.rangeClosed(0, dataSet.
                    length - 1).boxed().collect(Collectors.toList());
5               Collections.shuffle(iterationNumHist);
6           }
7           // (... codigo relacionado com o parsing do CSV inicial ...)
```

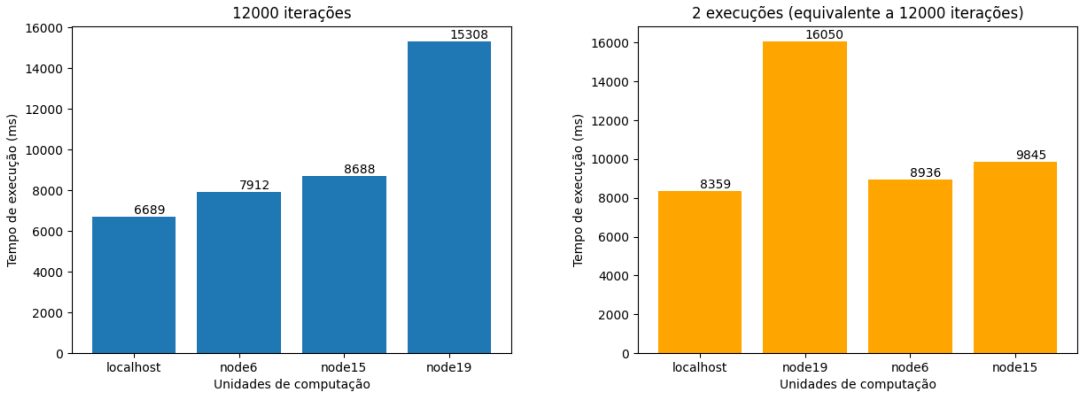
```
8       }
9
10
11      public int[] next() {
12          if(iteratorMode.equals("random")) {
13              sampleIdx = iterationNumHist.get(iterator);
14              iterator = (++iterator) % (dataSet.length - 1);
15          } else if(iteratorMode.equals("sequential"))
16              sampleIdx = (iterator++) % (dataSet.length - 1);
17
18          return dataSet[sampleIdx];
19      }
```

VII

Anexo 7: Resultados da secção 5.3.1

Neste anexo estão presentes os resultados relativamente à experiência que compara o cenário de treinar o modelo UbiSOM com o máximo de número de iterações possível ao cenário de vários mapas a treinar separadamente (numa única unidade de computação).

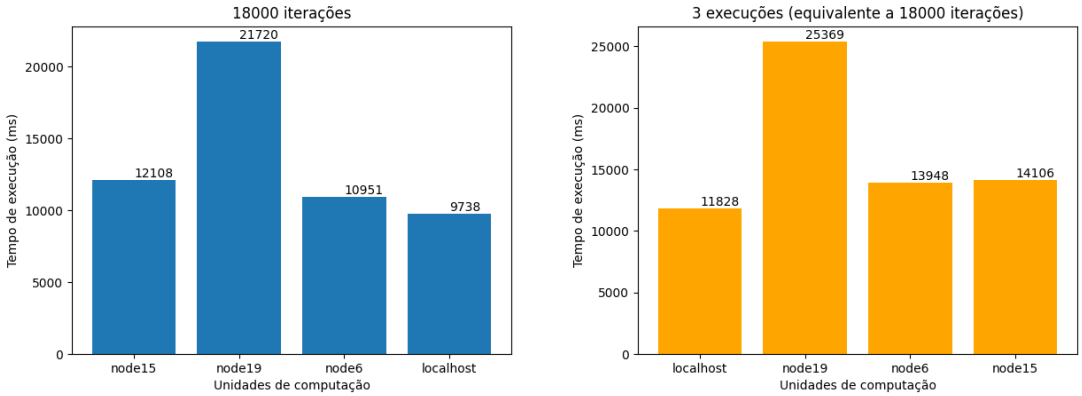
VII.1 *Dataset* Iris



(a) Resultados relativos ao tempo de execução do treino do algoritmo UbiSOM com 12000 iterações, usando uma unidade de computação, para o *dataset* Iris.

(b) Resultados relativos ao tempo de execução de dois treinos de 6000 iterações cada, utilizando várias unidades de computação, para o *dataset* Iris.

Figura VII.1: Cenário de treino com 12000 iterações *vs.* dois treinos em cada unidade de computação cada um com 6000 iterações.

