Gonçalo António Martins Biscaia

Licenciado em Ciências de Engenharia Eletrotécnica e de Computadores

"Deep Reinforcement Learning" na Otimização de Políticas de Encaminhamento na Manufatura

Dissertação para obtenção do Grau de Mestre em

Engenharia Eletrotécnica e de Computadores

Orientador: José António Barata de Oliveira, Professor Doutor, Universidade Nova de Lisboa

Co-orientador: Ricardo Silva Peres, Doutor, Universidade Nova de Lisboa

Júri

Presidente: Prof. Doutor João Carlos da Palma Goes
Arguentes: Prof. Doutor José Manuel Matos Ribeiro da Fonseca
Vogais: Prof. Doutor José António Barata de Oliveira

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2019

"Deep Reinforcement Learning" na Otimização de Políticas de Encaminhamento na Manufatura

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Para Sónia e Família.

Agradecimentos

Nesta secção quero expressar os meus genuínos agradecimentos a todos os que contribuíram para a realização desta dissertação.

Primeiramente, quero agradecer ao Professor Doutor José Barata de Oliveira por me aceitar como seu orientando, pela sua orientação e disponibilidade e por possibilitar a realização desta dissertação na área de robótica e manufatura.

Em seguida, quero agradecer ao meu co-orientador, Mestre Ricardo Silva Peres, pela oportunidade, por todo o conhecimento transmitido e por toda a disponibilidade, apoio e dedicação. Tenho também a agradecer ao Doutor André Rocha pela disponibilidade que demonstrou na escolha do tema, bem como todo o apoio prestado.

Não posso deixar de agradecer à Universidade Nova de Lisboa, à Faculdade de Ciências e Tecnologia e a todos os docentes que acompanharam o meu percurso académico, pois foi devido a eles que me foi possível crescer tanto a nível académico, como a nível pessoal.

Um agradecimento muito especial à minha família, especialmente aos meus pais e irmã, por todo o apoio, carinho, dedicação e paciência que têm demonstrado ao longo deste percurso, porque se não fossem eles não seria possível frequentar este curso académico e realizar esta dissertação.

Um agradecimento especialíssimo a uma das pessoas mais importantes da minha vida, a minha namorada Sónia, por todo o apoio, paciência, dedicação e ajuda durante o período de desenvolvimento desta dissertação e não só.

Não posso deixar de demonstrar a minha gratidão a dois grandes amigos, Bruno Ameixieiro e Guilherme Rolo, que me acompanharam durante o meu percurso académico, por estarem sempre disponíveis em todos os momentos e por todo o apoio que deram ao longo destes anos.

Um especial obrigado ao meu amigo Válber Cavalcanti por toda a paciência, ajuda, conhecimento transmitido e dúvidas esclarecidas.

Um muito obrigado ao Diogo Ferreira, ao Bruno Geraldo e a todos aqueles que me acompanharam, ajudaram e apoiaram no meu percurso académico.

A todos vós,

O meu mais sincero,

Muito Obrigado!

Resumo

O mercado encontra-se cada vez mais heterogéneo e individualizado. Com isto, existe a necessidade de desenvolver sistemas de produção que consigam produzir de uma forma otimizada e eficiente, produtos cada vez mais personalizados. Porém, estes sistemas estão a tornar-se cada vez mais complexos e consequentemente, os problemas inerentes a estes.

O escalonamento da produção, de uma forma sintética, procura o fabrico de um produto, através da execução de diferentes trabalhos sobre matéria-prima, dentro de um prazo estipulado, visando o uso eficiente de recursos. Um problema intrínseco ao escalonamento da produção é a otimização de políticas de encaminhamento. A otimização de políticas de encaminhamento consiste em escolher o caminho de produção mais rápido, consoante vários fatores, como a posição de cada produto, ou quais os trabalhos ocupados num dado instante.

São inúmeros os métodos utilizados para tentar otimizar políticas de encaminhamento numa linha de produção, sendo que dois dos utilizados são heurísticas e inteligência artificial, nomeadamente *Reinforcement Learning (RL)*. Nesta dissertação é proposta a utilização de um algoritmo de *Deep Reinforcement Learning (DRL)*, sendo este o algoritmo *Double Deep Q-Network (Double-DQN)*, adaptado a um ambiente multi-agente. É feita a comparação de diferentes métricas, no que toca à otimização de políticas de encaminhamento, entre este algoritmo e os anteriormente referidos, em duas simulações de ambientes de produção distintos, com caraterísticas específicas, como duas ou mais zonas de trabalho iguais e diferentes escolhas de caminhos para a produção do produto. Dito isto, o objetivo é demonstrar a genericidade deste algoritmo a diferentes ambientes de produção, bem como comprovar a facilidade de o implementar em novos ambientes. Para além disto, é também provado que este é capaz de encontrar o caminho mais rápido de produção, nas simulações propostas, utilizando os algoritmos supracitados como termo de comparação.

Palavras-chave: Escalonamento na Produção, Políticas de Otimização de Encaminhamento, Inteligência Artificial, Produção, Auto-Encaminhamento

Abstract

The market is becoming more heterogeneous and individualized. As such, there is the need to develop efficient and optimized production systems able to produce, in large scale, personalized products. However, this systems are becoming more and more complex and consequently, the problems inherent to these.

Production scheduling, in a summarized way, consists in the manufacture of a product, through the execution of different jobs on raw materials, within a time period, aiming at the efficient use of resources. A problem intrinsic to the production scheduling, is the optimization of routing politics. The optimization of routing politics, consists in choosing the fastest production path, depending on various factors, like the position of each product, or which jobs are occupied in a given instant.

There are numerous methods used to try to optimized routing politics in a production line, two of which are heuristics and artificial intelligence, such as reinforcement learning.

In this master's thesis, it is proposed the use of a DRL algorithm, named *Double-DQN*, adapted to a multi-agent system. It will be made the comparison of different metrics, regarding the optimization of routing politics, between this method and the above, in two different simulations with unique features, such as two or more jobs of the same type, as well as different production paths that lead to the same goal. The main objective of this master's thesis is to demonstrate the genericity of this algorithm to different production environments, as well as to prove the facility to implement it in new environments. In addition to this, it is proved that this algorithm is capable of finding the fastest production path, in the proposed simulations, using the above algorithms as a comparation term.

Keywords: Production Scheduling, Optimization of Routing Politics, Artificial Intelligence, Production, Self-Pathing

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Glossário

batch size — Termo utilizado em *machine learning*, que se refere ao número de exemplos de treino utilizados numa iteração .

buffer — Local de uma linha de produção que tem o objetivo de armazenar a matéria-prima à espera de ser manipulada .

cloud — Capacidade de guardar e aceder a informação e programas através da *internet*, em vez do computador pessoal .

CNN — Rede Neuronal Convolucional, ou Convolutional Neural Network em inglês, consiste numa classe de *Deep Neural Networks* utilizadas em processamento de imagem, devido ao facto de possuírem camadas convolucionais capazes de analisar imagens ao pormenor .

digital twin — Representação virtual de algo físico, como por exemplo, o *digital twin* de uma transportadora física, será a sua representação virtual .

operação — Definida por [27], uma operação é uma ação cujo objetivo é manipular matéria-prima, como por exemplo, soldar ou prensar. Estas operações são realizadas por diferentes tipos de máquinas, denominadas de recursos, que se encontram em estações específicas numa linha de produção. Podem também ser denominadas de *skills* em inglês .

pesos — Conjunto de valores alterados pela CNN, ao longo do seu treino, que permitem realizar uma estimativa cada vez melhor do valor a prever .

URL — *Url*, ou *Uniform Resource Locator*, é o endereço que permite aceder a qualquer *webpage*, servidor local, entre outros, através de um *browser* .

Siglas

A3C	*Advantage Actor Critic*.
ACPS	*Anthropocentric Cyber-Physical System*.
ADI	*Autodidactic Iteration*.
ANN	*Artificial Neural Network*.
API	*Approximate Policy Iteration*.
CIM	*Computer Integrated Manufacturing*.
CNN	*Convolutional Neural Network*.
CPPS	*Cyber-Physical Production System*.
CPS	*Cyber-Physical System*.
DANN	*Deep Artificial Neural Network*.
Double-DQN	*Double Deep Q-Network*.
DQN	*Deep Q-Network*.
DRL	*Deep Reinforcement Learning*.
FJSP	*Flexible Job-Shop Problem*.
FPS	*Frames Per Second*.

