ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Relatório Final do Projeto

Inteligência Artificial e Sistemas Cognitivos

Nome:

Duarte Valente | A47657

Docente:

Eng. Luís Morgado

Curso: MEIM

2024

Índice

Índice de ilustrações

Introdução

Este relatório descreve o projeto realizado no âmbito da cadeira de Inteligência Artificial e Sistemas Cognitivos, onde foram exploradas e aprendizado de técnicas avançadas de Machine Learning (ML), também conhecida como aprendizagem automática. Assim, o projeto visa abordar problemas específicos por meio de diferentes abordagens baseadas em ML, com foco em técnicas de Redes Neurais Artificiais, Aprendizagem por Reforço e Aprendizagem por Raciocínio Automático e de Arquitetura de Agentes Deliberativos.

a) Objetivos do Projeto

O principal objetivo do projeto passa por adquirir conhecimentos práticos e teóricos sobre as diferentes técnicas de ML mencionadas e aplicá-las na resolução de problemas reais. A proposta é explorar o potencial dessas abordagens em diferentes contextos, avaliando sua eficácia, limitações e possíveis melhorias.

b) Estrutura do Relatório

O presente relatório está estruturado em três diferentes secções que abrangem os conceitos fundamentais de cada uma das técnicas exploradas, seguido pela aplicação prática desses conceitos nos problemas sugeridos, os resultados e a analise e conclusões dos mesmos.

Parte 1 – Redes Neuronais Artificiais

1.1 Introdução teórica

1.1.1 Conceitos Fundamentais

As Redes Neuronais Artificiais (RNAs) são um tipo de modelos computacionais inspirados no funcionamento do cérebro humano. Pois são compostas por unidades interconectadas chamadas neurônios. Estas redes têm a capacidade de aprender padrões e representações complexas a partir de dados, pois a sua estrutura hierárquica permite a extração de características abstratas, tornando-as ferramentas poderosas em tarefas de reconhecimento de padrões, classificação e regressão.

1.1.2 Arquitetura de uma Rede Neural

Uma RNA é tipicamente organizada em camadas. Uma camada de entrada, uma ou mais camadas ocultas e uma camada de saída. Para a comunicação entre camadas, cada neurônio de uma certa camada encontra-se conectado a todos os neurônios na camada subsequente, formando uma matriz de pesos. Pesos esses, que tomam certos valores que vão sendo são ajustados durante o processo de treino da rede.

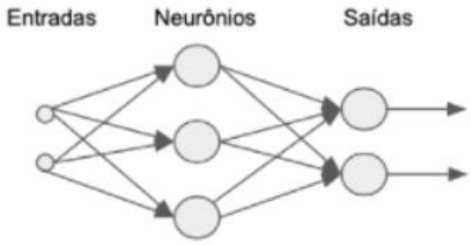

Figura 1 – Exempro de uma arquitetura de uma RNA

Como por exemplo a arquitetura da RNA presente na figura 1, a camada de entrada recebe os dados brutos em dois neurónios, enquanto que os três neurónios da camada oculta realizam transformações progressivas dos dados, de forma a gerar um resultado final na camada de saída, que neste caso retorna dois outputs.

1.1.3 Processo de treino de uma Rede Neural

Relativamente ao processo de treino de uma RNA, este envolve a apresentação iterativa de dados ao modelo, ajustando os pesos das conexões entre camadas, para minimizar a diferença entre as saídas previstas e as saídas pretendidas.

Para realizar esse processo de ajuste de pesos, são normalmente utilizados algoritmos de otimização, como por exemplo o método de Descida de Gradiente que procura encontrar um valor mínimo de uma função ao efetuar várias iterações, tomando cada passo iterativo em direção (negativa) do gradiente, correspondente à direção de declive máximo. O que possibilita de encontrar os melhores valores para os quais os pesos da Rede produzem um resultado na camada de saída, mais similar com o pretendido.

1.2 Implementação

1.2.1 Implementação da função XOR com uma rede neuronal do tipo perceptrão

Como primeiro, problema para a implementação de uma rede neuronal, foi sugerido implementar a função XOR. Função essa, bastante simples que recebe como entrada dois valores e classifica os mesmos em 0 caso o seu valor seja igual e em 1 caso o seu valor seja diferente.

Logo, tratando-se de um problema de classificação binaria, e como temos dois valores de entrada, vamos necessitar de uma arquitetura com dois neurónios na camada de entrada (*input layer*) e uma camada de saída (*output layer) com um neurónio.*

No entanto, para resolver o problema, se considerássemos utilizar uma única camada de neurónios, não conseguiríamos encontrar uma fronteira de separação linear que resolvesse o problema, pois como podemos observar na figura 2, necessitamos de duas fronteiras para classificar os dados de input de uma função XOR.

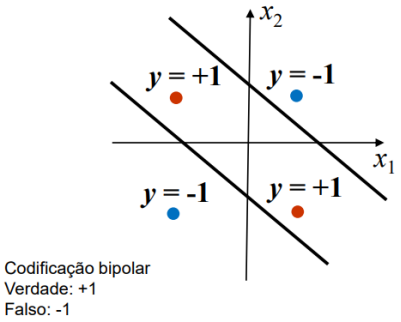

Figura 2 - Disjunção exclusiva (XOR)

Assim, para resolver o problema da separabilidade linear, podemos utilizar o perceptão, que utiliza pesos e um pendor para definirem uma fronteira linear entre regiões.

Para resolver o problema, foi então utilizada a seguinte arquitetura de rede:

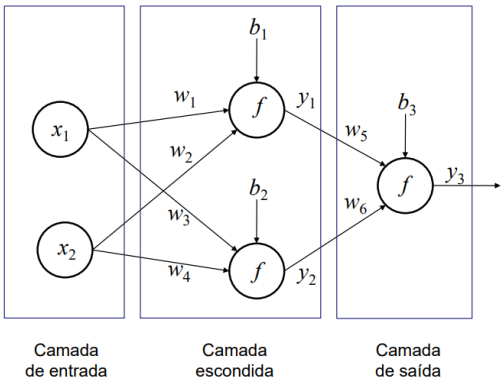

Figura 3 - Arquitetura da rede XOR

Assim, para desenvolver a rede neural que implemente a função XOR, após verificado que necessitaríamos de dois neurônios de entrada, dois neurônios na camada oculta e um neurônio na camada de saída. Foi desenvolvido um pequeno código na linguagem *Python*, onde foi implementada a função **xor_perceptron()** que configura esta rede. Começando nomeadamente pelos pesos e pendores que foram configurados de acordo com os valores estudados em aula:

- Camada Oculta:
  - Neurónio 1: w1 = 1, w2 = -1, b1 = -0.5;
  - Neurónio 2: w3 = -1, w4 = 1, b2 = -0.5;
- Camada de Saída:
  - Neurónio de Saída: w5 = 1, w6 = 1, b3 = -0.5;

Para a função de ativação, optamos pela função degrau, definida como:

$$f(x) = degrau(x) = \begin{cases} 1 \; se \; x > 0 \\ 0 \; se \; x \leq 0 \end{cases}$$

Figura 4 - Função de ativação

Após estas configurações, definimos os dados de entrada na matriz inputs:

[[0, 0], [0, 1], [1, 0], [1, 1]]

Por fim, foram então propagados os dados pela rede ao chamar a função **xor_preceptron()** tendo obtido os seguintes resultados:

```
Input: [0 0], Output: 0
Input: [0 1], Output: 1
Input: [1 0], Output: 1
Input: [1 1], Output: 0
```

Figura 5 - Resultados da rede XOR

Ao analisar os resultados, verificamos que estes se encontram de acordo com o pretendido, logo podemos concluir, que a arquitetura utilizada e os valores configurados manualmente dos pesos e pendores, permitem que a rede implemente corretamente a função XOR.

1.2.2 Implementação de uma rede neuronal multicamada e dos operadores OR, AND, NOT e XOR.

Como segundo, desafio, foi sugerida a implementação de uma rede neural multicamada (também conhecidas como redes neurais profundas) que acaba por ser uma extensão natural da abordagem anterior da rede do tipo perceptão.

Em relação ao intuito desta rede, pretende-se incorporar mais camadas ocultas, de forma a que a rede ganhe a capacidade de aprender representações mais complexas e realizar tarefas logicamente mais desafiantes. Assim, nesta fase podemos explorar a implementação de uma rede neural multicamada que implemente as funções dos operadores lógicos OR, AND, NOT e XOR.