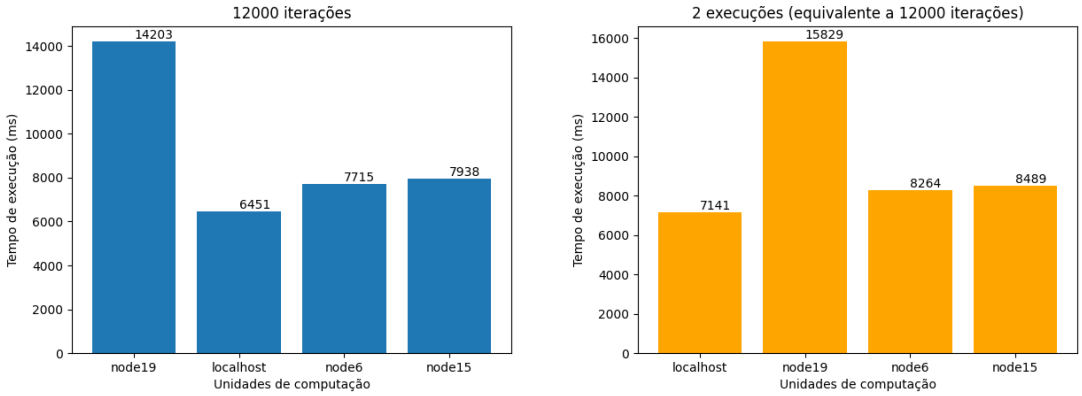


(a) Resultados relativos ao tempo de execução do treino do algoritmo UbiSOM com 18000 iterações, usando uma unidade de computação, para o *dataset* Iris.

(b) Resultados relativos ao tempo de execução de três treinos de 6000 iterações cada, utilizando várias unidades de computação, para o *dataset* Iris.

Figura VII.2: Cenário de treino com 18000 iterações *vs.* três treinos em cada unidade de computação cada um com 6000 iterações.

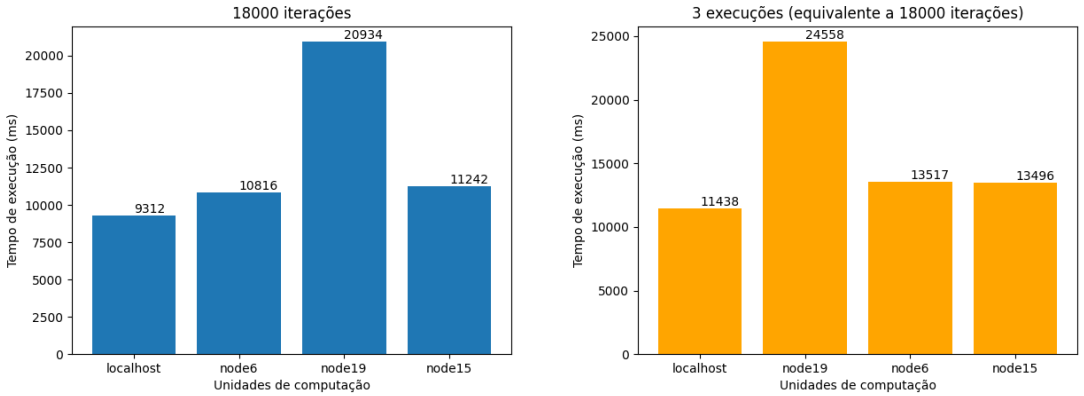
VII.2 *Dataset* Chain



(a) Resultados relativos ao tempo de execução do treino do algoritmo UbiSOM com 12000 iterações, usando uma unidade de computação, para o *dataset* Chain.

(b) Resultados relativos ao tempo de execução de dois treinos de 6000 iterações cada, utilizando várias unidades de computação, para o *dataset* Chain.

Figura VII.3: Cenário de treino com 12000 iterações *vs.* dois treinos em cada unidade de computação cada um com 6000 iterações.

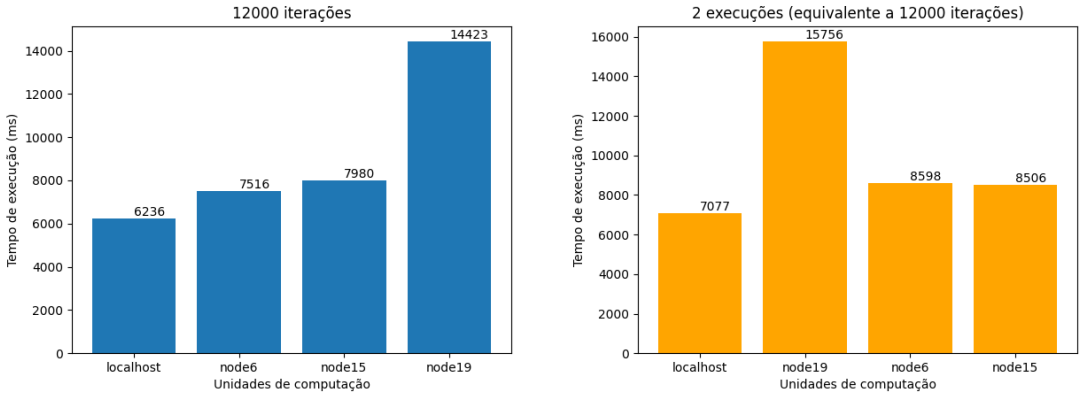


(a) Resultados relativos ao tempo de execução do treino do algoritmo UbiSOM com 18000 iterações, usando uma unidade de computação, para o *dataset* Chain.

(b) Resultados relativos ao tempo de execução de três treinos de 6000 iterações cada, utilizando várias unidades de computação, para o *dataset* Chain.

Figura VII.4: Cenário de treino com 18000 iterações *vs.* três treinos em cada unidade de computação cada um com 6000 iterações.