HTML	*Hypertext Markup Language*.
I4.0	Indústria 4.0.
IA	Inteligência Artificial.
IoT	*Internet of Things*.
IP	*Intelligent Product*.
MAE	*Mean Absolute Error*.
MDP	*Markov Decision Process*.
ML	*Machine Learning*.
MSE	*Mean Squared Error*.
NSF	*National Science Foundation*.
P-FJSP	*Partially-Flexible Job-Shop Problem*.
PLC	*Programmable Logic Controller*.
RAMI 4.0	*Reference Architecture Model Industry 4.0*.
RFID	*Radio Frequency Identification*.
RL	*Reinforcement Learning*.
SELSP	*Stochastic Economic Lot Scheduling Problem*.
SF	Sistema Físico.
SMA	Sistema Multi-Agente.

SP *Service Product*.

T-FJSP *Total Flexible Job-Shop Problem*.

URL *Uniform Resource Locator*.

Capítulo 1

Introdução

1.1 Problema

Estamos perante o aparecimento de um mercado heterogéneo e individualizado. Como tal é necessário alterar a indústria e desenvolver sistemas de produção que consigam criar produtos cada vez mais personalizados, de uma forma otimizada e eficiente, [7]. Assim é necessário revolucionar a manufatura e criar um novo marco industrial, a Indústria 4.0 (I4.0). Esta pretende agregar na mesma rede sistemas de produção atuais, com uma vasta combinação de sensores e atuadores, de modo a criar modelos virtuais de entidades físicas, permitindo a concessão de sistemas de produção autónomos, dinâmicos e inteligentes.

O aparecimento de um mercado cada vez mais heterogéneo, bem como a I4.0, veio aumentar a complexidade dos sistemas e consequentemente de alguns problemas já existentes, nomeadamente o escalonamento da produção.

O escalonamento da produção, de uma forma sucinta, consiste no fabrico de um produto da forma mais rápida e eficiente possível, através da execução de diferentes operações[1] sobre matéria-prima, dentro de um prazo estipulado. Para responder às novas exigências do mercado, foram criados sistemas de produção que permitem o desenvolvimento de diferentes tipos de produto, bem como o aparecimento de vários caminhos e diversos recursos que realizam o mesmo tipo de operações. Estas alterações vieram aumentar a complexidade destes sistemas e consequentemente a complexidade de problemas existentes, como o escalonamento da produção. Um caso específico dos diversos problemas de escalonamento da produção, consiste na otimização de políticas de encaminhamento, tendo em conta diversos fatores como, a posição dos restantes produtos, a

[1] Definida por [27], uma operação é uma ação cujo objetivo é manipular matéria-prima, como por exemplo, soldar, prensar, entre outras. Estas operações são realizadas por diferentes tipos de máquinas, denominadas de recursos, que se encontram em estações específicas numa linha de produção. Podem também ser denominadas de *skills* em inglês.

disponibilidade dos diferentes recursos, e consequentemente das diferentes operações.

São inúmeras as soluções utilizadas para atenuar estes problemas. Parte delas utilizam uma arquitetura multi-agente que permite atribuir inteligência coletiva, bem como a capacidade de comunicação entre os diferentes agentes de um sistema. Apesar da aplicação de uma arquitetura multi-agente ser benéfica numa linha de produção presente na I4.0 e na resolução deste tipo de problemas, [5], é também possível acrescentar inteligência individual a cada agente. A presente dissertação propõe a utilização de Inteligência Artificial (IA) num Sistema Multi-Agente (SMA), através de um algoritmo de DRL, de modo a conferir inteligência individual a agentes, comparando com outras alternativas cujo propósito é o mesmo.

1.2 Cenário Motivador

Imaginemos uma linha de produção, denominada de **linha de produção A**, presente na figura 1.1, onde são produzidos relógios do **tipo A**. A matéria-prima, necessária para a produção deste tipo de relógios, é introduzida no início da linha e necessita de passar pelos recursos **TA**, **TB** e **TC**, de modo a produzir um relógio de **tipo A**. Esta linha apresenta as seguintes caraterísticas:

- um único caminho de produção,
- três tipos de recursos diferentes, denominados de **TA**, **TB** e **TC**,
- cada recurso só consegue realizar uma operação,
- uma única estação para cada recurso.

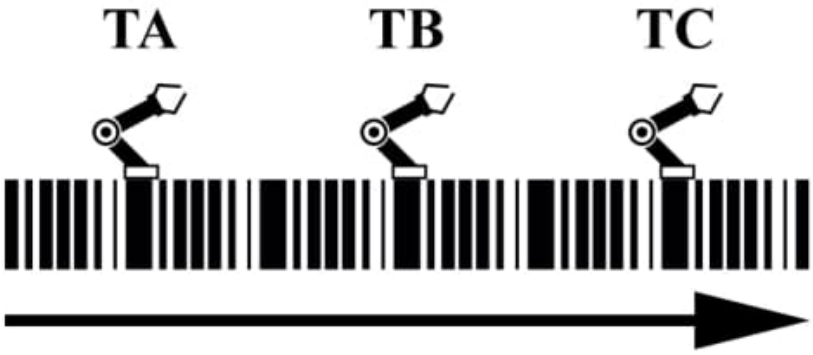


Figura 1.1: Representação da linha de produção A.

Esta linha de produção apresenta uma desvantagem evidente, sendo esta, o facto de ser em série e como tal a matéria-prima introduzida só é manipulada quando a primeira estação, neste caso a que contém o recurso **TA**, se encontra livre. Isto faz com que o

ritmo de produção seja reduzido e que seja necessário a existência de um *buffer*[2], de modo a acumular a matéria-prima que se encontra para ser manipulada pelas diferentes máquinas. Com isto, a produção não é eficiente e o tempo de produção de cada produto, neste caso relógio, é superior.

Imaginemos agora que seria necessário produzir dois tipos de relógio nesta linha, o **tipo A**, referido anteriormente, e o **tipo B**. No entanto, para produzir um relógio do **tipo B** é necessário acrescentar um novo recurso, sendo este o recurso **TD**, fazendo com que a matéria-prima tenha de passar pelos recursos **TA**, **TB** e **TD**, de modo a que o relógio do **tipo B** seja produzido. Sendo assim, o problema anteriormente referido é agravado, visto que, vão passar a existir quatro estações, cada uma com um recurso, bem como um novo tipo de relógios.

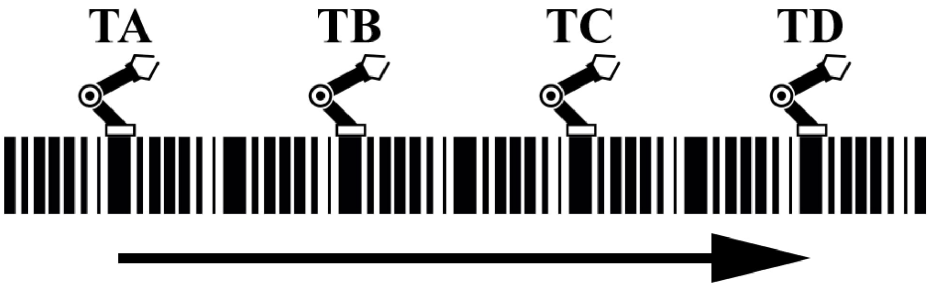


Figura 1.2: Representação da linha de produção A configurada para produzir os dois tipos de relógios.

Vamos agora imaginar uma nova linha de produção, **linha de produção B**, presente na figura 1.3, com as seguintes caraterísticas:

- quatro caminhos de produção, dois para cada tipo de relógio,
- quatro tipos de recursos diferentes, denominados de **TA**, **TB**, **TC**, **TD**,
- quatro estações para os recursos **TA** e **TB**, e duas estações para os recursos **TC** e **TD**,
- cada recurso só consegue realizar uma operação.

[2]Neste caso, o *buffer* é o local de uma linha de produção que tem o objetivo de armazenar a matéria-prima à espera de ser manipulada.

