

**Inteligência Artificial e Sistemas Cognitivos**

Mestrado em Engenharia Informática e Multimédia

**Relatório**

*Projeto*

Rodrigo Dias – 45881

# Introdução

# Inteligência artificial e sistemas cognitivos

# Projeto - Objetivo 1

Esta primeira parte foca-se em estudar o comportamento de redes neuronais com o objetivo de conseguir detetar padrões e semelhanças num conjunto de amostras para ser capaz de prever qual o resultado em dados nunca introduzidos. Esta abordagem baseia-se no comportamento de neurónios biológicos e a ideia é simular o comportamento deles, já que é destes que o cérebro do ser humano é composto e permitem a que sejamos capazes de raciocinar e detetar e reconhecer padrões.

Para este objetivo 1, são usadas técnicas de aprendizagem artificial supervisionada. Isto que dizer que, ao treinar a rede, indica-se qual o resultado esperado de forma a informar a rede se a previsão foi correta ou incorreta, ajustando assim os seus parâmetros até convergir para uma solução aceitável. Numa abordagem não supervisionada já não seria preciso indicar qual o resultado suposto da previsão.

## Realização de um protótipo com base numa plataforma de desenvolvimento de redes neuronais artificiais

### Aprendizagem da função lógica XOR

A picture containing diagram

Description automatically generatedA picture containing text, clock

Description automatically generatedPara realizar a função lógica XOR, pensou-se em implementar uma rede neuronal constituída por 2 entradas e 1 saída. Isto é problemático porque tendo um só neurónio, não existe maneira de criar uma fronteira para distinguir as combinações do XOR pois como se pode ver na F, a expressão de saída do neurónio gera apenas uma reta. A F ilustra os valores possíveis do XOR consoante os inputs e observa-se que uma reta não consegue separar os valores lógicos de 1 (a verde) dos valores lógicos 0 (a vermelho).

Uma solução a este problema seria ter 2 retas de maneira que a área entre elas (AND) possua apenas um valor lógico e a área fora (OR) possua o outro. Para tal, é possível utilizar 2 neurónios na mesma camada escondida, entre as entradas e a saída.

Começou-se por definir a rede neural com 2 entradas, uma camada escondida com 2 neurónios e 1 neurónio de saída. É possível pôr mais neurónios na camada escondida e isso leva a melhor desempenho na aprendizagem, mas decidiu-se medir o desempenho com o mínimo necessário, por isso espera-se que sejam precisas mais iterações para atingir um mesmo valor de erro. Deve-se ter em conta que não se deve colocar muitos neurónios nas camadas escondidas devido ao fenómeno de *overfitting*. Este fenómeno deve-se ao facto de haver muitos pesos para atualizar, levando ao modelo poder ficar demasiado bem treinado, passando, não a aprender, mas sim a memorizar. Quando depois se introduz dados que a rede nunca viu, iria cometer muito mais erros. Neste caso, como o número de entradas é limitado a 4, não seria problema.

A saída dos neurónios é ligada depois a uma função de ativação, podendo limitar os valores de saída através de uma função, como uma *sigmoid*, *tanh*, *relu*, entre outras. Escolheu-se usar a função de ativação *tanh* pois esta limita os valores (calculados pelos respetivos pesos) entre -1 e 1 e decidiu-se começar por fazer os testes da rede com uma configuração bipolar (o valor 0 dos bits é representado por -1) e caso se use a *sigmoid* por exemplo (que limita entre 0 e 1), não era possível obter -1 na saída. Esta função de ativação é importante no neurónio de saída, mas é possível usar outra função nos neurónios da camada intermédia, seria apenas uma questão de ver qual delas é que resultaria em melhor desempenho. Neste caso usou-se a mesma função de ativação (*tanh*) em todos os neurónios.