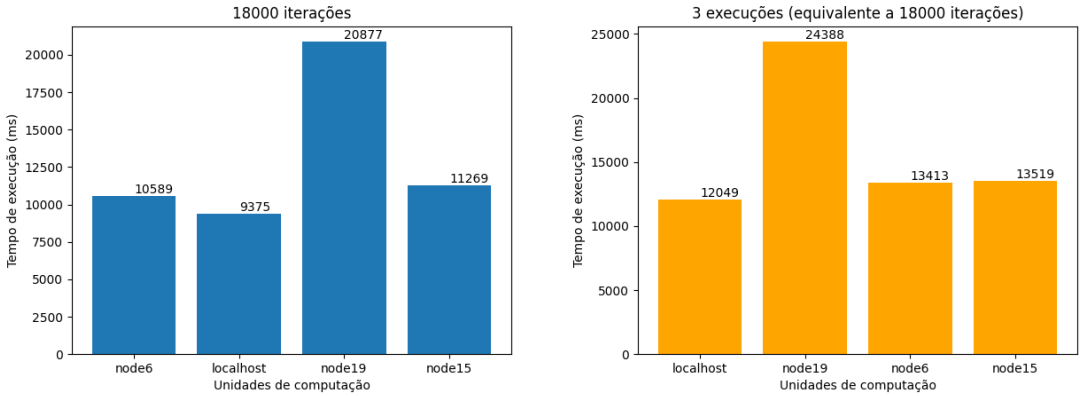
VII.3 *Dataset* Hepta



(a) Resultados relativos ao tempo de execução do treino do algoritmo UbiSOM com 12000 iterações, usando uma unidade de computação, para o *dataset* Hepta.

(b) Resultados relativos ao tempo de execução de dois treinos de 6000 iterações cada, utilizando várias unidades de computação, para o *dataset* Hepta.

Figura VII.5: Cenário de treino com 12000 iterações *vs.* dois treinos em cada unidade de computação cada um com 6000 iterações.

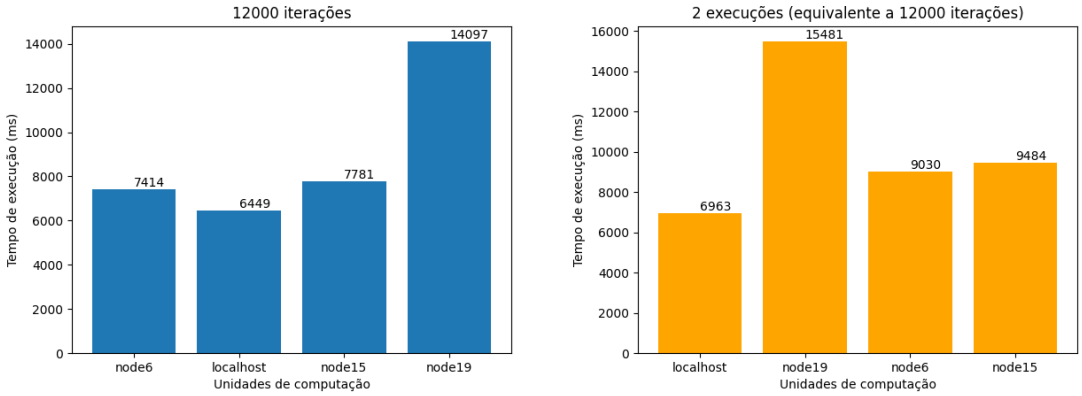


(a) Resultados relativos ao tempo de execução do treino do algoritmo UbiSOM com 18000 iterações, usando uma unidade de computação, para o *dataset* Hepta.

(b) Resultados relativos ao tempo de execução de três treinos de 6000 iterações cada, utilizando várias unidades de computação, para o *dataset* Hepta.

Figura VII.6: Cenário de treino com 18000 iterações *vs.* três treinos em cada unidade de computação cada um com 6000 iterações.

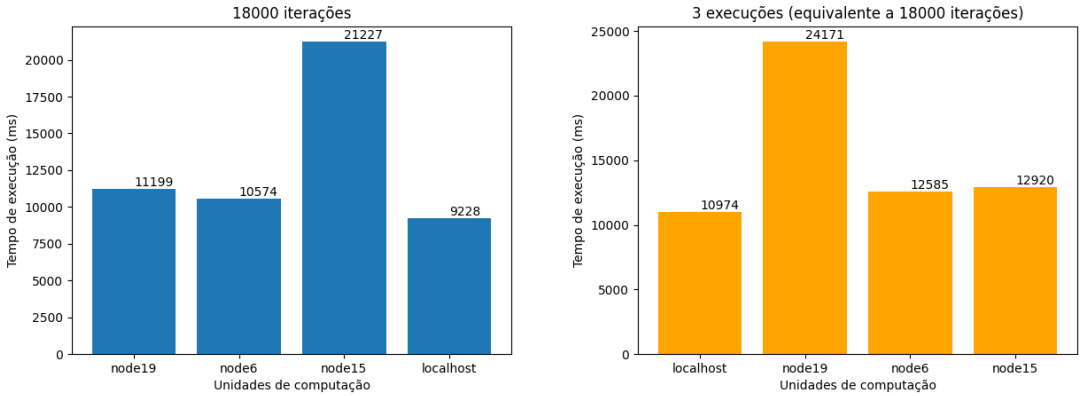
VII.4 *Dataset* Complex



(a) Resultados relativos ao tempo de execução do treino do algoritmo UbiSOM com 12000 iterações, usando uma unidade de computação, para o *dataset* Complex.