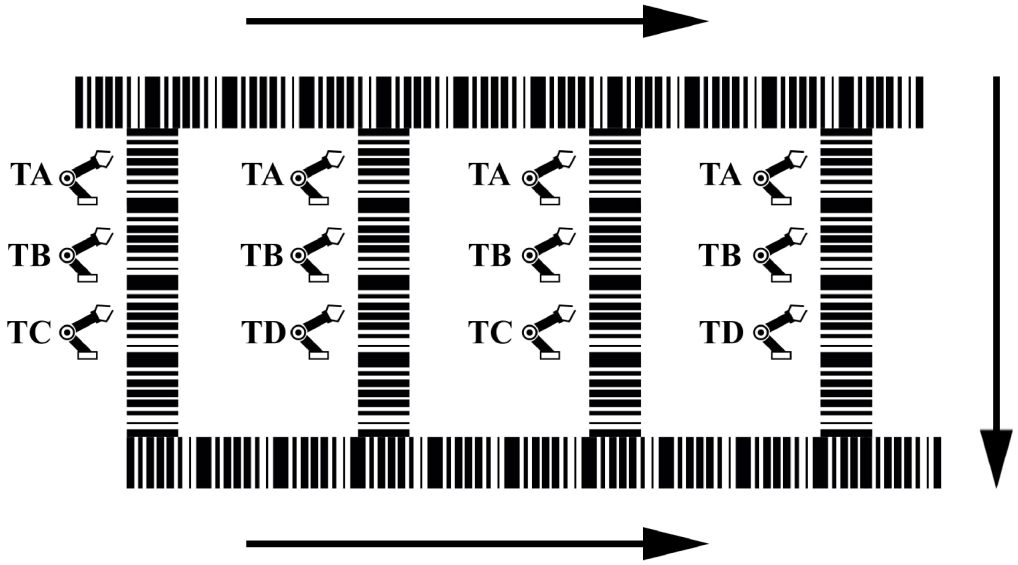


Figura 1.3: Representação da linha de produção B.

É possível observar que existem alterações significativas entre a **linha de produção A** e a **linha de produção B**, sendo estas, o facto de existirem dois caminhos possíveis para cada relógio ser produzido e terem sido acrescentados novos recursos. Com isto, é possível efetuar uma distribuição de matéria-prima de uma forma mais eficiente e fabricar vários tipos de relógios numa só linha de produção, indo de encontro com as exigências do mercado.

Apesar da **linha de produção B** ser mais vantajosa em termos de distribuição de matéria-prima, esta é mais complexa no que toca ao escalonamento da produção, nomeadamente em relação às políticas de encaminhamento necessárias para distribuir, de uma forma eficiente, a matéria-prima pelos diferentes caminhos.

1.3 Questões e Hipóteses

Tendo em conta o problema anteriormente referido, bem como o cenário acima apresentado, algumas questões podem ser colocadas:

- Como conseguirá o produto escolher o caminho mais rápido, de modo a que seja feita uma produção eficiente?
- Que métodos podem ser utilizados para garantir inteligência individual a elementos de uma linha de produção?
- É possível garantir inteligência global através de inteligência individual?

Com base nas perguntas supracitadas, é proposta como hipótese a utilização de um algoritmo de DRL, mais especificamente o algoritmo *Double-DQN*, onde, através de dois

ambientes de simulação distintos, sendo que um deles reflete um cenário real, será provado que o algoritmo proposto é capaz de garantir inteligência individual a elementos de uma linha de produção, de uma forma genérica, permitindo a sua reutilização, e fácil implementação, em ambientes de manufatura diferentes.

São também propostas outras duas hipóteses, que também visam a utilização do algoritmo acima referido, onde será provado e comparado, através de cenários de teste, métricas e outros dois algoritmos, que o algoritmo proposto possui a capacidade de escolher o caminho de produção mais rápido, tendo em conta a posição dos restantes produtos, visando uma produção eficiente e no menor tempo possível, bem como, através de transferência de conhecimento, permitir com que a inteligência individual dada aos diferentes agentes, lhes atribua também inteligência coletiva.

1.4 Estrutura do Documento

A presente dissertação apresenta diferentes capítulos que servem como fio condutor para a explicação do trabalho realizado.

O documento encontra-se estruturado em seis capítulos, sendo estes:

- Introdução,
- Estado da Arte,
- Desenho do Sistema,
- Implementação,
- Resultados,
- Conclusões e Trabalho Futuro.

No capítulo **Estado da Arte**, é realizada uma revisão da literatura acerca de temas chave relacionados com o trabalho presente nesta dissertação. Primeiramente, é apresentada uma visão global da I4.0, começando por uma contextualização histórica da evolução industrial, onde é referido o porquê do aparecimento desta nova era industrial, seguida de uma explicação da sua estrutura, da enumeração de algumas vantagens e desvantagens e da importância do papel humano nesta nova era industrial. Seguidamente, é abordado o tema dos CPPS, começando com a explicação do que é um CPPS, os benefícios que estes podem trazer para um sistema de produção e algumas arquiteturas propostas na literatura, para estes sistemas. Posteriormente, são abordados os SMA, começando por apresentar algumas definições de agente, seguidas por certas propriedades destes sistemas e pela apresentação de métodos para atribuir certas caraterísticas a um agente, realizando assim uma ponte para o tema seguinte. Em seguida, é abordada a definição de RL, as suas caraterísticas, problemas e respetivas soluções e o benefício da utilização de uma rede neuronal juntamente com RL. Por fim, é realizada uma explicação do que

consiste o escalonamento da produção, sendo também referenciados alguns exemplos, presentes na literatura, que utilizam heurísticas e RL ou DRL, na resolução de problemas de escalonamento.

No capítulo **Desenho do Sistema**, primeiramente, é explicado o conceito de RL, onde é abordada a definição de *Markov Decision Process* (MDP) e referidas caraterísticas chave que um ambiente, ou simulação, necessita de apresentar, de modo a que seja possível implementar um algoritmo de RL. Ainda no contexto de RL, é introduzido o conceito de DRL, assim como são abordados os algoritmos *Q-learning*, e *Double-DQN*, frisando as suas principais diferenças, vantagens e desvantagens. Por fim, são apresentados os conceitos genéricos do algoritmo heurística *A**. Em seguida, é abordada uma definição genérica de *job-shop* e os seus principais componentes, fazendo o enquadramento com o conceito de SMA e os seus agentes. Por fim, é proposto um desenho genérico para uma solução, que visa a relação entre o ambiente físico e a entidade virtual, abordando o conceito de CPPS, permitindo a implementação de algoritmos inteligentes virtuais, num sistema de produção real.

O capítulo **Implementação**, consiste na explicação dos ambientes de simulação desenvolvidos, referindo as suas estruturas, caraterísticas e objetivos, bem como a apresentação dos parâmetros utilizados nos algoritmos propostos e a sua implementação mais detalhada.

No capítulo **Resultados**, são explicados os diferentes cenários de teste utilizados para comparar os diversos algoritmos, através de métricas, são apresentados e analisados os resultados obtidos nestes cenários e é demonstrada a genericidade do algoritmo *Double-DQN*, demonstrando que este é capaz de aprender as políticas ótimas para simulações distintas, com poucas alterações na sua implementação.

Por fim, no capítulo **Conclusões e Trabalho Futuro**, é averiguado se foi possível responder às perguntas propostas, através da hipótese apresentada, bem como formular uma conclusão acerca do trabalho desenvolvido e apresentar alternativas para trabalho futuro, desde otimizações ao algoritmo proposto, até novas implementações de algoritmos de DRL num ambiente de produção.

1.5 Interpretação Geral do Trabalho Realizado

Na presente dissertação são apresentadas duas simulações baseadas num SMA. O objetivo destas simulações é representar linhas de produção e os seus componentes principais, utilizando agentes, bem como servirem de ambiente para treinar e testar os diferentes algoritmos apresentados.

A primeira simulação tem como objetivo comparar o desempenho de três algoritmos, através de cenários de teste e métricas, relativamente ao encaminhamento de produtos, de modo a serem produzidos o mais rapidamente possível.