Logo, para realizar a implementação desta rede, foi utilizada a linguagem *Python* e a biblioteca *NumPy*. Sendo então desenvolvida a classe **RedeNeuronal.** Responsável por estender e incluir as novas camadas ocultas. O que permitiu uma maior flexibilidade na aprendizagem de padrões mais complexos. Em relação à função de ativação utilizada foi escolhida uma função sigmoide, e foi utilizado um algoritmo de retro propagação para ajustar os pesos durante o processo de treino.

Em relação aos dados de entrada, foi definido o conjunto **X** para os operadores:

OR, AND e XOR: X = **[[0, 0], [0, 1], [1, 0], [1, 1]]** NOT: X = **[[0], [1]]**

E sendo necessário valores de comparação para calcular o erro de forma a ajustar os pesos e pendores da melhor manira possível foi criado o conjunto **Y** respetivo aos valores de output pretendido para cada uma das funções:

- OR: Y = **[[0], [1], [1], [1]]**
- AND: Y = **[[0], [0], [0], [1]]**
- NOT: Y = **[[1], [0]]**
- XOR: Y = **[[0], [1], [1], [0]]**

Passando a uma explicação mais detalhada da implementação podemos descrever os diferentes métodos implementados na classe **RedeNeuronal**:

- **__init__:** Método de inicialização da classe que inicializa os pesos e os pendores com valores aleatórios pois estes vão ser ajustados durante o treino;
- **train:** Método responsável por treinar a rede, de forma a encontrar os melhores valores dos pesos e pendores no número de épocas fornecido;
    - Propaga pela rede os dados de entrada;
    - Utiliza a função de ativação sigmoid que retorna valores positivos e é representada pela seguinte equação:

$$\phi(x) = \frac{1}{1 + e^{-x}}$$

Figura 7 - Equação da função sigmoide

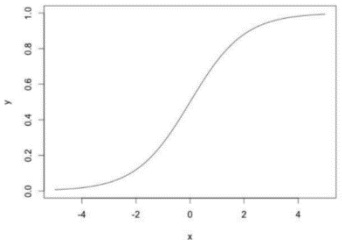

Figura 6 - Gráfico da equação sigmoide

    - Calcula o erro para os valores propagados ao comparar com os valores do conjunto **Y.**
    - Efetua a retro propagação dos dados de forma a minimizar a função de perda, que quantifica o quão distantes as previsões do modelo estão dos valores reais;
    - São atualizados os valores dos pesos e pendores, tendo em conta o resultado da retro propagação efetuada;
- **predict:** Método que simplesmente propaga os valores pela rede e retorna os valores obtidos na camada de saída da rede;

Assim, após o desenvolvimento desta classe, foi então possível, criar uma nova Rede, indicando os valores dos parâmetros da camada de entrada (que no caso era igual a um, pois é um array de valores), o número de neurónios da camada escondia (dois neurónios) e o número de neurónios da camada de saída (um neste caso).

Após a criação da rede, foram definidos os valores da taxa de aprendizagem que determina a magnitude dos ajustes feitos nos pesos e número de épocas. E foi então chamado o método treinar da classe **RedeNeuronal** para treinar com os diferentes dados.

No fim da fase de treino da rede, é então chamado o método **predict** para propagar novamente os dados pela rede, no entanto pela rede já treinada, de forma a obter os melhores resultados que a rede conseguiu chegar.

Para concluir, podemos observar os ressoldados destas mesmas previsões que se encontram compreendidos entre 0 e 1 tendo em conta que foi utilizada uma função de ativação sigmoid, pelo que se um valor previsto foi maior que 0.5 podemos classificar como 1 e se for menor como 0.

```
Operador lógico OR:
Entrada: [0 0], Saída Prevista: 0.04
Entrada: [0 1], Saída Prevista: 0.98
Entrada: [1 0], Saída Prevista: 0.98
Entrada: [1 1], Saída Prevista: 0.99
```

```
Operador lógico AND:
Entrada: [0 0], Saída Prevista: 0.00
Entrada: [0 1], Saída Prevista: 0.03
Entrada: [1 0], Saída Prevista: 0.03
Entrada: [1 1], Saída Prevista: 0.95
```

```
Operador lógico NOT:
Entrada: [0], Saída Prevista: 0.97
Entrada: [1], Saída Prevista: 0.03
```

```
Operador lógico XOR:
Entrada: [0 0], Saída Prevista: 0.07
Entrada: [0 1], Saída Prevista: 0.94
Entrada: [1 0], Saída Prevista: 0.92
Entrada: [1 1], Saída Prevista: 0.06
```

Figura 8 - Previsões da Rede

Para alem das previsões podemos também observar a evolução do erro durante as diferentes épocas para cada uma das operações:

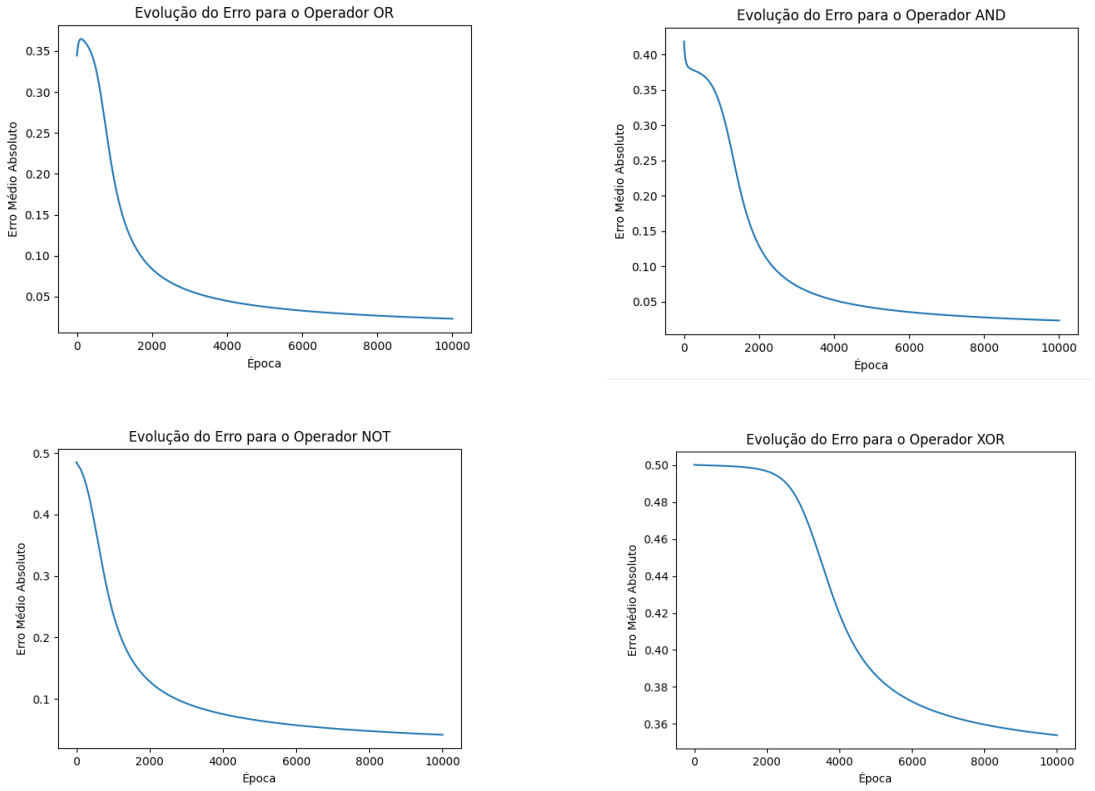

Figura 9 - Evolução do erro

1.2.3 Implementação de uma rede neuronal multicamada com capacidade de aprendizagem e efeitos da taxa de aprendizagem, termo de momento e amostragem fixa e aleatória.

Neste capítulo, foi explorada a implementação de uma Rede Neural Multicamada com foco na influência de parâmetros, como taxa de aprendizagem, termo de momento e ordem de apresentação dos dados no treinamento da rede.

Estes parâmetros acabam ter uma alta relevância nestas redes pois afetam diretamente o ajuste dos pesos da rede, sendo que o principal objetivo passa por analisar o efeito que cada um causa e como podemos otimizar estes valore para chegar ao melhor resultado de previsões possível.

No entanto, para uma melhor compreensão podemos descrever também estes parâmetros de forma a entender as suas funções e onde podem afetar o desempenho na rede.