(b) Resultados relativos ao tempo de execução de dois treinos de 6000 iterações cada, utilizando várias unidades de computação, para o *dataset* Complex.

Figura VII.7: Cenário de treino com 12000 iterações *vs.* dois treinos em cada unidade de computação cada um com 6000 iterações.



(a) Resultados relativos ao tempo de execução do treino do algoritmo UbiSOM com 18000 iterações, usando uma unidade de computação, para o *dataset* Complex.

(b) Resultados relativos ao tempo de execução de três treinos de 6000 iterações cada, utilizando várias unidades de computação, para o *dataset* Complex.

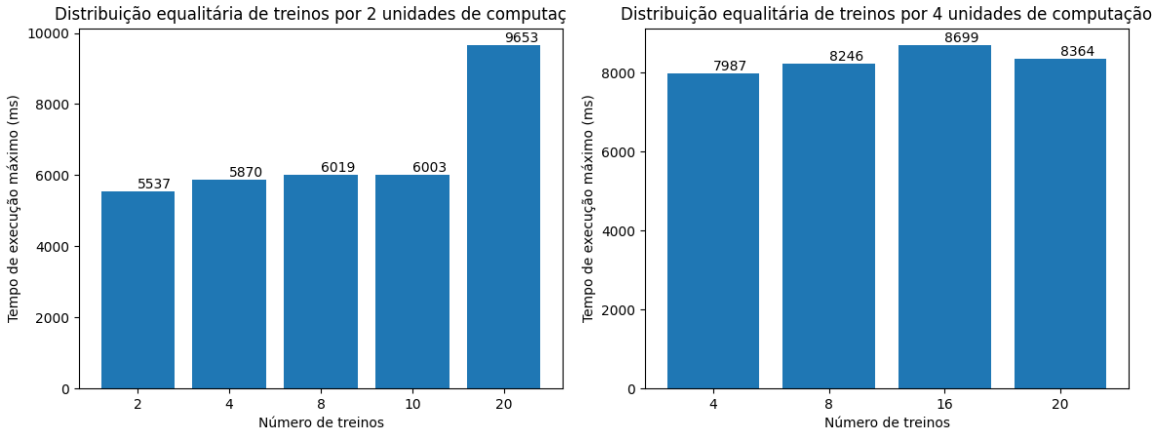
Figura VII.8: Cenário de treino com 18000 iterações *vs.* três treinos em cada unidade de computação cada um com 6000 iterações.

VIII

Anexo 8: Resultados da secção 5.3.2

Neste anexo estão presentes os resultados relativamente à experiência que compara o cenário de executar treinos do modelo UbiSOM numa única unidade de computação ao cenário de utilizar várias unidades de computação em uníssono para executar vários treinos do modelo UbiSOM.