Usou-se um optimizador SGD (*Stochastic Gradient Descent*) que atualiza os pesos com o algoritmo de *back-propagation*, isto é, para cada iteração, coloca-se um dos conjuntos de valores de entrada na entrada, observa-se o que está à saída (que inicialmente há de ser aleatório pois os pesos são iniciados aleatoriamente) e define-se um valor de erro (neste caso o *mean-squared-error*) que é calculado subtraindo o valor na saída com o valor esperado (*target*) ao quadrado. Com esta medida, é possível calcular o gradiente para cada peso e subtrai-se o valor do peso, pelo valor do gradiente que corresponde a esse peso. Como o gradiente indica a derivada para qual o valor do peso maximiza o valor da função, como se está a falar de uma função de erro, quer-se minimizá-la, daí subtrai-se o valor da derivada correspondente. Esta subtração é feita a multiplicar por um valor de taxa de aprendizagem que permite aproximar mais rapidamente ou lentamente o valor do erro ao mínimo mais próximo. Isto é importante pois se o valor da taxa for muito pequeno, mais iterações são precisas para atingir um mínimo e no caso de ser muito alto, o valor do erro em vez de se aproximar do mínimo, estaria a afastar-se. Por isso, procura-se um valor intermédio que seja adequado.

Para tal, começou-se por medir a rede neural com *momentum* de 0 e com configuração de amostras de treino com ordem fixa e mediu-se o desempenho da taxa de aprendizagem. Registou-se 10 vezes, para cada taxa de aprendizagem, o número de iterações até o valor de erro passar abaixo de 0,1. Testou-se até 1000 *epochs* (iterações) e caso o valor de erro não fosse atingido, regista-se NA.

#### Efeito da taxa de aprendizagem



FIGURA0

É possível observar que há vezes em que o modelo não conseguiu atingir o valor de erro que era pretendido. É possível observar que a taxa de aprendizagem que mais levou a este problema foi a de 0,05, indicando que esta taxa é muito pequena, atualizando os pesos muito lentamente. Os restantes NA indicam que o modelo pode ter ficado preso num mínimo local.

#### Efeito da introdução de um termo de momento

Fez-se a mesma coisa, mas colocou-se um termo de momento de 0,5 e outro de 1. Isto serve para ajudar a “dar um empurrão” ao valor do erro para dar hipótese de sair de um mínimo local.



FIGURA



FIGURA

Com estes resultados, é possível observar que com momento de 0,5, o modelo conseguiu atingir o valor de erro pretendido com menos iterações que com momento 0, indicando que ajuda a sair dos vários mínimos locais que possa ter encontrado, indicando também que a taxa de aprendizagem mais adequada é a intermédia, por volta de 0,5.

Com momento de 1, repara-se que voltam a ser registados muitos mais valores NA, indicando que o valor de momento é demasiado grande e que está a “dar um empurrão” com muita intensidade.

#### Efeito da apresentação das amostras de treino com ordem fixa ou aleatória

Voltou-se a colocar o valor de momento a 0.5, pois foi o melhor dos testes previamente feitos, para agora testar o desempenho ao tornar a configuração com amostras de treino de ordem aleatória. São esperados ligeiramente melhores resultados pois ao treinar o modelo com ordem aleatória, exclui-se a hipótese de ele aprender qualquer padrão da ordem a que lhe são apresentados os valores de treino.



FIGURA

Pelos resultados obtidos, continua-se a ver que a taxa de aprendizagem perto de 0,5 é a mais adequada, pois tem uma média de iterações mais reduzida, e não tem valores NA comparando com os valores vizinhos. Em comparação com a F0, vê-se que a melhor taxa é a 0,5 porque possui um termo de momento, enquanto sem termo de momento, sem “empurrão”, a melhor acaba por ser taxa de 1, chegando ao mínimo mais rapidamente, mas sem maneira para sair de mínimos locais.

#### Efeito de uma codificação binária ou polar

Para este teste, voltou-se a pôr as amostras com ordem fixa e mudou-se para configuração binária. De maneira a comparar os resultados, não se alterou as funções de ativação, mantendo-as funções de *tanh*. No entanto faria sentido usar ativações como a *sigmoid* que limita os valores entre 0 e 1.