Taxa de aprendizagem

A taxa de aprendizagem é um hiperparâmetro que determina a magnitude dos ajustes feitos nos pesos da rede durante o treino. Ajustes esses que desempenham um papel fundamental na convergência e estabilidade no processo de otimização da rede.

Em relação aos seus valores, podemos afirmar que se estes forem bastante altos, o algoritmo pode oscilar ou divergir. Mas por outro lado, se o valor da taxa de aprendizagem for demasiado baixo, o processo de treino pode demorar bastante tempo ou ficar preso em mínimos locais. Logo, é importante encontrar um valor que se adeque ao problema e que consiga efetuar os ajustes necessários para chegar ao melhor resultado possível.

Termo de momento

Em relação ao termo de momento, este tem o papel de acelerar o processo de treino e melhorar a convergência da rede, especialmente em casos onde o gradiente tem variações significativas ou há presença de ruídos nos dados.

Amostragem Fixa ou Aleatória

A forma como apresentamos os dados ao modelo também importa, pois se a rede treinar com os dados fornecidos sempre pela mesma ordem, esta pode identificar padrões que não são relevantes para o problema, ou correndo o risco de *overffiting*, onde a rede fica apta para prever um certo conjunto de dados, mas quando lhe apresentarmos um conjunto diferente, esta pode diminuir drasticamente a sua preformasse a efetuar previsões.

Assim, para implementar a rede tendo em conta estes parâmetros, foi utilizada a biblioteca Keras que facilita a criação dos modelos e configurações das respetivas camadas. O que foi essencial, nesta fase, uma fez que o foco passou por analisar a influencia dos diferentes parâmetros propostos.

Em relação à implementação da rede, foi criada uma classe **RedeNeuronal** à semelhança da rede anterior onde:

- São recebidos os valores dos parâmetros da taxa de aprendizagem, termo de momento e ordem de apresentação;
- É criado o modelo pelo método **criar_modelo();**
- O método **criar_modelo():**
    - Define o tipo de modelo do Keras, que no caso foi do tipo Sequencial() onde cada camada possui exatamente um tensor de entrada e um tensor de saída;
    - Adiciona as camadas do modelo (1 camada de entrada com 2 neurónios e uma camada de saída com função de ativação sigmoid à semelhança da rede anterior);
    - Inicializa o optimizador da rede do tipo SGD (*Stochastic Gradient Descent)* que é um algoritmo de descida de gradiente que utiliza a taxa de aprendizagem e o termo de momento que controla a influencia dos gradientes anteriores nas atualizações dos pesos.
    - Por fim, compila o modelo, com um Loss de Erro Quadrático Médio o que quantifica o quão bem o modelo está a realizar a tarefa e com o optimizador já inicializado anteriormente;
- O método **treinar():**
    - Verifica se a ordem de apresentação dos dados ao modelo é aleatória, pois caso seja, baralha os dados e as respetivas labels antes de propagar os dados pelo modelo;
    - Propaga os dados pelo modelo;
    - Guarda em memoria a história de evolução do *loss* do modelo para o podermos apresentar mais à frente;
- O método **prever():**
    - Utiliza a função **predict()** do keras para obter as previsões do modelo para os dados;

Após ser explicado o funcionamento detalhado da classe **RedeNeuronal**, foi então executado um pequeno código para instanciar uma Rede com base nesta classe e visualizar os seus resultados.

Foi também, nesta esta fase onde foi importante definir os valores dos parâmetros da rede a serem testados assim como os dados de entada e as respetivas labels. Em relação ao conjunto de dados de entrada e labels, estes são os mesmo que no exemplo anterior já que se trata da função XOR.

X = **[[0, 0], [0, 1], [1, 0], [1, 1]]**

Y = **[[0], [1], [1], [0]]**

Em relação aos parâmetros: taxa de aprendizagem, termo de momento e ordem de apresentação, foram efetuadas algumas combinações de forma a perceber o efeito no desempenho do modelo que cada um causa. Assim, foram definidas 8 diferentes combinações que combinam os valores de taxa de aprendizagem 0.1 ou 0.5, termo de momento: 0.1 ou 0.9 e taxa de amostragem fixa ou aleatória.

Após definir estas configurações, foram então criadas e trainadas 8 diferentes redes, das quais foram obtidos os seguintes resultados:

Configuração	1	2	3	4	5	6	7	8
Taxa de aprendisagem	0.1	0.1	0.5	0.5	0.1	0.1	0.5	0.5
Termo de momento	0.1	0.9	0.1	0.9	0.1	0.9	0.1	0.9
Ordem de amostragem	fixa	fixa	fixa	fixa	aleatória	aleatória	aleatória	aleatória
Loss	**0.243**	**0.002**	**0.007**	**0.125**	**0.249**	**0.007**	**0.145**	**0.167**

Figura 10 - Configurações e valor de Loss obtido

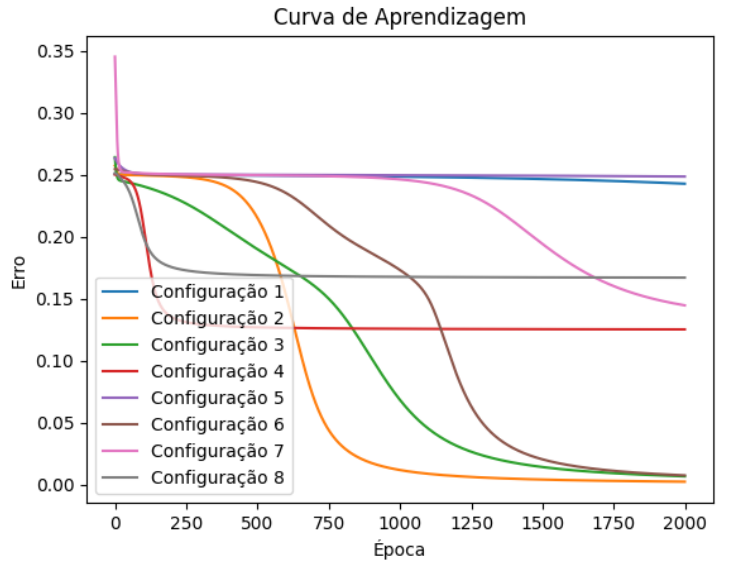

Figura 11 - Curva de aprendizagem de cada Rede

Ao analisarmos os resultados obtidos, podemos verificar que a rede que obteve o melhor desempenho foi a com a configuração 2, com taxa de aprendizagem igual a 0.1, termo de momento de 0.9 e ordem de amostragem fixa, no entanto podemos também retirar algumas conclusões ao verificar no geral quais foram os melhores valores para cada parâmetro.

Por exemplo, ao analisarmos os as duas redes com os piores desempenhos (1 e 5), verificamos que estas foram as redes onde tanto a taxa de aprendizagem como o termo de momento tinham o menor valor, o que comprova o que foi dito anteriormente de que se estes valores fossem demasiado baixos o modelo poderia levar bastante tempo a aprender ou ficar preso em mínimos locais.

Por outro lado, se olharmos para as três melhores redes (2, 3 e 6), podemos chegar à conclusão que não existiu um destaque de um tipo de configuração. Pois por exemplo a rede 3 que possuía uma taxa de aprendizagem mais alta e um termo de momento mais baixo obteve um desempenho muito similar à rede 6 com taxa de aprendizagem baixa e termo de momento mais alto. E o mesmo em relação à rede 2 que é igual à rede 6 mas com ordem de amostragem fixa. O que conclui que podem existir diferentes configurações que conseguem chegar ao resultado pretendido.

Já em relação à ordem de amostragem, também foi possível verificar que não ouve uma melhoria significativa no desempenho das redes. Também devido ao facto de o problema ser simples e os dados de entrada serem reduzidos. No entanto para um dataset maior, esta política de amostragem pode trazer resultados mais significativos e evitar possíveis problemas de overffiting.

1.2.4 Resolução do problema de classificação de padrões

Por fim, como última tarefa da primeira parte do projeto, foi sugerido desenvolver uma RNA para classificar dois diferentes padrões (figura 12).