VIII.1 *Dataset* Clouds



(a) Tempo de execução resultante de treinos distribuídos de forma equalitária por duas unidades de computação.

(b) Tempo de execução resultante de treinos distribuídos de forma equalitária por quatro unidades de computação.

Figura VIII.1: Resultados referentes às experiências com distribuição equalitária de treinos por várias unidades de computação, utilizando o *dataset* Clouds.

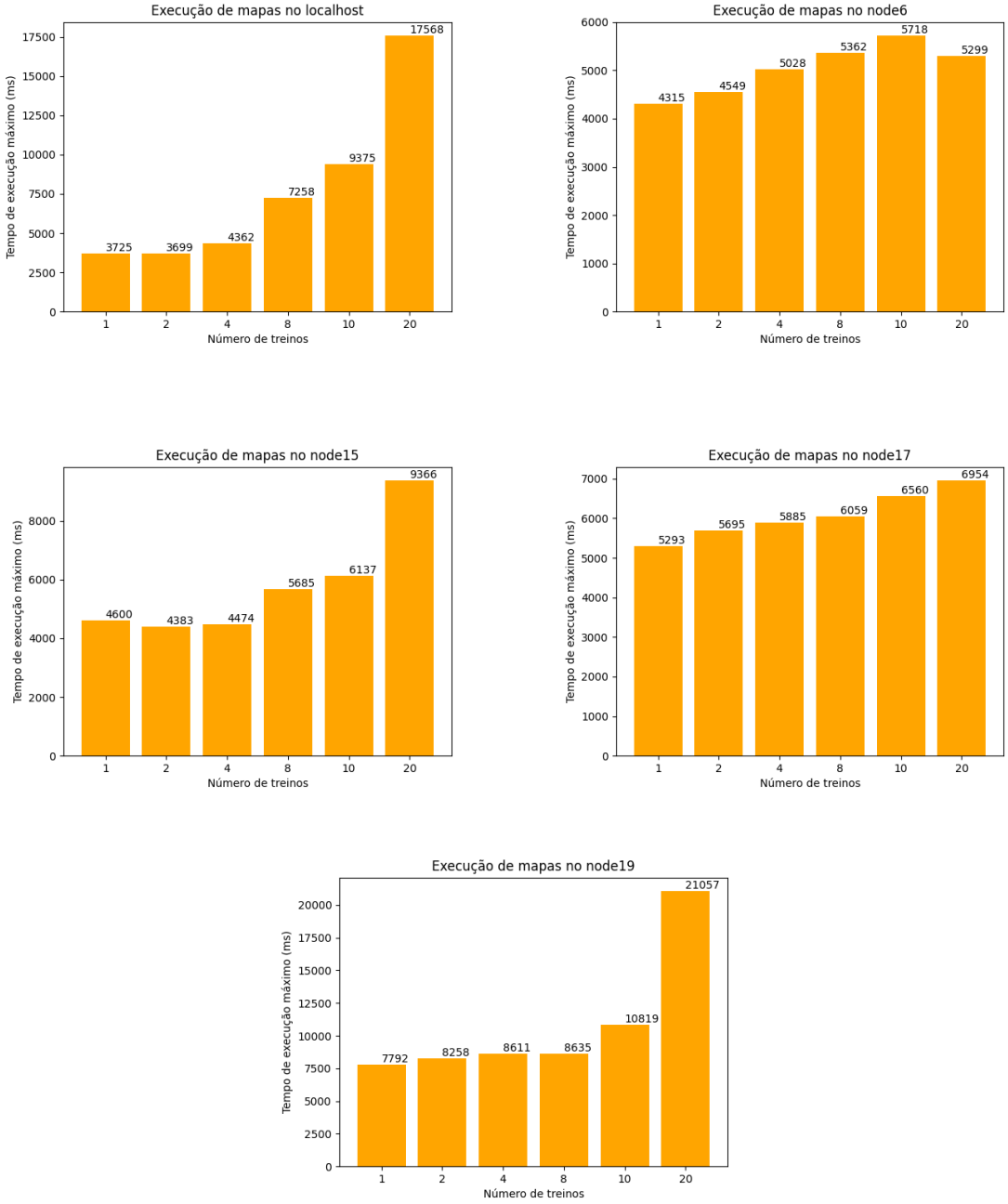
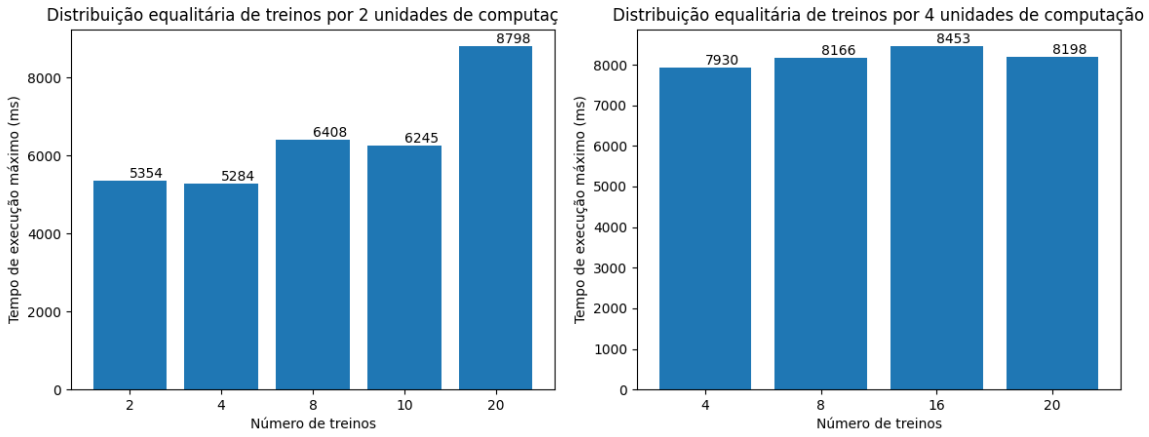


Figura VIII.2: Tempos de execução resultantes de um número de treinos variáveis numa unidade de computação única utilizando o *dataset* Clouds.

VIII.2 *Dataset* Complex



(a) Tempo de execução resultante de treinos distribuídos de forma equalitária por duas unidades de computação.

(b) Tempo de execução resultante de treinos distribuídos de forma equalitária por quatro unidades de computação.

Figura VIII.3: Resultados referentes às experiências com distribuição equalitária de treinos por várias unidades de computação, utilizando o *dataset* Complex.