FIGURA

É possível observar que os resultados obtidos estão extremamente maus e isto deve-se exatamente ao estar a usar a ativação *tanh* permitindo uma maior gama de valores (entre -1 e 1) sendo mais complicado a rede neural entender que os valores devem ser ajustados entre 0 e 1.

### Aprendizagem de padrões de imagem

Com o objetivo de concretizar uma rede neural para deteção de padrões, implementou-se um modelo com 16 entradas (para os inputs dos pixéis de padrões 4x4) e 4 saídas, já que se terá 4 padrões diferentes para detetar, ilustrados na F. Caso seja detetado o padrão A, então as saídas estarão a [1,0,0,0], caso seja detetado o B, as saídas serão [0,1,0,0] e caso não seja nenhum padrão as saídas serão [0,0,0,0].



Como este é um problema complexo, criou-se uma rede neural com 32 neurónios na única camada intermédia e geraram-se amostras de treino com várias amostras A, B, C e D. Fez-se um algoritmo para gerar mais alguns padrões aleatórios, atribuindo o conjunto de saídas correspondente, tendo em conta que, apesar de gerado aleatoriamente, pode ter resultado num dos padrões definidos.

Chart

Description automatically generatedDesta vez utilizou-se um optimizador Adam com as configurações predefinidas pois resultou em melhor desempenho do que com aquele analisado no exercício anterior. Para treinar o modelo, considerou-se usar *validation\_split* para ajudar no desempenho, já que se carregam bastantes amostras de treino.

As funções de ativação aqui usadas foram a *sigmoid* à saída (pois os valores de saída são 0 ou 1) e a *relu* para a camada intermédia, pois foi a que deu melhores resultados após alguns testes.

Text

Description automatically generatedO processo é semelhante ao exercício anterior e após treinar o modelo com 200 iterações registou-se o desempenho do erro e da precisão para cada iteração, podendo observar na F que o erro está bastante baixo em contraste com a precisão, que está acima de 90%, indicando um desempenho muito bom na aprendizagem. Ao carregar depois o modelo, introduz-se um conjunto de padrões para ver o desempenho final e verifica-se que o modelo acerta nestes, como mostra a F.

## Aplicação de redes neuronais artificiais para resolução de um problema de escolha livre

A picture containing text, crossword puzzle

Description automatically generatedDecidiu-se fazer uma rede neural para resolver *Nonograms*, *puzzles* de lógica que consistem em preencher as células de uma matriz consoante os números indicados à esquerda das linhas e no topo das colunas, como ilustrado na F. No caso de uma linha ter os números “2 2”, indica que vão ser preenchidos dois pares de células da matriz separados por, pelo menos, uma célula em branco.

Para tal, o objetivo é colocar nas entradas da rede neural os números das linhas e colunas da matriz de forma a colocar na saída a matriz resolvida. Assumindo que se usará matrizes de 4x4, serão precisos 16 neurónios de saída. Para os neurónios de entrada, decidiu-se usar um para cada linha/coluna, caso contrário, o número de entradas vai depender do número de células preenchidas em cada linha/coluna. Imaginando que uma linha tem os números “2 1”, coloca-se no neurónio de entrada o valor 21, se tiver “1 1”, coloca-se o valor 11, se tiver “1”, coloca-se 1, etc. Dessa maneira, o número de neurónios de entrada serão 8, 4 para os números correspondentes às linhas e os outros 4 para os números das colunas.

Para treinar a rede neural, gerou-se primeiro um *array* de *arrays* únicos com 16 posições preenchidas com 1’s e 0’s aleatoriamente. Estes *arrays* de 16 posições são as matrizes quadradas transformadas de 4x4 para 1x16, que são já os valores que se espera ter à saída. Por isso, foi necessário um algoritmo que interpretasse essas matrizes para obter os números a serem introduzidos na entrada.