Por sua vez, a segunda simulação é uma representação digital, de um caso real, cujo

objetivo é demonstrar que o algoritmo proposto nesta dissertação é genérico e fácil de implementar em diferentes ambientes de produção, bem como comprovar a importância de um *digital twin* nesta implementação. Através da utilização de um *digital twin* é possível treinar o algoritmo em *offline*, onde posteriormente, tanto o *digital twin* como o algoritmo, são utilizados para controlar a linha de produção real. Desta forma, a linha de produção real não é afetada durante o treino do algoritmo, não sendo necessário parar a produção.

Nestas simulações existem locais específicos onde os agentes podem tomar decisões, consoante o caminho que querem tomar, de modo a serem produzidos. Para além dos produtos, são também representados outros componentes chave de uma linha de produção, como por exemplo as transportadoras.

Após o desenvolvimento das simulações, foram criados três algoritmos, sendo estes o algoritmo *Double-DQN*, o algoritmo *Q-learning* e o algoritmo heurístico *A**. Os algoritmos supracitados foram testados na primeira simulação, sendo que os algoritmos *Double-DQN* e *Q-learning* necessitaram de ser submetidos a um treino, antes de serem testados. Por sua vez, na segunda simulação só foi treinado e testado o algoritmo *Double-DQN*.

Ao longo do desenvolvimento deste projeto, poderá ser criado um artigo científico com base na implementação e resultados obtidos.

Capítulo 2

Estado da Arte

Atualmente estamos perante uma evolução tecnológica acentuada. Com a evolução dos componentes físicos, nomeadamente memórias, processadores, componentes de processamento gráfico, entre muitos outros, foi possível criar e otimizar algoritmos computacionais, de tal forma que hoje em dia é possível atribuir inteligência, capacidade de tomada de decisão e dinamismo a sistemas que outrora eram estáticos e não possuíam qualquer tipo de inteligência. A respeito da manufatura industrial, esta foi também afetada por estes atributos, permitindo a criação de sistemas e fábricas com as características supracitadas.

Na manufatura industrial estas alterações são vantajosas, pois permitem alterar o modo de funcionamento das indústrias e fábricas, no entanto vêm tornar os sistemas e alguns problemas já existentes na manufatura mais complexos. Sendo assim, o foco desta revisão de literatura recai sobre alguns problemas desta temática, mais especificamente sobre os problemas de escalonamento de produção e otimização de políticas de encaminhamento.

2.1 Indústria 4.0

Ao longo dos anos houve uma acentuada evolução no que toca à indústria e manufatura. Esta evolução foi marcada por diferentes marcos, denominados de revoluções industriais. As revoluções industriais são caraterizadas pela descoberta de novos métodos que possibilitam a otimização e evolução do processo de manufatura e, consequentemente, da indústria.

Desde o século XVIII ao século XX existiram três revoluções industriais. Segundo, [17], a primeira revolução ficou marcada pelo aparecimento das primeiras máquinas, transformando uma indústria outrora artesanal, numa indústria mecanizada, onde era

predominante o uso da máquina a vapor, permitindo o aumento da velocidade de produção. A segunda, por sua vez, ficou marcada pelo aparecimento das máquinas a combustão e pela utilização da energia elétrica, fazendo com que fosse possível produzir em massa e criar linhas de montagem. O aparecimento dos computadores, da digitalização e da automação, marcou a terceira revolução industrial.

O termo I4.0 foi inicialmente proposto em 2011 pelo governo Alemão, de modo a desenvolver a sua economia, [19], bem como, segundo os autores em [54], ser utilizado para aplicar três fatores interconectados, sendo estes:

- digitalização e integração de qualquer relação técnica e económica simples para redes técnicas e económicas complexas,
- digitalização de produtos e serviços,
- novos modelos de mercado.

De acordo com [17], esta nova revolução industrial permite o aparecimento de uma indústria descentralizada e flexível, onde é possível o fabrico de produtos personalizados, permitindo um aumento da eficiência económica e ecológica. Para tal acontecer, é atribuída às máquinas a capacidade de comunicar e tomar decisões, tornando-as inteligentes e cientes do seu estado de funcionamento e produção, [23]. Isto só é possível através da utilização de inúmeras tecnologias nomeadamente, *Cyber-Physical System* (CPS), *Radio Frequency Identification* (RFID), *Internet of Things* (IoT) e utilização da *cloud*[1], [19].

Instituições e empresas alemãs desenvolveram e publicaram um modelo capaz de representar todos os constituintes de um sistema de produção complexo, como o que é proposto na I4.0. Este modelo é denominado de *Reference Architecture Model Industry 4.0* (RAMI 4.0) e consiste num modelo em três dimensões, que permite distribuir os diferentes componentes do processo de produção de um produto, por dois eixos, sendo estes um eixo vertical e um eixo horizontal. O eixo vertical é composto por diferentes camadas, sendo estas:

- **camada dos componentes físicos**, sendo esta uma camada que representa a realidade, tais como ideias, arquivos, documentos, o ser humano, entre outros e encontra-se interligada com a componente virtual através da camada de integração,
- **camada de integração**, que tem como objetivo fornecer informação acerca dos componentes físicos, de forma a ser utilizada em processamento computacional,
- **camada de comunicação**, cuja função é providenciar uma comunicação genérica entre os componentes,

[1] *Cloud* é uma tecnologia que permite guardar informação na *internet*, em vez de no computador pessoal.

- **camada de informação**, cuja função é garantir a integridade da informação, integrar diferentes tipos de dados, obter informação de melhor qualidade, receber e transformar eventos, de modo a que estes sejam compatíveis com a informação presente nas camadas mais elevadas,

- **camada funcional**, tem como objetivo criar uma plataforma para integrar horizontalmente diferentes funções, bem como criar regras e decisões lógicas,

- **camada de negócio**, garante a integridade das funções relacionadas com o fluxo de valor, bem como mapear modelos de negócios e interligar outros processos relacionados com o negócio.

O eixo horizontal por sua vez é dividido pela parte esquerda e pela parte direita. A parte esquerda do eixo representa o ciclo de vida de cada produto, nomeadamente o tipo de cada produto ou máquina envolvendo o *design*, desenvolvimento e teste de protótipos e a instância de cada produto, ou seja, a representação de cada tipo de produto, após este ter sido produzido em massa. A parte direita do eixo representa todos os sistemas e máquinas flexíveis com funções distribuídas ao longo de toda a rede, capazes de interagir e comunicar ao longo de todos os níveis hierárquicos, [54].

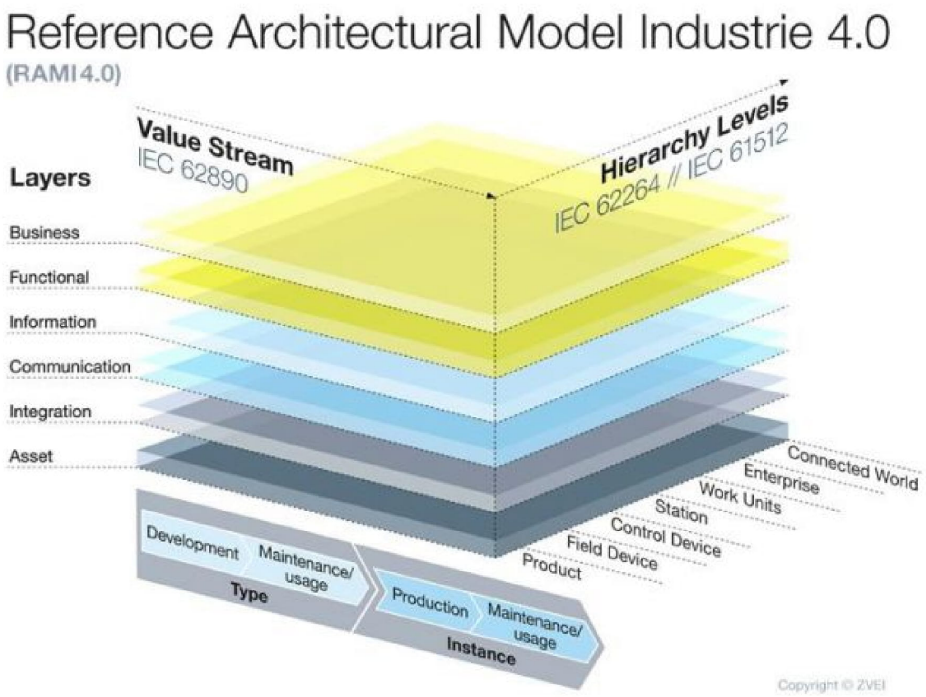


Figura 2.1: Modelo RAMI 4.0, [45].