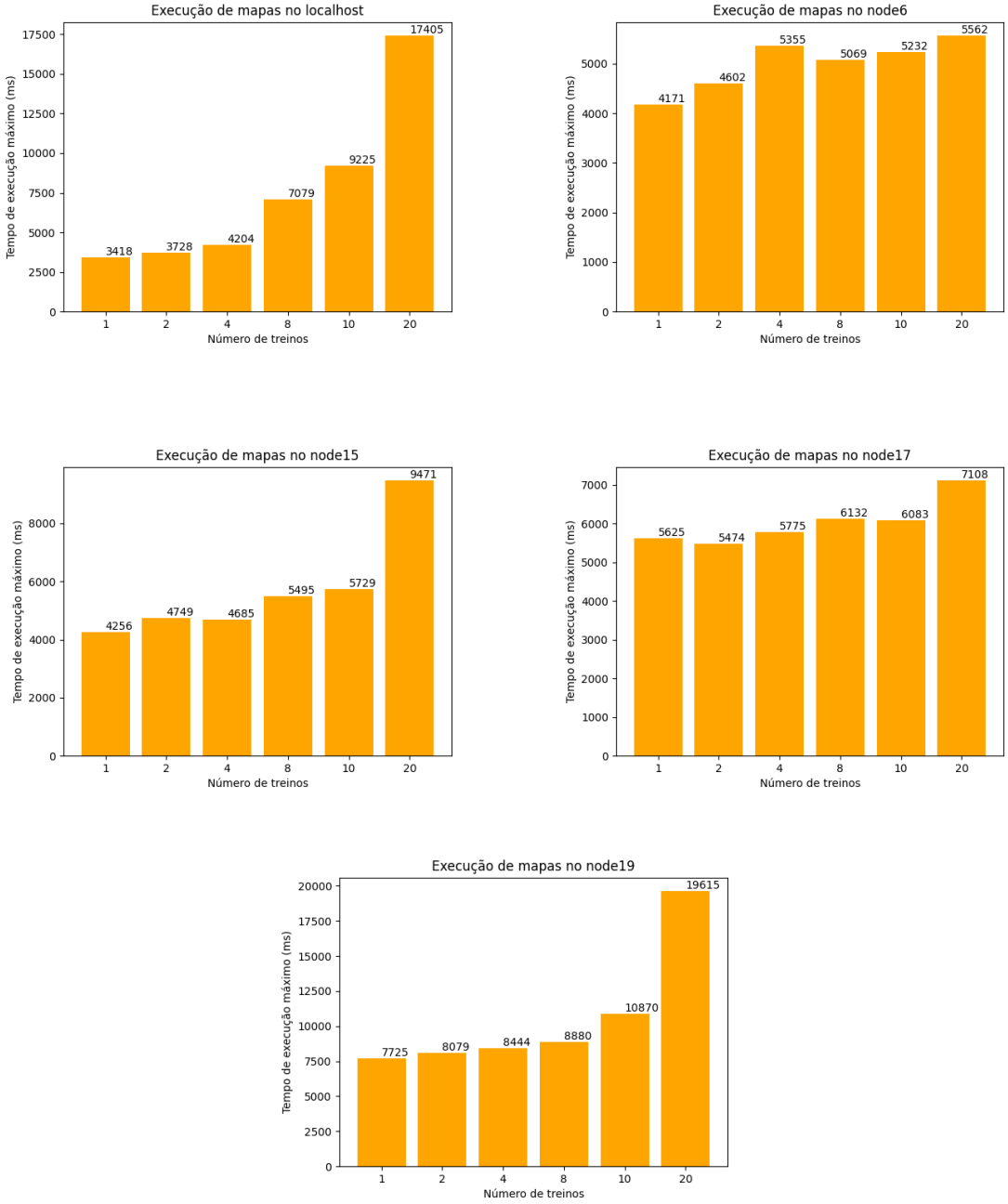
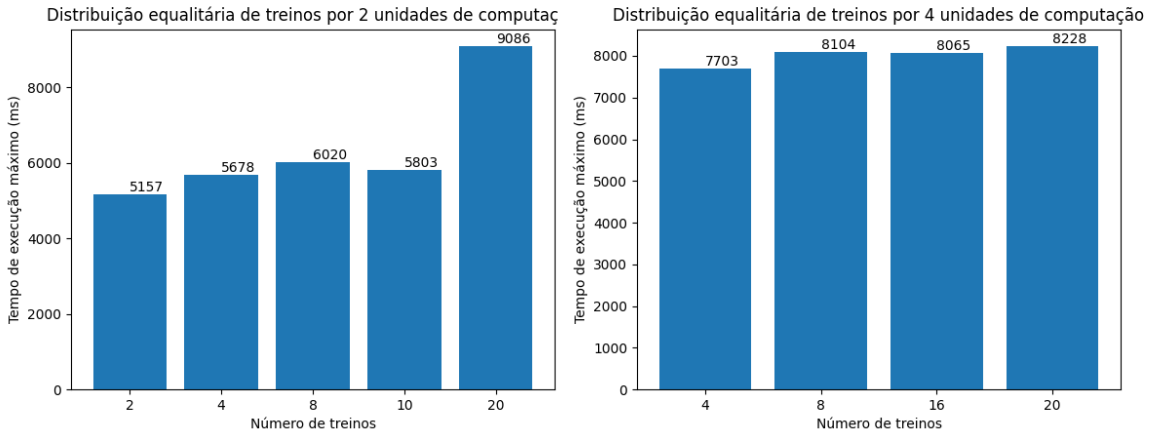


Figura VIII.4: Tempos de execução resultantes de um número de treinos variáveis numa unidade de computação única utilizando o *dataset* Complex.

VIII.3 *Dataset* Hepta



(a) Tempo de execução resultante de treinos distribuídos de forma equalitária por duas unidades de computação.

(b) Tempo de execução resultante de treinos distribuídos de forma equalitária por quatro unidades de computação.