Após gerar as amostras configurou-se uma rede neural semelhante ao exercício anterior, mas com 3 camadas intermédias, cada uma com 1024 neurónios. A razão para esta configuração deve-se ao facto deste problema ser muito mais complexo e precisar de muitos mais pesos de forma a evitar *underfitting*, que é semelhante ao que acontece com o problema do XOR ao não introduzir pelo menos 2 neurónios na camada intermédia. Experimentou-se com valores maiores, mas tornava a complexidade muito maior e demorava muito mais tempo para treinar o modelo, sem um aumento muito significativo nos resultados. As funções de ativação foram semelhantes, na saída uma *sigmoid* e nas camadas intermédias *relu*, pois após alguns testes, esta foi a combinação que pareceu dar melhores resultados.

Após treinar o modelo, registaram-se os valores de erro e de precisão, tanto do conjunto de treino como o conjunto de validação (que contém 20% das amostras de treino) e, como mostra a F, observa-se que o modelo treinou até uma precisão acima de 90% e a Chart

Description automatically generatedprecisão do conjunto de validação perto dos 70%. Estes valores são distintos pois o conjunto de validação serve para ir testando o modelo enquanto treina e essas amostras são desconhecidas, logo tende a errar mais facilmente. A mesma coisa acontece para os erros.

Text

Description automatically generatedDe seguida testou-se o modelo com várias amostras e obteve-se valores próximos de 10% de erro e 80% de precisão, como se vê na F. Introduziu-se uma amostra *sample* para visualizar o resultado de saída e também é possível observar que o resultado foi o igual esperado.

# Projeto - Objetivo 2

Nesta segunda parte implementam-se algoritmos de raciocínio automático para otimização para resolver problemas como o N-Rainhas e o Caixeiro viajante. Este tipo de algoritmos tem como objetivo encontrar a (ou as) solução ótima a partir de um critério de avaliação ou função de desempenho para um determinado problema.

O código foi estruturado tendo uma classe para cada problema com os métodos adequados, uma classe para agrupar os algoritmos de *hill-climbing* estocástico, *hill-climbing* com reinício aleatório e *simulated annealing* e outra classe para agrupar as funções para o algoritmo genético, como ilustra a F.

DIAGRAMA DE CLASSES

#### N-Rainhas

O problema das N-Rainhas consiste em encontrar uma configuração de um tabuleiro NxN com N rainhas dispostas nesse tabuleiro de maneira que nenhuma intercete com outra na horizontal, vertical ou diagonal.

O critério de avaliação para este problema será o número de colisões entre as rainhas, sendo que, quanto maior o valor, pior e pretende-se encontrar o estado com 0 colisões (solução ótima).

Cada estado é disposto num *array* de tamanho N em que cada *index* representa a posição da rainha no tabuleiro. Ou seja, cada *index* do *array* representa uma coluna do tabuleiro e o número nesse *index* indica a linha do tabuleiro em que a rainha está. Com este tipo de configuração diminui-se a complexidade forçando uma rainha por coluna, evitando só que haja colisões na horizontal e na diagonal.

Os estados vizinhos serão aqueles que, a partir de um estado atual, se move apenas uma das rainhas, ou seja, apenas se muda um dos números no *array*, movendo uma rainha apenas na vertical.

#### Caixeiro viajante

O problema do Caixeiro viajante consiste em encontrar o caminho entre as cidades que minimiza a distância percorrida sem percorrer duas vezes pela mesma cidade e voltando à cidade inicial.

Aqui o critério de avaliação é a distância total do percurso, sendo que, quanto maior o valor, pior e pretende-se encontrar um estado com a menor distância possível.

Cada estado será composto por *tuplos*, que indicam as coordenadas de cada cidade, dispostos num *array*, sendo a ordem dessa disposição no *array* a ordem das cidades que serão percorridas.

Os estados vizinhos serão obtidos ao trocar uma cidade com outra qualquer. Ao calcular depois a distância, é colocado a cidade inicial no final do *array* calculando o percurso de volta à cidade que começou.