Segundo [56], o paradigma da I4.0 é definido por três dimensões de integração, nomeadamente, a **integração horizontal**, a **integração vertical** e a **integração *end-to-end***.

A **integração horizontal** consiste na incorporação entre um recurso e uma rede de informação dentro da cadeia de valor, de modo a permitir uma cooperação total entre várias empresas, proporcionando um produto e serviço em tempo real. A **integração vertical**, por sua vez, integra numa só rede todos os sistemas de manufatura presentes numa fábrica inteligente, permitindo que a manufatura seja personalizada e dinâmica, em alternativa aos sistemas de produção tradicionais. Por fim, a **integração *end-to-end*** permite uma integração ao longo de toda a cadeia de valor, sendo esta implementada na premissa de que todos os terminais possuem uma cadeia de valor digital, incorporando diferentes empresas.

O objetivo principal da I4.0 é permitir o fabrico de produtos cada vez mais personalizados, de modo a preencher um mercado cada vez mais heterogéneo e ao mesmo tempo otimizar e melhorar o tempo, custo e modo de produção destes, [46].

Apesar de todas as vantagens e avanços supracitados, a I4.0 ainda apresenta alguns desafios e problemas. De acordo com [46], os sistemas ainda não apresentam as capacidades sociais e autonomia necessária para se tornarem totalmente independentes, a modelação de sistemas auto-sustentáveis ainda se encontra demasiado complexa, a interligação de todos estes sistemas numa só rede está sujeita a ataques informáticos, sendo que atualmente ainda não existem protocolos de segurança suficientes para prevenir estes ataques e por último, é necessário um grande investimento por parte da indústria, de modo a implementar todos estes sistemas e tecnologias. [31] defendem que um dos maiores desafios na I4.0, é a representação e troca de informação proveniente de elementos heterogéneos, sendo que o processo de generalização é considerado um dos maiores obstáculos na implementação desta nova indústria.

2.1.1 Papel Humano na Indústria 4.0

Uma das grandes preocupações presentes na I4.0 é o papel humano num ambiente altamente tecnológico, onde máquinas conseguem tomar decisões próprias e muitas das vezes realizar tarefas de forma mais eficiente do que um humano.

Segundo [46], a grande diferença entre a I4.0 e a **Manufatura Integrada por Computadores**, em inglês *Computer Integrated Manufacturing* (CIM), é o papel humano no ambiente de produção, onde CIM é considerado um ambiente totalmente independente do papel humano, enquanto a I4.0 é considerado um ambiente onde o papel humano é importante.

Em [3], é defendida uma aprendizagem mútua no contexto da I4.0, sendo esta *"(...) a bidirectional process involving reciprocal exchange, dependence, action or influence within human and machine collaboration, which results in creating new meaning or concept, enriching the existing ones or improving skills and abilities in association with each group of learners."*, ou seja, um método de aprendizagem bidirecional que envolve a colaboração entre a máquina e o humano, de modo a melhorar as capacidades de cada um. Nesta abordagem são identificados dois tipos de grupos de aprendizagem, os **humanos** e as **máquinas**. A

interação e colaboração entre máquina e humano é importante para criar conceitos de aprendizagem híbridos, onde há absorção de conhecimento por ambas as partes, sendo que também é necessário ter em conta a capacidade do humano e da máquina em realizar diferentes tipos de tarefas. Neste tipo de abordagem existe a possibilidade de uma tarefa ser realizada só por um humano, só por uma máquina ou por ambos. Também está a ser estudada a possibilidade de *robots* aprenderem certas operações, através de *machine learning*, ao observarem humanos a executá-las.

Assim é possível concluir que com a I4.0 serão criados novos postos de trabalho, em que os atuais não serão destruídos mas sim transformados, existindo uma interação e simbiose entre **humano-máquina**, de modo a levar a produção a outro nível.

2.2 Sistemas Ciber-Físicos de Produção

O conceito de sistemas ciber-físicos, ou *Cyber-Physical Systems* (CPS) em inglês, foi referido pela primeira vez em 2006, num *workshop* organizado pela *National Science Foundation* (NSF) em Austin, Texas, USA. Este termo foi definido como *"(. . .) a system composed of collaborative entities, equipped with calculation capabilities and actors of an intensive connection with the surrounding physical world and phenomena, using and providing all together services of treatment and communication of data available on the network."*, [34]. Esta definição significa que um CPS consiste num sistema que combina uma entidade virtual que abstrai e confere inteligência a uma entidade física numa relação de um para um, criando assim um *digital twin*[2] da componente física.

A evolução tecnológica, o aparecimento de novos sensores de custo reduzido, a criação de novas formas de aquisição de dados e um mercado cada vez mais personalizado e competitivo, está a obrigar a indústria a tornar-se cada vez mais dependente da implementação de altas tecnologias, [18]. Assim sendo, é possível fornecer a informação do sistema físico, adquirida por sensores, a serviços da *internet*, que por sua vez atuam no mundo físico através de atuadores, [37]. Na literatura, um CPS utilizado num sistema de produção é denominado de sistema de produção ciber-físico, ou CPPS. Um CPPS representa um sistema de sistemas, de elementos autónomos e cooperativos conectados uns com os outros através de todos os níveis de produção, [34].

Em [32], é utilizado um agente baseado em CPPS para desenvolver um ambiente capaz de integrar o processamento e controlo de qualidade num *multi-stage manufacturing* para obter uma percentagem nula de produtos com defeito, através da integração dos componentes de um *shop-floor*. Nesta implementação o agente baseado em CPPS é uma virtualização do sistema físico de produção, ou seja, de um *shop-floor*, que possui a capacidade de receber várias regras, provenientes de camadas mais elevadas, e utilizá-las para monitorizar e controlar os parâmetros de qualidade do sistema em tempo real, de modo a obter uma percentagem nula, ou quase nula, de produtos defeituosos.

[2] *Digital win* consiste numa representação virtual de algo físico, como por exemplo, o *digital twin* de uma transportadora física, será a sua representação virtual.

Segundo [37], um CPPS apresenta inúmeros benefícios para um sistema de produção, nomeadamente a otimização de processos de produção, possibilidades de configuração e conhecimento das áreas de aplicação. A produção de produtos personalizados é otimizada, visto que, as propriedades dos produtos, custo, tempo de produção, entre outros, são conhecidos por sistemas de produção inteligentes. Utilizando CPPS é possível tornar os sistemas mais eficientes, a nível de recursos, pois é possível melhorar a alocação destes. Por fim, as máquinas apresentam a capacidade de seguir as instruções e velocidade dos trabalhadores humanos, o que é importante pois estes processos são centrados nos humanos.

2.2.1 Arquiteturas de Sistemas Ciber-Físicos de Produção

A implementação de CPPS em sistemas de produção, só foi possível através da criação de arquiteturas. Uma dessas arquiteturas é denominada de arquitetura de 5 níveis de CPPS (5C).

Segundo [5], a arquitetura 5C consiste num conjunto de metodologias e instruções que permitem desenvolver e inserir um CPPS num sistema de produção, atribuindo a capacidade de influenciar o sistema desde a aquisição de dados até a análise e criação final do produto. Esta arquitetura possuí cinco níveis nomeadamente, o nível de conexão, o nível de conversão, o nível computacional, o nível cognitivo e o nível de configuração.

O nível de conexão consiste na ligação de sensores a máquinas de produção, de modo a adquirir informação sobre estas, para posteriormente, através de medições precisas e protocolos de comunicação, criar uma representação virtual destas. Por sua vez, o nível de conversão está relacionado com a transformação dos valores, medidos através dos sensores, para informação viável. O nível computacional descreve um modelo, análogo à entidade física, numa entidade virtual, *digital twin*. O nível cognitivo apresenta a função de sugerir ao utilizador qual a melhor ação a ser tomada, apresentando informação e dados de modo a auxiliar na tomada de decisão. Por fim, o nível de configuração permite realizar uma retroação com o mundo físico, [2].