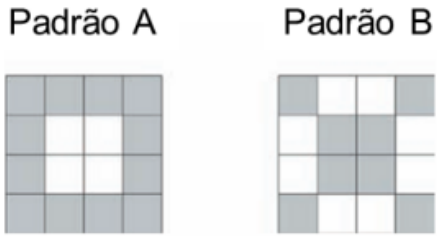

Figura 12 - Padrões a serem classificados

Logo, para resolver este problema, foi verificado que a rede destinada a aprender a resolver este problema, necessitava de receber como entrada um input que representasse estes mesmos padrões e que gerasse um output que classificasse o padrão. No caso 0 se identificasse o padrão como o padrão A e B se identificasse como o padrão B.

Assim, como dados de entrada da rede, foram criados dois *arrays* que representam os padrões e as respetivas labels.

```
# padrão A
A = np.array([1, 1, 1, 1,
              1, -1, -1, 1,
              1, -1, -1, 1,
              1, 1, 1, 1])

# padrão B
B = np.array([1, -1, -1, 1,
              -1, 1, 1, -1,
              -1, 1, 1, -1,
              1, -1, -1, 1])

# labels respetivas
y = np.array([0, 1])
```

Figura 13 - Dados de entrada e classes pretendidas

Já para construir a rede, o processo foi muito semelhante aos anteriores, pois foi criada a classe **RedeNeuronal** que:

- Inicia um modelo do tipo Sequencial do Keras;
- Define a taxa de aprendizagem e o termo de momento, que neste caso foram definidos como 0.1 e 0.9 respetivamente, pois foram os que apresentaram melhores resultados na fase anterior;
- Adiciona as camadas do modelo:
  - Camada de entrada com 16 neurónios e função de ativação *relu*;
  - Camada de saída com 1 neurónio e função de ativação *sigmoid* pois trata-se de um problema de classificação binaria;
- Cria o optimizador da rede, que no caso também é do tipo SGD;
- Compila o modelo com um Loss de Erro Quadrático Médio e com o optimizador criado anteriormente;

Para alem disto a classe ainda possui o método **treinar()** que utiliza a função **fit()** do keras para propagar os dados pelo modelo a ajustar os pesos, assim como armazenar as métricas de avaliação do treino dos modelos. E a função prever que utiliza d função **predict()** do keras para obter as previsões do modelo treinado de um conjunto de dados.

Após criada a classe do modelo, foi possível criar um novo modelo, e treina-lo, tendo sido obtidos os seguintes resultados:

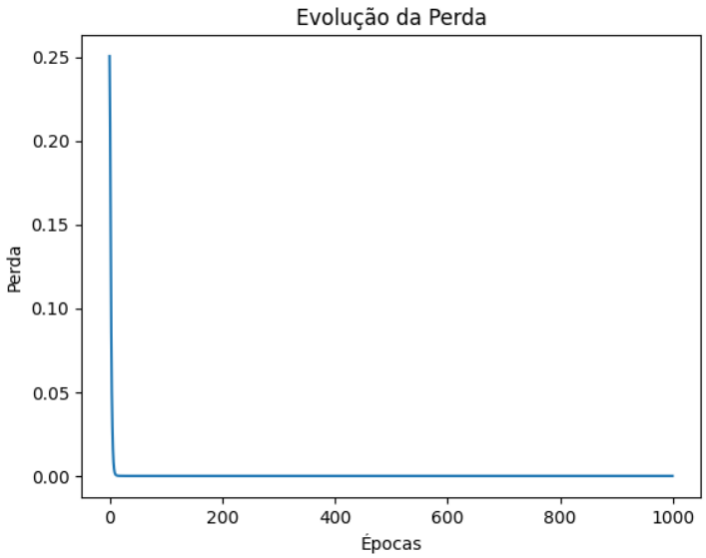

Figura 14 - Evolução do Loss da rede

Loss Final: **0.00172**

Ao observar, a evolução da perda e o seu valor final, podemos verificar que a rede obteve um desempenho praticamente perfeito, pelo que tentamos então efetuar as previsões da classificação dos padrões para ver se o mesmo se verifica.

Para tal, foi utilizado o método **prever()** da classe RedeNeuronal para prever a classe do padrão A e do padrão B, tendo sido obtidas as seguintes previsões:

```
1/1 [==============================] - 0s 79ms/step
Resultado para o padrão A: [[0.00235129]]
1/1 [==============================] - 0s 22ms/step
Resultado para B: [[0.99950546]]
```

Figura 15 - Previsões da rede

Sendo a função de ativação da camada de saída do tipo sigmoid, o output da rede são valores compreendidos entre 0 e 1 logo, ao analisarmos os resultados das previsões verificamos que para o padrão A foi previsto um valor muito perto de zero e para o padrão B um valor muito perto de um. O que indica que a rede tem uma excelente capacidade de classificar estes padrões.

1.3 Conclusões

Concluindo a primeira parte do projeto, pode-se afirmar que a resolução destes problemas, apesar de serem simples problemas de implementação de funções lógicas e de deteção de padrões, foi bastante importante para intender melhor o funcionamento das redes neuronais. Assim como os diferentes parâmetros, configurações e apresentação dos dados à rede influenciam o no desempenho. Já que para chegar a um resultado pretendido, possam existir vários caminhos sendo que existe sempre um grau de aleatoriedade. No entanto podemos sempre minimizar esta dependência da sorte ao perceber como podemos ajudar as redes a chegar ao resultado pretendido tendo em conta o contexto individual de cada problema.

Parte 2 – Aprendizagem por Reforço

2.1 Introdução teórica

1.1.1 Aprendizagem por Reforço

A Aprendizagem por Reforço é um paradigma de ML que lida com agentes interativos que tomam decisões com base na recompensa de comportamentos desejados ou na punição de comportamentos indesejados.

Para tal efeito, o objetivo principal passa por criar um agente que seja capaz de interpretar o ambiente em que se encontra, que tome ações e que aprenda com as mesmas. Já para aprender com estas ações, passa por ser essencial sermos nós a identificar quais os comportamentos que pretendemos que o agente realize, e os quais não queremos que sejam realizados. Logo devemos atribuir um certo valor a cada ação para que o agente consiga destingir o certo do errado.

Para explicar de uma melhor forma este sistema de aprendizagem, podemos observar a figura 16 que representa o funcionamento geral que se pretende implementar.

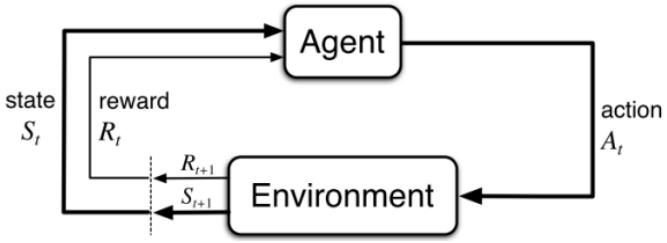

Figura 16 - Sistema de aprendizagem por reforço

1.1.1 Elementos Chave

- **Agente:** É o objeto que interage com o ambiente e que toma as decisões;
- **Ambiente:** Representa o contexto no qual o agente opera;
- **Ações:** Conjunto de decisões que um agente pode tomar;
- **Estado:** Descrição do posicionamento do agente no ambiente num determinado momento;
- **Recompensas:** Feedback que o agente recebe do ambiente após cada ação;
- **Política:** É a estratégia que o agente desenvolve sendo esta a ação que vai tomar para cada estado;

2.2 Implementação

Passando à implementação prática do projeto, esta foi realizada na linguagem *Python* e iniciou-se por definir o ambiente sobre o qual o agente vai aprender. Logo, como indicado, foi utilizado o modulo de simulação de ambiente disponibilizado que implementa:

- O conjunto de ações possíveis. Sendo estas mover-se para uma coordenada para a sua esquerda, uma coordenada para a sua direita, uma coordenada para cima ou uma coordenada para baixo;
- Indicação dos elementos presentes no ambiente:
  - O agente é representado pelo símbolo “@”;
  - O alvo é representado pelo símbolo “+”;
  - Um obstáculo é representado pelo símbolo “#”;
  - E os espaços vazios são representados por um espaço “ “;
- Três diferentes ambientes com diferentes graus de complexidade;

Figura 17 - Exemplo de um dos ambientes

- E a classe que define o ambiente. Classe esta que implementa diferentes métodos sendo os mais relevantes:
  - Obter a posição inicial do agente no ambiente;
  - Reiniciar a posição do agente no ambiente;
  - Realizar uma determinada ação para movimentar o agente no ambiente;
  - Observar o ambiente para obter a posição e elemento nessa na posição;
  - Mostrar a política para cada ação;

Após termos o ambiente preparado para receber o agente, foi então proposto efetuar três diferentes abordagens sobre a aprendizagem para comparação de preformasse, sendo estas:

- Q-Learning;
- Q-Learning com memória episódica;
- Q-Learning com seleção de ação com valores iniciais otimistas;

1.2.1 Q-Learning

O Q-Learning é um algoritmo de Aprendizagem por Reforço, projetado para permitir que um agente aprenda a tomar decisões sequenciais em um ambiente desconhecido. Este algoritmo é conhecido por sua simplicidade conceitual e eficácia em uma variedade de cenários. Pois este, numa fase inicial escolhe uma ação de forma aleatória e vai utilizando um método de tentativa e erro para ir atualizado os valores da sua tabela Q para cada par estado-ação. Estes valores representam assim a utilidade estimada de executar uma determinada ação em um determinado estado.