## Realização de uma biblioteca de métodos raciocínio automático para otimização

Neste ponto implementaram-se algoritmos que iniciam o problema a ser resolvido com uma configuração (estado) aleatória, a partir da qual se foi procurando a solução ótima, obtendo sempre um estado vizinho e avaliando se esse estado vizinho é melhor ou pior de acordo com o critério de avaliação do problema.

### Hill-Climbing estocástico

Neste algoritmo, a partir de um estado inicial aleatório, escolhe-se um estado vizinho que tenha a melhor avaliação de todos os vizinhos encontrados e repete-se até se encontrar um vizinho com estado pior ou até estar preso (encontrar os mesmos vizinhos várias vezes. Se isto acabar por acontecer é porque o estado chegou a um ótimo local e não consegue sair (preso) ou para sair só encontra estados com avaliação pior. Esta segunda opção é mais comum acontecer caso tenha encontrado a solução global, pois todos os outros estados são piores que o final, no entanto é possível encontrar um estado local cujos vizinhos sejam todos piores sem este ser a solução ótima.

### Hill-Climbing com reinício aleatório

Chart, surface chart

Description automatically generatedDe modo a resolver este problema de não encontrar solução ótima por ficar preso num ótimo local, implementou-se o algoritmo de *hill-climbing* com reinício aleatório. Este algoritmo basicamente faz o mesmo que o anterior, mas repete várias vezes e, para cada iteração, inicia com um estado aleatório. Isto faz com que se explore mais soluções a partir de pontos iniciais diferentes, dando a oportunidade a que algum deles encontre a solução ótima global. A F ilustra como este algoritmo ajuda a explorar os vários estados possíveis.

A cada iteração guarda-se o melhor estado encontrado, sendo mais provavelmente o estado ótimo, para no final apresentar como solução.

No entanto, se o problema for demasiado complexo há sempre hipótese de não chegar a encontrar um ótimo global. Outro problema é que para este tipo de algoritmo, é preciso procurar qual o estado vizinho que possui a melhor avaliação, levando a que se tenha de verificar todos os estados vizinhos possíveis e claro que, para problemas muito complexos, mais poder computacional será preciso e mais tempo irá demorar a encontrar qualquer solução.

#### A screenshot of a computer Description automatically generated with low confidenceN-Rainhas

Para resolver este problema, usou-se o algoritmo de *hill-climbing* com reinício aleatório com 25 iterações, com um tabuleiro com 8 rainhas. A F mostra o resultado final, verificando que não existem colisões entre rainhas.

#### Caixeiro viajante

Chart, scatter chart

Description automatically generatedChart, line chart

Description automatically generatedPara este problema, usou-se também o algoritmo de *hill-climbing* com reinício aleatório com 25 iterações, com 20 cidades num mundo de 100x100. A F mostra onde as cidades estão localizadas e a F mostra o percurso obtido e é possível observar que a ligação entre cidades está bem otimizada.

### Simulated Annealing

O algoritmo de *simulated annealing* foi pensado de forma a otimizar o desempenho dos algoritmos anteriores, não tendo de analisar todos os estados vizinhos possíveis para encontrar o melhor. Neste caso, o algoritmo escolhe um estado vizinho aleatório e caso o novo vizinho tenha melhor valor, atualiza o estado atual para continuar o processo. No entanto, caso seja pior, dá-se uma chance de atualizar o estado atual para esse estado pior. Isto deve-se ao facto de, caso o estado esteja preso num ótimo local, há possibilidade de andar para trás para o tentar evitar e possivelmente encontrar estados melhores.

Esta lógica é implementada dando a possibilidade a que no início se possa atualizar o estado atual por um pior, mas que ao longo do tempo esta hipótese diminua. Assim é possível explorar os estados iniciais mais aleatoriamente até ir reduzindo apenas aos melhores.