Em [14], é proposta uma arquitetura 8C, onde são adicionadas três novas caraterísticas à arquitetura supracitada, sendo estas o **cliente**, o **conteúdo** e a **integração**. Estas permitem alargar a integração horizontal dos CPPS, bem como permitir a sua agregação a diferentes entidades.

O **cliente** é uma caraterística que se foca em tudo o que é afetado pelo cliente, desde o processo de produção até ao serviço pós-venda do produto. Com a utilização de fábricas inteligentes e todas estas tecnologias, é permitido ao cliente personalizar o seu próprio produto, rastrear o processo de produção e modificar especificações do produto durante a produção deste, permitindo assim uma produção que é simultaneamente personalizada e em massa. O **conteúdo** apresenta a função de armazenar toda a informação do produto, de modo a ser possível rastreá-lo. Através do conteúdo, é possível armazenar toda a informação do produto, o material utilizado, os fornecedores, os processos, os parâmetros

de produção entre muitas outras especificações do produto, bem como todos os serviços pós-venda, tais como componentes suplentes, manutenção e instruções de operação. Com todos estes dados é possível criar um ciclo de vida completo do produto, desde a sua produção até quando é reciclado. Por fim, a **integração** permite agregar diferentes entidades presentes no processo de produção, sendo assim possível agendar simultaneamente diferentes linhas de produção, o que leva à produção de produtos personalizados de uma maneira temporalmente eficiente e flexível, [14].

[14] defende que ao acrescentar as três caraterísticas supracitadas à arquitetura 5C, é possível criar uma arquitetura focada tanto na integração vertical como horizontal, sendo isso benéfico tanto para a produção personalizada como para a produção em massa, fazendo ênfase ao ciclo de vida do produto.

Em [2], é definida uma nova arquitetura, que consiste na junção da arquitetura 3C com a arquitetura do Sistema Ciber-Físico Antropocêntrico, ou em inglês *Anthropocentric Cyber-Physical System* (ACPS). Um ACPS é um sistema que centraliza a entidade humana, tendo como função apresentar o nível mais alto de abstração para uma arquitetura de referência cibernética antropocêntrica **(ACPA4SF)** para fábricas inteligentes. O ACPS vai para lá da arquitetura clássica de CPS, tentando capturar o significado de CPS e ao mesmo tempo distinguir as relações entre as suas entidades, sendo um modelo de referência para integrar os componentes principais de uma fábrica inteligente, nomeadamente, **componente física**, **componente computacional** e **componente humana**, sendo que esta última tem um papel importante na relação com os CPS, [33]. A nova arquitetura 3C, é baseada na arquitetura ACPS, utilizando três componentes chave nomeadamente, a **componente computacional**, a **componente humana** e a **componente física**. Esta utiliza também sub-elementos da arquitetura ACPS e parâmetros de ligação, tais como conectores e protocolos, [2].

Para sumarizar, um CPPS consiste numa representação virtual de um modelo físico, o que permite a virtualização de cada componente de uma máquina, através de dados fornecidos por sensores. Esta entidade virtual atribui à máquina certas caraterísticas, tais como tomada de decisão, *consciência* própria e auto predição, através da captura de registos temporais e da sintetização de passos futuros. Neste sentido, após a criação de uma entidade virtual para cada componente, é possível criar uma componente virtual para a máquina em si. Utilizando a informação recolhida pelos sensores e através das entidades virtuais, tanto dos componentes como das máquinas, é possível atribuir caraterísticas como adaptabilidade e manutenção preditiva a um sistema de produção, garantindo um sistema mais fiável com equipamento resiliente e inteligente, [5].

2.3 Sistemas Multi-Agente

Antes de compreendermos o que é um SMA, ou *Multi-Agent System* em inglês, é necessário compreender o que é um **agente**. O termo agente é há muito utilizado na literatura e possui diversas definições. Os autores em [39] defendem que um agente é *"(...) a*

persistent software entity dedicated to a specific purpose.", ou seja, um agente é uma entidade de *sofware* desenvolvida para um propósito específico.

Por sua vez, em [52] um agente é interpretado por duas definições. Na primeira um agente é considerado um sistema computacional baseado em *hardware* ou *software* e apresenta propriedades, como autonomia, interação social, reatividade e pro-atividade. O autor considera esta definição fraca e inconsistente. Na segunda definição, um agente é um sistema computacional, que possui todas as caraterísticas supracitadas, no entanto o seu desenvolvimento e implementação é baseado em conceitos e caraterísticas aplicadas em humanos, tentando recriar ao máximo a figura humana num sistema computacional.

Atualmente segundo [12], um agente possui também duas definições, uma mais fraca e geral e outra mais consistente e específica no contexto da IA. O autor defende que, de uma forma mais heterogénea, um agente é uma entidade autónoma, que tem a capacidade de interagir com outros agentes e tomar decisões que poderão influenciar um certo ambiente. Este defende também, a existência de uma outra definição mais consistente no contexto da IA, sendo esta, *"(...) an Intelligent Agent (IA) is an autonomous entity that observes through sensors and acts upon an environment using actuators (i.e., it is an agent) and directs its activity towards achieving goals (...)"*, ou seja, um agente inteligente é uma entidade autónoma cujo objetivo é alcançar uma finalidade, através da observação e atuação num sistema, utilizando sensores e atuadores. Estes possuem a capacidade de adquirirem conhecimento de modo a cumprirem os seus objetivos.

A aplicação e implementação destes agentes num sistema utópico e estático seria relativamente simples, no entanto na realidade estamos perante sistemas altamente dinâmicos e complexos. Como tal é necessário que os agentes apresentem propriedades como reatividade, pro-atividade e interação social, referido em [52].

Estas propriedades são importantes, visto que, o agente necessita de interagir e responder, em tempo real, às diferentes alterações do ambiente em que se encontram, ter a capacidade de trabalhar para um objetivo de longo prazo e conseguirem colaborar com outros agentes do sistema, de modo a concluírem uma tarefa mais complexa, [15], originando o conceito SMA. Este conceito já utilizado no século XX, é definido como um grupo de agentes que possuem a capacidade de evoluírem de forma dinâmica permitindo a coordenação das suas atividades, [25].

A atribuição destas caraterísticas a um agente pode ser feita através da aprendizagem pela experiência. Esta aprendizagem pode ser realizada utilizando técnicas de *machine learning*, sendo assim possível atribuir inteligência, de forma individual, aos agentes, de modo a melhorar e otimizar um sistema onde estes já comunicavam entre si. A aprendizagem de um agente num sistema onde não haja conhecimento prévio do seu funcionamento, pode ser feita através da técnica de RL. Segundo [15], RL é uma das técnicas mais adequadas para este tipo de sistemas, visto que, não necessita de conhecimento prévio de quais as ações corretas ou incorretas em cada estado. Através da interação do agente com o sistema, é possível observar-se os efeitos das suas ações no ambiente, sendo depois atribuído um sinal numérico a cada ação. O agente é então recompensado através de um

sinal numérico, sendo esta recompensa baseada na reação positiva do sistema à ação do agente.

2.4 *Reinforcement Learning*

Reinforcement learning é uma sub-classe de *machine learning* que difere de outras sub-classes, como *supervised* e *unsupervised learning*. O processo de RL baseia-se no comportamento humano, onde a experiência é adquirida através do desempenho de uma ação e da análise da resposta baseada nessa ação, sendo que respostas com efeito positivo são mais prováveis de acontecer do que aquelas que produzem um efeito negativo, após um processo de aprendizagem. Assim RL é aplicado a sistemas computacionais, de modo a ensiná-los a tomarem boas ações numa determinada situação, através da sua experiência com o ambiente em que se encontram.

Num sistema computacional, RL consiste na observação da resposta de um sistema, ou ambiente, a uma determinada ação causada por um agente, entidade que aprende e é capaz de tomar decisões. Esta ação é realizada de modo a maximizar uma recompensa escalar ou um sinal de reforço. Nesta técnica o agente necessita de descobrir quais as ações que irão originar uma maior recompensa através de tentativa e erro. Em certos casos, as ações do agente podem não só afetar a recompensa imediata, mas também todas as recompensas consecutivas, originando assim uma recompensa tardia. Tentativa e erro e recompensa tardia, são consideradas as duas caraterísticas mais importantes em RL, [42].