Assim, de forma a maximizar a sua preformasse e diminuir o seu erro, os valores Q da tabela são atualizados segundo a seguinte formula:

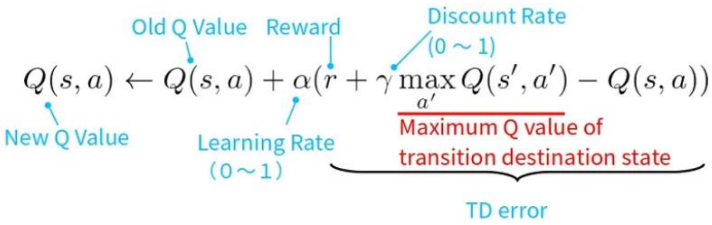

Figura 18 - Formula para atualizar o Valor Q da tabela

- Onde *Q(s,a)* é o valor *Q* para um estado *s* e ação *a;*
- Onde *r* é a recompensa recebida após executar a ação *a*;
- Onde α é a taxa de aprendizagem;
- Onde γ é o fator de desconto para uma recompensa futura;
- E onde *max'Q(s',a')* é o valor Q máximo para o próximo estado *s';*

Logo, para implementar este algoritmo, foi também necessário implementar classes que auxiliem e representem componentes como a memoria onde será armazenado o valor Q para cada par estado-ação. Um mecanismo de seleção de ações e uma classe para representar o agente.

Logo, para tal foram implementadas as seguintes classes:

MemoriaEsparsa

A classe **MemoriaEsparsa**, implementa e especializa a interface **MemoriaAprend**, e tem como principal objetivo criar uma memoria onde o agente pode armazenar o valor Q correspondente a cada par estado-ação em que se encontra, consultar o valor Q correspondente para um certo par estado-ação e atualizar estes mesmos valores em memoria.

Assim, esta classe, possui os métodos:

- **atualizar():** que recebe o par estado-ação e um valor q e guarda esta associação em memoria;
- **obterEstados():** que devolve os estados já que já têm um valor definido em memoria;
- **q():** que dado um par estado-ação retorna o valor Q em memória;

EGreedy

A classe EGreedy, implementa e especializa a interface **SelAccao** e é a responsável por efetuar as escolhas das ações que o agente deve tomar. Esta classe, utiliza uma estratégia de seleção **ε-greedy** onde o valor de ε determina um compromisso fixo entre exploração e aproveitamento. Sendo o principal objetivo da utilização desta estratégia, fazer com que o agente, á medida que obtém conhecimento, possa aproveitar esse conhecimento, evitado perdas por exploração desnecessária.

Logo em termos de implementação da classe **EGreedy**, esta possui métodos para:

- Obter para um determinado estado qual a ação onde o valor de Q é maior;
- Retornar uma ação aleatória;
- Selecionar uma ação, podendo esta ser aleatória ou com base no valor máximo de Q, pois utiliza a estratégia **ε-greedy** para de vez em quando ir tomando decisões aleatória e nem sempre com base nos valores de Q.

QLearning

Para melhor compreensão das seguintes classes, é possível também explicar já a implementação da classe **QLearning**, que implementa a interface **AprendRef** e que tem como principal função aprender sobre o ambiente. Para tal, esta classe possui apenas um método **aprender()** e que executa os seguintes passos:

- Recebe como entrada, um estado, uma ação, uma recompensa e o próximo estado;
- Calcula a próxima ação através ao chamar o método da classe EGreedy que escolhe uma ação com base num estado;
- Obtém o valor de Q do par estado-ação atual presente na memoria da classe **MemoriaEsparsa**;
- Obtém o valor de Q do par estado-ação futuro presente em memoria;
- Calcula o novo valor de Q para o par estado-ação atual pela fórmula explicada anteriormente na figura 18.
- Utiliza o método atualizar da memoria esparsa para atualizar o valor Q para o par estado-ação atual.

MecAprendRef

A classe **MecAprendRef** é responsável por juntar as diferentes componentes de aprendizagem para criar um mecanismo capaz de selecionar ações e aprender com as mesmas. Logo, esta classe, recebe como parâmetros de inicialização:

- A lista de ações possíveis;
- O valor de épsilon necessário á classe **EGreedy**;
- O valor alfa que representa o fator de desconto de uma recompensa futura, utilizado na classe **QLearing** para aprender e calcular o novo valor Q;
- O gama, que representa a taxa de aprendizagem utilizada também no método **aprender()** da classe **QLearning**;

A classe **MecAprendRef** possibilita então um agente de selecionar ações, aprender e obter a política que esta a ser utilizada para determinar uma dada ação. Para tal, implementa os seguintes métodos:

- **aprender():** Recebe como input um estado, uma ação, uma *reward* e o próximo estado. E chama o método aprender da classe **QLearning**;
- **selecionar_accao():** Recebe um estado como input e chama o método **seleccionar_accao()** do **EGreedy** para obter uma determinada ação;
- **obterPolitica():** Acede à memoria esparsa para obter os estados para os quais tem um valor Q definido, e obtém as respetivas ações correspondentes para cada estado ao chamar o método **accao_sofrega()** do **EGreedy**;

AgenteAprendRef

Por fim, chegamos à classe que representa o agente. Classe esta de estrema importância, pois é nela onde é são chamadas as diferentes componentes de forma a que seja possível executar o processo de aprendizagem e de gerar reforço para que possamos obter melhores resultados ao longo dos episódios.

Assim, para inicializar o agente, são necessários parâmetros como o ambiente que o agente deve explorar e o mecanismo de aprendizagem que deve utilizar.

Em relação aos métodos que a classe implementa, estes são os seguintes:

- **fim_episodio():** Verifica se o agente chegou ao alvo;
- **gerar_reforço():** Método onde são definidas as *rewards* atribuídas a cada ação, sendo 1 se o agente encontrar o Alvo, -0.5 se colidir com um obstáculo, e -0.1 se não colidir nem encontrar o alvo;
- **passo_episodio():** Método responsável pela ordem de execução em cada episodio:
    - Seleciona em primeiro lugar uma ação;
    - Executa a ação no ambiente;

- Observa o resultado da ação tomada, obtendo a sua nova posição e o elemento nessa posição;
- Obtém a *reward* respetiva á ação que tomou;
- Aprende com a ação tomada;
- Verifica se colidiu com um obstáculo. Caso aconteça volta à posição inicial;

- **executar():** Principal método responsável por executar os episódios de aprendizagem. Este método para cada episodio:
  - Reinicia o agente à posição inicial;
  - Observa o ambiente para obter a sua posição e elemento nessa posição;
  - Executa o método **passo_episodio();**
  - Armazena o número de passos nesse episodio;

1.2.2 Q-Learning com Memoria Episódica

Como segunda abordagem, foi sugerida a criação de uma memoria de experiência que consegue armazenar e recuperar experiências passadas, o que neste caso pode ajudar o agente a aprender mais rapidamente sobre o ambiente. Pois no caso do Q-Learning, o agente toma decisões com base no valor Q que tem em memoria, no entanto, esse valor é apenas no passo anterior e não tem acesso ao que aconteceu anteriormente. Logo, ao fornecermos mais informação ao agente, estamos a possibilita-lo de toma decisões mais fundamentadas.

Assim, para implementar esta memoria, foram criadas duas classes:

MemoriaExperiencia

Esta classe passa por criar uma memoria que armazena as experiências passadas do agente, logo, esta possui dois métodos:

- **atualizar():** Possibilita adicionar um valor á memoria. Valor esse que representa a experiência. Sendo esta experiência um estado, uma ação, uma *reward* e um próximo estado.
- **amostrar():** Retorna um conjunto aleatório de experiencias presentes em memoria.