Figura VIII.5: Resultados referentes às experiências com distribuição equalitária de treinos por várias unidades de computação, utilizando o *dataset* Hepta.

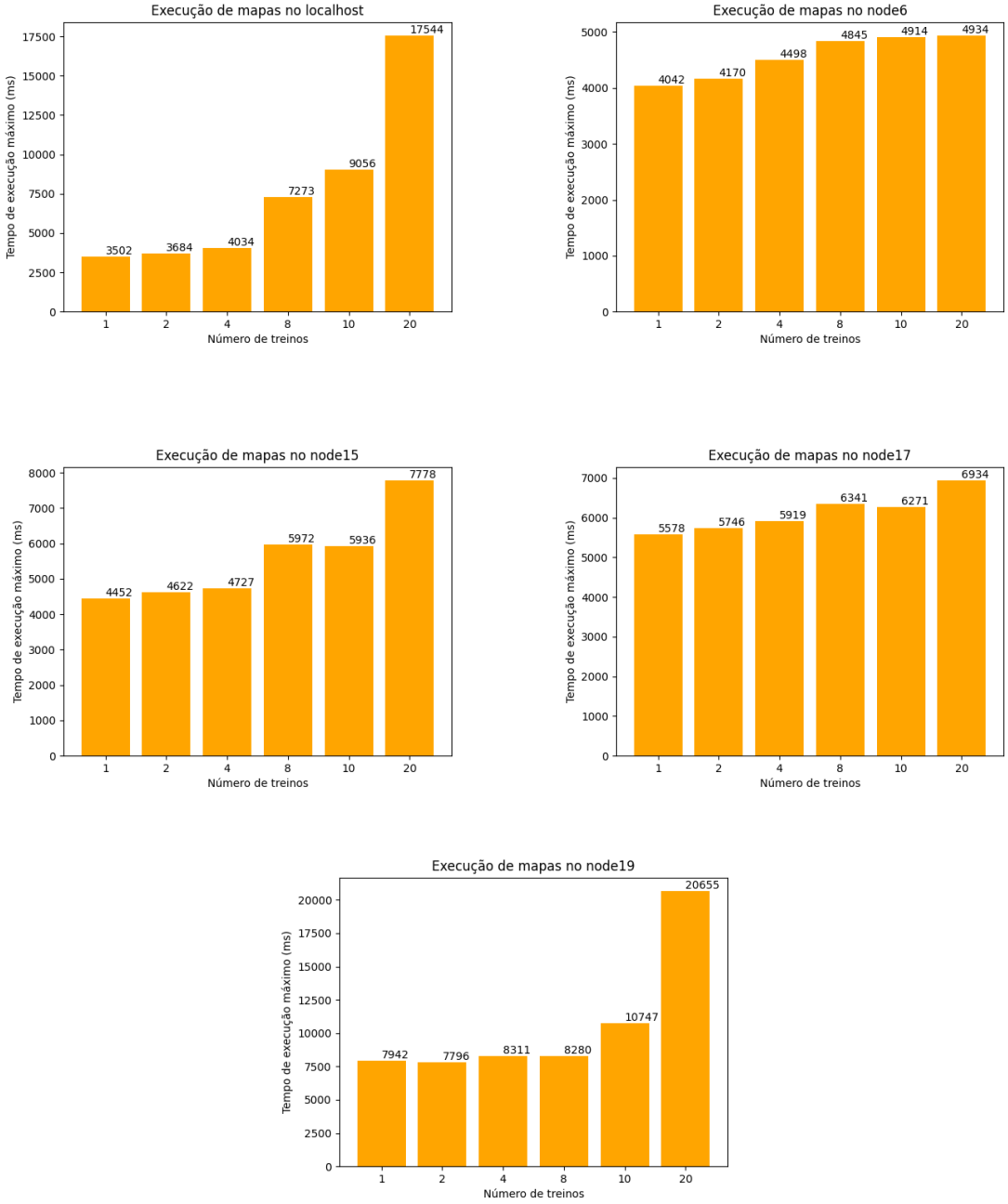
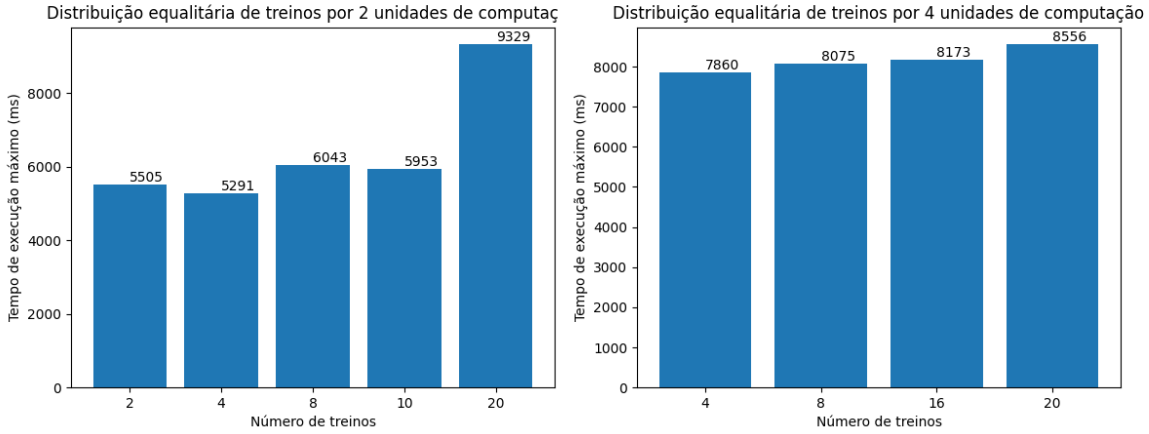


Figura VIII.6: Tempos de execução resultantes de um número de treinos variáveis numa unidade de computação única utilizando o *dataset* Hepta.