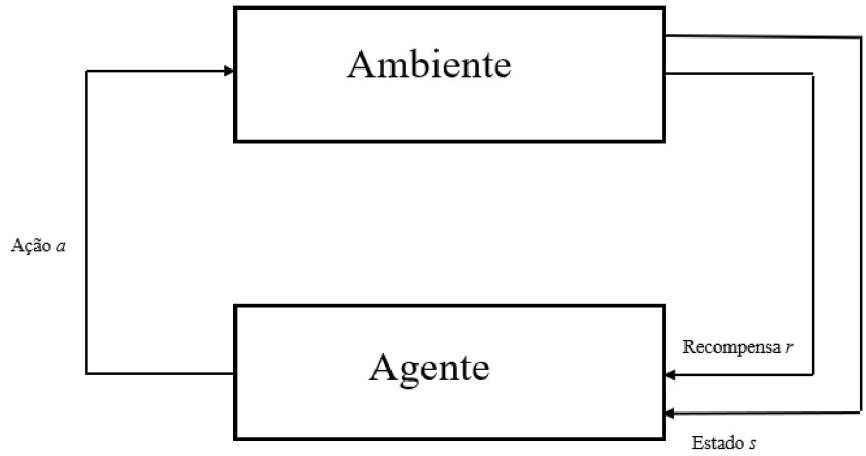


Figura 2.2: Modelo genérico de RL. Adaptado de [15].

Um modelo genérico de RL é constituído por:

- um conjunto de estados S;

- um conjunto de ações A;
- um conjunto de recompensas numéricas $R \in IR$;
- uma função de transição T.

De uma forma mais detalhada, em cada unidade de tempo t o agente recebe, como *input* i, uma indicação do estado atual, $s_t \in S$, e o conjunto de todas as ações possíveis, $A(s_t)$, nesse estado. Em seguida, este escolhe uma ação, $a \in A(s_t)$, e recebe, por parte do ambiente, o novo estado, s_{t+1}, e a recompensa, r_{t+1}, sendo assim possível mapear todas as probabilidades de selecionar cada ação para um determinado estado. Este mapeamento é denominado de política do agente, denotada por π_t, onde $\pi_t(s,a)$ corresponde à probabilidade do agente selecionar a ação a no estado s no tempo t. Por outras outras palavras, é a probabilidade de $a_t = a$ se $s_t = s$, [15].

Num problema de RL, o objetivo é definido pela função de recompensa, ou *reward function* em inglês, sendo o objetivo desta função mapear o par estado-ação do ambiente, num só valor numérico, definindo os eventos bons e maus para o agente. Este valor numérico é denominado de recompensa e tem o propósito de indicar o quão benéfico é para o agente realizar uma certa ação num respetivo estado. O objetivo de um agente de RL é o de maximizar a recompensa total recebida a longo prazo, [15].

2.4.1 *Q-Learning*

Um algoritmo de RL muito conhecido é o algoritmo *Q-learning*. *Q-learning*, que também pode ser visto como um método de programação dinâmica assíncrona, é um modelo onde não é necessário que o agente construa mapas do ambiente, visto que, este método fornece aos agentes a capacidade de aprender qual o procedimento ótimo num domínio *Markovian*, [51].

Esta técnica permite atualizar, sistematicamente, a função $Q(s_t, a_t)$, denominada de matriz Q em [6]. De notar que na literatura existem várias nomenclaturas para a função de *Q-learning*, sendo que nesta revisão de literatura será usada a nomenclatura Q. Esta função contêm as probabilidades de cada ação ser escolhida num determinado estado. A particularidade de *Q-learning* é que permite a atualização desta função, sem possuir qualquer conhecimento prévio do modelo do sistema. A regra para atualizar o par estado-ação (s,a), está apresentada na equação 2.1.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[\sum_{k=0}^{T-t} \gamma^k r_{t+k} - Q(s_t, a_t) \right] \tag{2.1}$$

O algoritmo de *Q-learning* tem como função guardar todos os estados, ações e recompensas após a execução de um episódio. Após a execução do episódio terminar, os valores de Q são atualizados.

Algoritmo simplificado de *Q-learning*

Escolha do ritmo de aprendizagem α
Inicialização arbitrária de Q
Execução do episódio:
Escolha de uma ação a_t no estado atual s_t de modo que
$a_t = argmax_{a \in A} q_\pi(s_t, a)$
Observação da recompensa r_t
Armazenar os valores de s_t, a_t e r_t
Para cada observação do par (s_t, a_t), **atualizar**:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[\sum_{k=0}^{T-t} \gamma^k r_{t+k} - Q(s_t, a_t) \right]$$

O algoritmo acima referido, apresenta algumas desvantagens. Segundo [9], cada estado, ação ou recompensa, observada em cada episódio, tem de ser armazenada de modo a atualizar Q no final de cada episódio, o que pode originar problemas de falta de memória, se estivermos perante um sistema com um elevado número de estados. Este algoritmo apresenta também a desvantagem de só atualizar a função Q no final de cada episódio, o que torna a aprendizagem lenta. Sendo assim, seria benéfico se fosse possível atualizar Q sempre que uma ação é escolhida e uma recompensa recebida.

As desvantagens supracitadas podem ser removidas com a utilização de *bootstrapping*. De acordo com [9], para entender o conceito de *bootstrapping*, é necessário reformular $Q(s_t, a_t)$, tal como presente na equação 2.2.

$$Q(s_t, a_t) = r_t + \sum_{k=1}^{T-t} \gamma^k r_{t+k} \tag{2.2}$$

O conceito de *bootstrapping*, consiste em aproximar o somatório, $\sum_{k=1}^{T-t} \gamma^k r_{t+k}$ com $\max_a Q(s_{t+1}, a_{t+1})$, de modo a que a regra de atualização do algoritmo de *Q-learning*, deixe de ser calculada pela equação 2.1 e passe a ser calculada pela equação 2.3.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \tag{2.3}$$

Utilizando o conceito de *bootstrapping* é possível atualizar Q após cada ação, como tal quanto mais frequentes forem as atualizações mais rápida será a aprendizagem, visto que, as atualizações poderão acontecer durante um episódio. Apesar de ser vantajosa, esta definição também apresenta um contratempo, sendo este a redução da precisão de cada atualização devido à aproximação do somatório, [9].

Quando um agente escolhe ações positivas, este tende em seguir o caminho associado a essas ações, ao invés de explorar ações inexploradas. Este fenómeno é originado porque a política do agente, π_t, é gananciosa e como tal tende a escolher a ação que, à partida, origina a maior recompensa. Isto acontece porque uma ação que é conhecida por possuir uma recompensa positiva, tem mais tendência a ser escolhida do que uma ação cuja recompensa é desconhecida. No entanto com a introdução de uma nova política,

denominada de $\epsilon - greedy$, é possível ultrapassar esta desvantagem. Esta nova política, consiste em atribuir uma probabilidade ϵ a cada ação para que esta seja escolhida. Para tal, sempre que o agente vai escolher uma ação, durante a fase de aprendizagem, com a probabilidade de exploração ϵ, este escolhe uma ação aleatória. Com a introdução desta aleatoriedade, é possível forçar o agente a explorar todo o ambiente.

Introduzindo o conceito de *bootstrapping* e a política $\epsilon - greedy$, é possível reformular o pseudo-código anteriormente apresentado e corrigir as desvantagens do algoritmo simplificado de *Q-Learning*.