QME

A classe QME estende as funcionalidades da classe **QLearning** ao adicionar a memoria de experiência e a capacidade de realizar simulações com base nestas experiências armazenadas em memoria, de forma a melhorar ainda mais o modelo.

Logo, para tal, esta classe implementa o método **aprender()** da classe **QLearning** mas para alem de atualizar o valor de Q do par estado-ação atual, este armazena a experiencia em memoria e utiliza o método **simular()** para um dado conjunto de amostras de experiencias em memoria para aprender e atualizar novamente os valores Q para essas experiencias em memoria.

1.2.3 Q-Learning com seleção de ação com valores iniciais otimistas

Por fim, foi sugerido que o algoritmo de seleção de ação fosse iniciado com valores inicias otimistas, o que acaba por ser variação do algoritmo clássico, mas que apresenta uma simples estratégia de iniciar os valores Q na tabela Q. Assim, são atribuídos para todos os pares estado-ação valores iniciais, geralmente sendo estes valores elevados. Essa escolha otimista tem o principal objetivo de incentivar o agente a explorar mais agressivamente no início do treino, procurando descobrir ações que possam levar a recompensas mais altas.

Para implementar esta abordagem, foi apenas necessário passa um valor inicial na inicialização do Mecanismo de Aprendizagem para que ao chamar o método **q()** da memoria episódica, caso o para estado-ação ainda não tenha valor em memoria, é atribuído o valor otimista passado como parâmetro como valor de Q.

2.3 Resultados obtidos

Após a implementação das diferentes propostas, foi criada uma classe de teste que cria uma instância do ambiente, cria uma instância do mecanismo de aprendizagem, cria um agente passando como parâmetros o ambiente e o mecanismo de aprendizagem do ambiente, executa o treino do agente num certo número de episódios passados por parâmetro, obtém a política resultante desse treino e retorna essa mesma política e o número de passo que o agente deu em cada episodio.

Assim, foi possível criar três diferentes testes, um para cada algoritmo, e mostrar a evolução do número de passos por episodio de cada agente em cada algoritmo, assim como as respetivas políticas.

Logo após realizar estes testes, foi possível observar os seguintes resultados para o ambiente 2 e para 100 épocas:

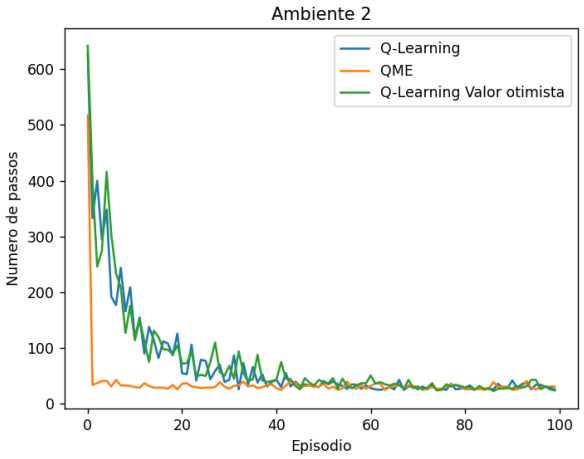

Figura 19 - Número de passos por episodio

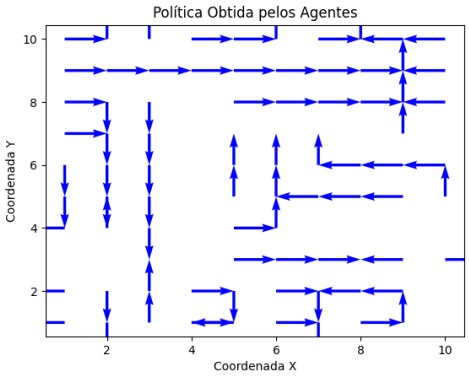

Figura 21 - Política do Q-Learning

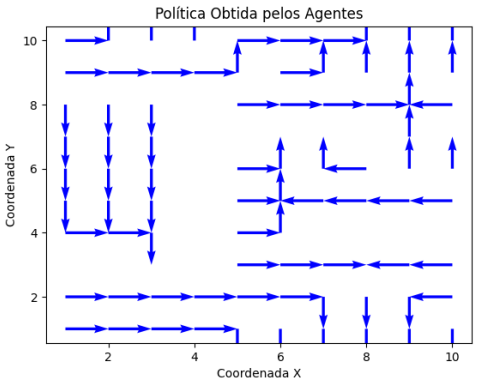

Figura 20 - Política do QME

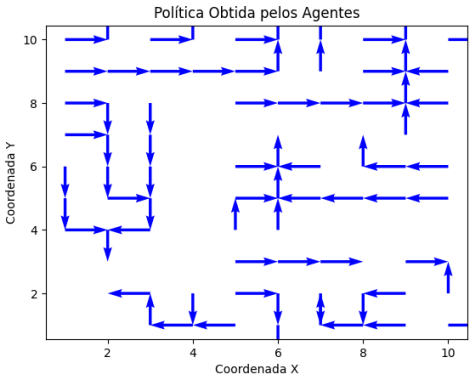

Figura 22 - Política do Q-Learning com valores otimistas

Analisando os resultados obtidos, podemos analisar os mesmos e verificar que claramente foi o algoritmo QME que obteve os melhores resultados. O que seria de esperar tendo em conta que o agente tem mais informações por onde pode extrair mais conhecimento e melhor os valores da tabela Q. No entanto, em relação aos valores iniciais otimistas, não se verificou uma melhoria em relação ao algoritmo original, o que se pode concluir que no problema em questão não faz grande diferença definir estes valores, ou que então o algoritmo pode estar mal implementado e existir algum erro.

2.4 Conclusões

Pode-se então concluir que o cenário proposto e as diferentes abordagem de resolução do problema, permitiram explorar e perceber melhor o funcionamento de um sistema de aprendizagem por reforço. Nomeadamente de como o algoritmo Q-Learning consegue criar uma memoria capaz de fornecer a capacidade de escolher quais a melhores ações a serem tomadas em cada circunstância.

Foi possível, identificar os diferentes aspetos que envolve este tipo de aprendizagem, como por exemplo a importância de definir correntemente a *reward* adequada para cada ação e como o agente deve escolher as ações a tomar. Tendo também constatado a estratégia de seleção de ação ε-greedy melhorou consideravelmente a rapidez com que o agente aprendeu. O que demostra que armazenar experiências passadas e ir aprendendo com as mesmas ao longo do tempo, trás benefícios ao agente.

Concluindo, a parte 2 do projeto permitiu claramente perceber mais sobre o tema aprendizagem por reforço, assim como perceber o que pode influenciar o desempenho desta abordagem.

Parte 3 – Raciocínio Automático e Arquitetura de Agentes Deliberativos

3.1 Introdução teórica

Raciocínio Automático

Raciocínio Automático refere-se à capacidade de um sistema computacional inferir, deduzir e tomar decisões com base em informações disponíveis. Através da utilização de diferentes algoritmos e técnicas que capacitam o sistema de processar dados, analisar contextos e derivar conclusões lógicas.

Representação do Conhecimento

A representação do conhecimento desempenha um papel crucial no Raciocínio Automático. Pois, os sistemas inteligentes precisam de ser capazes de expressar informações de maneira estruturada e manipular essas representações para inferir novos factos ou tomar decisões. Entre as técnicas mais utilizadas estão ontologias, redes semânticas e lógicas descritivas.

Algoritmos de Raciocínio

Diversos algoritmos são desenvolvidos com o intuito de resolver problemas de raciocínio lógico. Desde métodos baseados em regras, inferência estatística, procura em espaços de estados ou até abordagens mais avançadas, como a resolução de problemas por meio procura heurística, o Raciocínio Automático abrange uma gama variada de técnicas.

No entanto tendo em conta o objetivo proposto de implementar um agente deliberativo com planeamento, podemos analisar como funciona este método e o que pretendemos desenvolver através da arquitetura do mesmo.

Raciocínio Automático por Planeamento

O raciocínio Automático por Planeamento é uma abordagem focada em gerar uma certa sequencia de ações que levem a atingir um determinado objetivo. Ou seja, por exemplo um agente em vez de simplesmente reagir a estímulos ambientais, deve antecipar cenários futuros, identificar metas desejadas e desenvolver uma estratégia para alcançar essas mesmas metas.