Esta implementação melhora drasticamente o desempenho pois procura um estado vizinho aleatório sem precisar de verificar todos os possíveis. No entanto, há possibilidade de não encontrar a solução ótima pois está a depender de uma fórmula de têmpera (*annealing*) e quando a temperatura chegar a 0, este processo de escolher o vizinho acaba, sendo o resultado o último estado atualizado. Este algoritmo é útil pois há um *trade-off* entre tempo de processamento e a solução mais ótima, ou seja, o tempo de processamento é reduzido drasticamente, mas a solução encontrada não é a melhor, no entanto, é bastante boa.

A fórmula de têmpera refere-se a que o algoritmo começa flexível (quente), permitindo escolher piores estados vizinhos, e ao longo do tempo torna-se mais restrito (arrefece), diminuindo a hipótese de os atualizar. Esta fórmula depende dessa temperatura e da diferença de avaliações entre o estado atual e o vizinho escolhido aleatoriamente. A temperatura vai diminuindo ao longo do tempo, diminuindo a probabilidade de escolher um vizinho mau, e a diferença de avaliações, quanto mais negativa for (se for positiva esta fórmula não é usada e o estado atual é logo atualizado), menor é a hipótese de atualizar com esse estado.

Chart

Description automatically generatedA chave está em escolher uma função de temperatura que permite maior flexibilidade ao início e no fim dê mais tempo para aprofundar. A função ideal seria algo semelhante à ilustrada na FTEMP.

#### A picture containing chart Description automatically generatedN-Rainhas

Desta vez usou-se o algoritmo *simulated annealing* com uma função de temperatura , que tem o comportamento igual à da FTEMP. A F mostra o resultado obtido por este algoritmo e vê-se que foi possível encontrar a solução ótima.

#### Caixeiro viajante

Para este problema, usou-se a mesma função de temperatura (FTEMP), com as mesmas cidades usadas no *hill-climbing* e observa-se na F que a solução não é tão boa como a outra, mas a distância também não é muito pior e a olho vê-se que a solução é bastante adequada, visto que o poder computacional necessário foi muito mais reduzido.

Chart, line chart

Description automatically generated

## Realização de uma biblioteca de algoritmos genéticos

Para este exercício foi implementado um algoritmo genético e este baseia-se em princípios da natureza como na seleção, cruzamento e mutação de indivíduos de uma população. Isto significa que, para um conjunto de indivíduos (estados) faz-se seleção de 2, tendo em conta uma função de adaptação (que dá maior probabilidade a indivíduos mais adaptados de serem escolhidos), faz cruzamento entre eles, ou seja, trocam-se parâmetros do estado entre eles para criar um novo, e com uma probabilidade baixa, realiza-se mutação para dar oportunidade de haver diversidade (com o objetivo de evitar soluções locais). Após estas operações, o individuo resultante é adicionado a uma nova população e o processo repete-se até gerar uma nova população com o mesmo número de indivíduos que a inicial. Com a nova população gerada, repete-se todo o processo um número de vezes suficiente até obter uma solução adequada.

Para pôr o algoritmo a funcionar é necessário ter em conta o tipo de problema que se pretende resolver pois, por exemplo, no caixeiro-viajante não pode haver cidades repetidas, então tem de se implementar um algoritmo de mutação e de cruzamento que tenha isso em conta, ao contrário do N-Rainhas que pode trocar apenas um número (dentro do limite do tabuleiro).

Para tal, adicionaram-se os métodos necessários para este algoritmo às classes dos problemas como mostra a F. Estes foram o método de reprodução (cruzamento), função de adaptação e a função para gerar a população inicial. Uma coisa que não foi feita e seria mais adequado fazer era também fazer a função e mutação como método de cada problema pois o tipo de mutação é diferente no exemplo dos problemas que se está a resolver, onde no N-Rainhas pode haver indivíduos com genes repetidos e no Caixeiro viajante não.

#### N-Rainhas

#### Caixeiro viajante

## Estudo e utilização de uma outra plataforma de algoritmos genéticos

# Projeto - Objetivo 3

# Tema de escolha livre

# Conclusão