VIII.4 *Dataset* Chain



(a) Tempo de execução resultante de treinos distribuídos de forma equalitária por duas unidades de computação.

(b) Tempo de execução resultante de treinos distribuídos de forma equalitária por quatro unidades de computação.

Figura VIII.7: Resultados referentes às experiências com distribuição equalitária de treinos por várias unidades de computação, utilizando o *dataset* Chain.

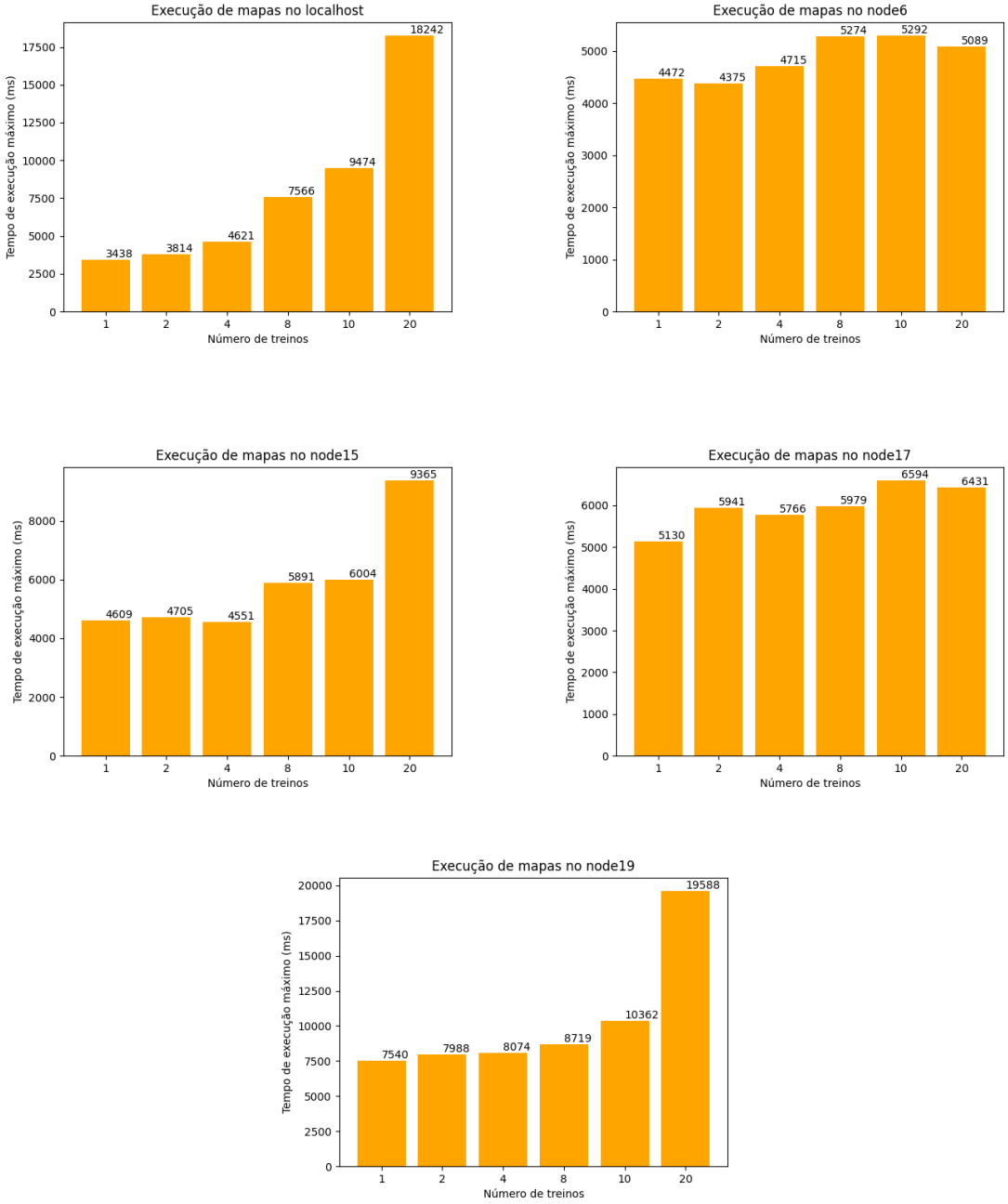


Figura VIII.8: Tempos de execução resultantes de um número de treinos variáveis numa unidade de computação única utilizando o *dataset* Chain.