Para desenvolver estas estratégias, o sistema pode utilizar diferentes algoritmos, como é o caso do algoritmo *Wavefront* (Frente-Onda) proposto para desenvolver esta parte do projeto.

Algoritmo Frente-Onda

O Algoritmo Wavefront baseia-se na propagação de "ondas" ou "gradientes" em um certo espaço discreto a partir de um ponto inicial em direção a pontos definidos como de destino. Para tal efeito, é criado um campo de valor representativo do espaço de procura, onde a cada célula do campo é atribuído um valor que representa a distância até o ponto inicial. Que ao realizar esta propagação de maneira iterativa, resulta um mapa de distâncias que codifica a acessibilidade de cada célula em relação ao ponto inicial.

Por exemplo na figura 23 podemos observar um ambiente, onde pretendemos que um determinado agente encontre o melhor caminho do ponto de partida (verde) até ao destino (vermelho). Onde as casas com o valor 0 (zero) representam espaços vazios e os valores a 1 representam obstáculos.

0	0	0	0	0	0	0	1
0	0	1	1	0	0	0	0
0	0	1	1	1	0	0	0
0	1	1	1	1	0	1	1
0	1	1	1	0	0	0	1
0	0	1	0	0	0	0	0

Figura 23 - Ambiente de exemplo

Ao programar um agente que faça implementação do algoritmo Wavefront, este pode então criar um campo de valores que represente este ambiente e que propague os valores pelo campo de forma iterativa até que encontre um mapa de distâncias que permitam chegar ao destino da melhor maneira possível. Na figura 24 podemos ver este mesmo ambiente e os valores atribuídos pelo agente para percorrer este mesmo caminho.

13	12	11	10	9	8	9	1
14	13	1	1	8	7	8	9
15	14	1	1	1	6	7	8
16	1	1	1	1	5	1	1
17	1	1	1	5	4	3	1
18	19	1	6	5	4	3	2

Figura 24 - Valores propagados pelo algoritmo Wavefront

Agentes Deliberativos

Um agente, é um sistema computacional capaz de perceber, decidir e tomar ações de forma autónoma para atingir determinados objetivos. Logo, para tal, este tem de conseguir adquirir informação de alguma forma, armazenar essa informação, tomar decisões tendo em conta a informação que possui e executar essas mesmas decisões. A forma como é implementado este processo, designa as capacidades cognitivas de um agente.

No entanto, este nível cognitivo de um agente pode estar organizado em diferentes tipos:

- Reativo (escala temporal curta);
- Adaptativo (escala temporal média);
- Deliberativo (escala temporal longa);

Sendo que um agente com nível cognitivo Reativo, possui necessidades imediatas e um tipo de processo executivo, enquanto que um agente adaptativo já possui objetivos de concretização e um processo mais tático.

Já um agente deliberativo, foca-se mais na otimização do seu desempenho e na estratégia utilizada para concretizar o objetivo proposto. Um agente deste tipo, baseia o seu comportamento com base em processos de planeamento suportados por representações internas do ambiente. Estas representações internas são responsáveis por representar os estados mentais do agente, definindo as suas crenças, desejos e intenções. Este modelo pode ser representado pela arquitetura BDI (*Belief-Desire-Intention*) como mostra a figura 25.

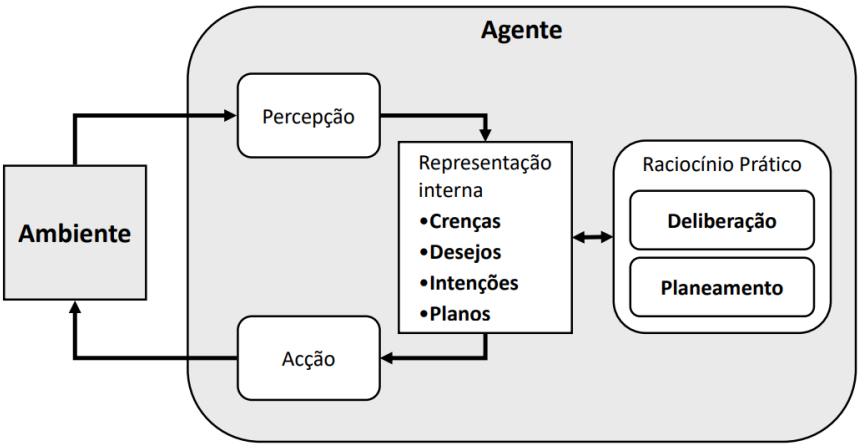

Figura 25 - Arquitetura Deliberativa com base no modelo BDI

Assim, podemos ter uma base do processo que fundamenta o Agente Deliberativo, que consegue implementar um algoritmo de Raciocínio Automático por Planeamento. Podendo então passar à parte da implementação do cenário proposto para esta fase.

3.2 Implementação

Como primeiro passo, foi necessário criar o ambiente sobre o agente deve aprende. No entanto, foi sugerida a utilização do modulo de simulação de ambiente pelo foi utilizada a mesma classe **Ambiente** da parte 2 do projeto e os mesmos 3 ambientes anteriores sendo que o ambiente mais utilizado nesta implementação foi o ambiente 2(figura 26) à semelhança da parte anterior.

Figura 26 - Ambiente 2

Para alem do ambiente, também foi implementada uma classe **Estado** para representar uma posição no ambiente, uma classe **Elemento** para representar os diferentes elementos presentes no ambiente (agente, alvo, obstáculo e espaço vazio) e uma classe **Accao** que representa uma ação a ser realizada.

Assim, em primeiro lugar, foi necessário criar a classe **FrenteOnda** responsável por implementar o algoritmo Wavefront para propagar os valores associados aos diferentes estados do modelo do mundo. Para melhor compreender a implementação desta classe podemos descreve-la da seguinte forma:

- Para ser chamada, necessita de dos valores “gama” e “max_value” onde o primeiro representa o fator de desconto e o segundo representa o valor máximo possível a ser associado a um estado;
- Implementa o método **adjacentes()** que tem o principal objetivo de dado um estado retorne os estados adjacentes a essa posição.
- Implementa o método **propagar_valor()** que utiliza o algoritmo Wavefront para propagar os valores pelo campo de valores do modelo do mundo. Este método:
    - Executa um *loop* para percorrer as diferentes posições do mundo;
    - Para cada posição itera sobre os estados adjacentes á mesma ao chamar o método **adjacentes();**
    - Atenua o valor de do campo de valores ao multiplicar o seu valor atual pelo fator de desconto e pela distância entre o a posição atual e o estado adjacente;
    - Se este novo valor for inferior ao anterior então substitui o valor anterior pelo novo valor;

Após ser implementada a classe **FrendeOnda**, responsável por efetuar a propagação e atualização dos valores do campo de valores, foi implementada a classe abstrata **ModeloMundo** utilizada para ser especializada pela classe **ModeloMundo2D.**

ModeloMundo2D

A classe **ModeloMundo2D** especializa a classe abstrata ModeloMundo2D e começa por definir as possíveis ações que no caso tomam os valores:

- **(1,0):** Se for pare se movimentar para este;
- **(-1,0):** Se for para se movimentar para oeste;
- **(0,1):** Se for para se movimentar para norte;
- **(0,-1):** Se for para se movimentar para sul;

Esta classe tem a principal função de representar a maneira de como o agente vai ver o mundo pelo que implementa métodos que forneçam informação sobre o mundo ao agente ou que o permitam simular ações no mesmo:

- **atualizar():** Recebe uma perceção do mundo e transforma essa mesma precessão em valores importantes como, os elementos em cada estado, as dimensões do mundo e lista dos estados para onde se pode movimentar;
- **obter_posicoes_alvo():** Retorna o estado onde se encontra o alvo;
- **distancia():** Premite calcular a distancia entre dois estados;
- **estado_valido():** Possibilita verificar se um estado é valido, ou seja, se é um espaço vazio;
- **simular_accao():** Dado um estado e uma ação, este método verifica se é possível executar essa ação ao chamar o método **estado_valido()**;

Ao ser implementada a classe **ModeloMundo2D**, foi possível implementar a classe **PlanFrenteOnda**, que elabora o planeamento de ações a serem executadas conforme os estados passados, os objetivos e o modelo do mundo indicado.

Assim, esta classe, implementa três diferentes métodos, com os objetivos de:

- Retornar os valores de cada estado do campo de valores;
- Calcular o valor de uma ação num determinado estado, ao obter o estado sucessor dada uma determinada ação e obter o valor desse estado sucessor no campo de valores;
- Produzir um plano de estratégia global ao utilizar o método **propagar_valor()** da classe **FrenteOnda** para propagar os valores. E a definir uma política com base nos valores que maximizam cada ação em cada estado.

Passando agora à implementação do agente, foi primeiramente criada a classe abstrata **AgenteDelib** que por sua vez foi especializada pela classe **AgenteFrenteOnda**, com o intuito de defenir o ambiente em que o agente aprende, o modelo desse mesmo mundo (**ModeloMundo2D**), o método de planeamento que vai utilizar (**PlanFrenteOnda**) e que consiga:

- Percecionar o mundo, ao obter o *array* que o representa;
- Atualizar as suas crenças, ao passar a perceção que teve do mundo para o método **atualizar()** da classe **ModeloMundo2d();**
- Identificar qual o objetivo que pretende atingir ao chamar o método **obter_posicoes_alvo()** da classe **ModeloMundo2d();**
- Construir um plano para alcançar o objetivo pretendido. Para tal, chama o método **planear()** da classe **PlanFrenteOnda** passando o objetivo pretendido.
- Visualizar o plano obtido com a política que o agente deve utilizar para chegar ao seu objetivo.

Para esta última funcionalidade de visualizar o plano obtido com a política que o agente deve utilizar para chegar ao seu objetivo, foi construída uma classe auxiliar que recebe a política e o campo de valores que resultou do processo de planeamento e utiliza a biblioteca **matplotlib** para mostrar o ambiente e o valor calculado correspondente a cada posição, sendo assim possível observar tanto a diferença de valores nas diferentes posições como as ações correspondentes a cada posição.

Por fim, para executar um teste de todo este sistema, foi possível criar um ficheiro de teste onde é indicado o ambiente para o qual pretendemos elaborar o plano e criar e executar o agente.

Resultados obtidos:

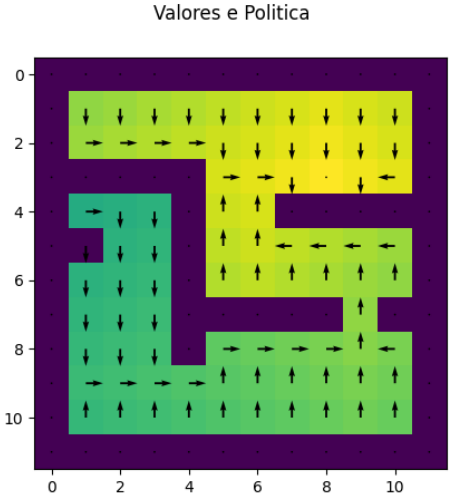

Figura 27 - Política planeada pelo agente

O resultado foi bastante satisfatório pois conseguimos observar a política obtida pelo agente e como os valores do campo de valor foram calculados. Em tons mais verdes, conseguimos ver os valores mais distantes do objetivo a conforme a aproximação do objetivo estes tons ficam cada vez mais amarelos. Já em relação às ações podemos verificar também que estas estão orientadas de acordo com o caminho correto para atingir o objetivo.

3.3 Conclusões

Numa fase inicial da terceira parte do projeto, foi explorado o conceito de Raciocínio Automático e a arquitetura de Agentes Deliberativos, focando especialmente no Raciocínio Automático por Planeamento. Nesta fase, foi também destacado o Algoritmo Wavefront, uma técnica eficaz de propagação de "ondas" em um espaço discreto, utilizado para efetuar o planeamento em diferentes ambientes. Este algoritmo foi implementado no contexto do projeto, demonstrando sua capacidade de gerar estratégias para um agente alcançar objetivos em um ambiente com obstáculos.

Após um estudo teórico, passou-se á implementação prática, que envolveu a criação de classes para representarem desde do ambiente até a implementação do Algoritmo Wavefront e da arquitetura do Agente Deliberativo. Aqui, podemos destacar a classe **AgenteFrenteOnda**, que para alem de interligar todas as componentes do sistema utiliza o algoritmo de planeamento para perceber, decidir e agir de maneira autônoma, na procura de atingir os objetivos previamente definidos.

Em relação aos resultados obtidos foram satisfatórios, evidenciando a eficácia da abordagem. A visualização da política do agente, com tons mais verdes distantes do objetivo e transição para amarelos conforme a aproximação, bem como a orientação correta das ações, demonstrou a robustez do sistema implementado.

Assim, podemos concluir que a combinação de Raciocínio Automático por Planeamento e Agentes Deliberativos oferece uma abordagem sólida para a resolução de problemas complexos, possibilitando que agentes autônomos tomem decisões estratégicas diferentes em ambientes.

4 Conclusão Final

Ao longo deste projeto, foi explorado e aprofundo o conhecimento em três áreas cruciais da inteligência artificial: Redes Neuronais Artificiais, Aprendizagem por Reforço e Raciocínio Automático com Arquiteturas de Agentes Deliberativos. Onde cada parte do projeto proporcionou insights valiosos e experiências práticas, permitindo compreender os desafios e complexidades inerentes a cada abordagem.

Na primeira parte, foram então implementadas redes neurais para resolver problemas de implementação de funções lógicas e deteção de padrões. Onde foi identificado que o desempenho da rede é fortemente influenciado por diversos parâmetros, configurações e pela apresentação adequada dos dados. A compreensão desses elementos não apenas aprimorou a resolução dos problemas propostos, mas também permitiu minimizar a aleatoriedade, destacando a importância de ajustar a abordagem a contextos individuais.

Já na segunda parte, foram explorados os conceitos relacionados com a aprendizagem por reforço, tendo como principal foco o algoritmo Q-Learning. A análise do cenário proposto e a implementação prática revelaram a importância da definição apropriada de recompensas, estratégias de seleção de ações e a eficácia da técnica ε-greedy. Os resultados evidenciaram que a aprendizagem a partir de experiências passadas, aliada a uma estratégia de seleção inteligente, acelera significativamente o processo de aprendizagem do agente.

Na terceira parte, foi abordado o Raciocínio Automático por Planeamento e a arquitetura de Agentes Deliberativos, destacando o Algoritmo Wavefront como uma ferramenta eficaz para elaborar planos em ambientes com obstáculos. A implementação prática, incorporou o algoritmo e a arquitetura do Agente Deliberativo, proporcionou resultados satisfatórios. A combinação desses elementos oferece uma abordagem sólida para a resolução de problemas complexos, capacitando agentes autônomos a tomar decisões estratégicas em diversos ambientes.

Em síntese, este projeto proporcionou uma visão abrangente das diversas áreas da inteligência artificial. Ao aprofundar o conhecimento sobre redes neurais, aprendizagem por reforço e raciocínio automático, foi adquirido não apenas conhecimento teórico, mas também habilidades práticas essenciais para enfrentar desafios na implementação de soluções inteligentes. A combinação destas abordagens abre portas para aplicações inovadoras, demonstrando o potencial da inteligência artificial na resolução de problemas complexos e na tomada de decisões autônomas.

5 Referencias

[1] Documentação de apoio à unidade curricular Luís Morgado, ISEL-DEETC.

[2] Ateliware. (2022). Redes Neurais Artificiais. Disponível em: https://ateliware.com/blog/redes-neurais-artificiais

[3] Universidade Federal do Paraná. (s.d.). Gradiente Descendente. Disponível em: http://cursos.leg.ufpr.br/ML4all/apoio/Gradiente.html

[4] Imobilis - UFOP. (s.d.). Redes Neurais: Funções de Ativação. Disponível em: https://www2.decom.ufop.br/imobilis/redes-neurais-funcoes-de-ativacao/

[5] Data Science EU. (s.d.). Como Funciona o Algoritmo de Retropropagação? Disponível em: https://datascience.eu/pt/inteligencia-artificial/como-funciona-o-algoritmo-de-retropropagacao/

[6] Udacity. (s.d.). The Q in Q-learning: A Comprehensive Guide to This Powerful Reinforcement Learning Algorithm. Disponível em: https://itsudit.medium.com/the-q-in-q-learning-a-comprehensive-guide-to-this-powerful-reinforcement-learning-algorithm-896cbbedcd33

[7] Ashesi Robotics. (2012, março 29). Task 4: Path Planning. Disponível em: https://ashesirobotics.wordpress.com/2012/03/29/task4-path-planning/