Algoritmo de *Q-learning* com *bootstrapping* e política $\epsilon - greedy$

Escolha da probabilidade de exploração ϵ
Escolha do ritmo de aprendizagem α
Inicialização arbitrária de Q
Inicialização de um novo episódio e estado s_0
Execução infinita:
  Escolha de uma ação a_t no estado atual s_t, de acordo com a política $\epsilon - greedy$
  Observação da recompensa r_t e do próximo estado s_{t+1}
  Atualização de Q(s,a):
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$
  Se s_{t+1} for o último estado
    Break

A função de *Q-learning* pode ser representada através de uma *lookup table fashion*, ou seja, uma tabela onde o algoritmo observa cada par estado-ação e procurar o valor Q correspondente. Com a utilização desta tabela, cada estado e cada ação têm de ser visitados múltiplas vezes, de modo a que seja possível realizar uma atualização dos valores e originar uma boa aproximação do valor Q, sendo assim necessário que o conjunto de pares estado-ação seja discreto e finito. A desvantagem aparece quando o sistema em questão é um sistema sem limites, contínuo no estado-espaço ou possui inúmeros estados e ações. Perante um sistema limitado mas com um grande número de estados, é muito demorado visitar todos os espaços, por sua vez perante um sistema ilimitado ou contínuo no estado-espaço é impossível visitar todos os espaços, isto porque a probabilidade de visitar novamente um estado único é zero. O método *lookup table fashion* é bom para ser utilizado em sistemas discretos e pequenos, porém a sua utilização não é possível nos sistemas supracitados, como tal é necessário aproximar uma função para representar $Q(s, a)$, [9].

Em sistemas muito complexos, é necessário que o mapeamento do par espaço-ação para o valor Q seja condensado numa função, de modo a que seja possível utilizar RL neste contexto. Assim, é necessário utilizar uma função de aproximação que aproxima a função Q(s,a), de modo a encontrar uma função parametrizada $f(x;\theta)$ que satisfaça a condição $f(x) \approx f(x;\theta)$, para prever valores Q em estados ainda não explorados. As diferenças entre a *lookup table* e o método da função de aproximação estão presentes na

figura 2.3. Nesta é possível observar que na *lookup table* os pares x_i e y_i são armazenados, enquanto na função de aproximação é possível obter-se uma aproximação para um *input* que passe pela função $Q(s,a;\theta)$, onde só os parâmetros θ são armazenados.

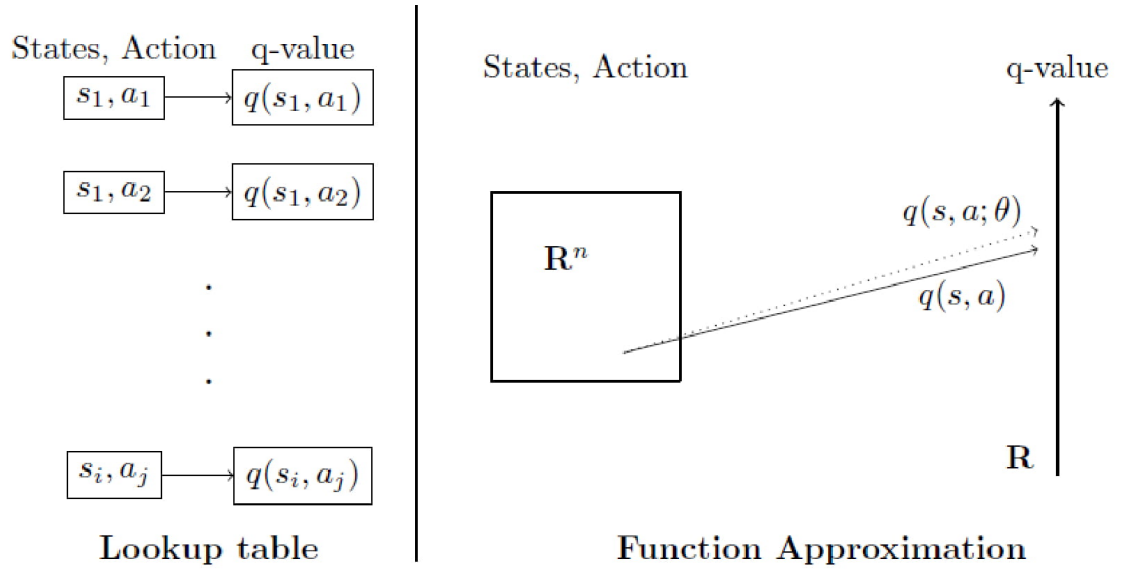


Figura 2.3: Ilustração de uma *lookup table* à esquerda e uma aproximação de função à direita, [9].

As redes neuronais artificiais, ou *Artificial Neural Network* (ANN) em inglês, são funções parametrizadas que processam informação de uma forma semelhante às redes neuronais biológicas. As ANN são um grupo versátil e poderoso no que toca à aproximação de funções, onde a escolha apropriada de parâmetros faz com que estas sejam capazes de aproximar funções não lineares e complexas.

A união entre RL, mais especificamente entre os algoritmos de *Q-learning* acima referidos e a utilização de ANN, nomeadamente *Deep Artificial Neural Network* (DANN), para aproximação da função Q(s,a), faz com que seja possível originar uma nova ramificação de RL, denominada de *Deep Reinforcement Learning* (DRL), útil para a resolução de problemas em sistemas complexos.

2.4.2 *Deep Reinforcement Learning*

A técnica de RL apresenta vários tipos de algoritmos, sendo que o mais interessante para esta revisão de literatura é o algoritmo *Q-learning*, já referido anteriormente. Com a união entre RL e DANN foi criado um novo algoritmo, sendo este o algoritmo *Deep Q-Network* (DQN). Um algoritmo poderoso, que permite a aproximação da função Q(s,a) com a função parametrizada $Q(s,a;\theta)$, permitindo assim a sua utilização em sistemas mais complexos.

O algoritmo DQN, introduzido por [22], consiste num único algoritmo capaz de desenvolver uma grande variedade de competências para realizar diversas tarefas. Os ambientes

utilizados foram os jogos da consola *Atari 2600*[3], onde o agente é capaz de aprender e jogar todos os jogos, tendo melhores prestações que jogadores profissionais em grande parte deles, tal como pode ser comprovado na figura 2.4.

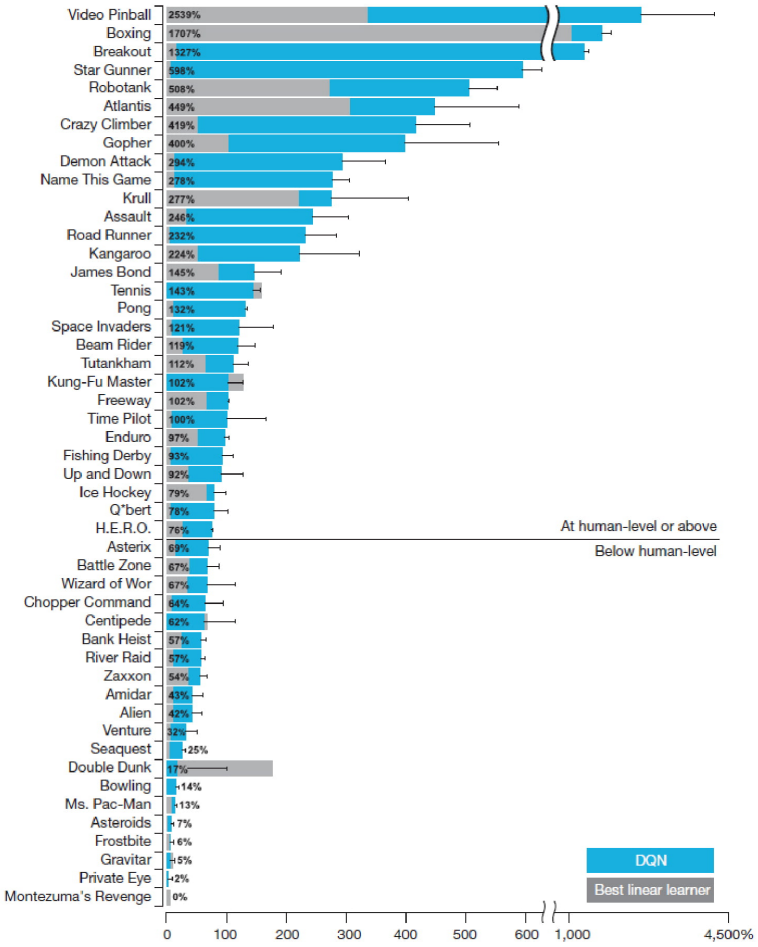


Figura 2.4: Figura que representa a prestação de um algoritmo DQN, comparativamente a um dos melhores algoritmos de aprendizagem linear, bem como a prestação humana, em jogos da consola Atari 2600, [22].

Quando um agente DQN é treinado, a componente ANN é treinada de modo a minimizar o custo da função 2.4.

$$C(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, \hat{y}_i). \tag{2.4}$$

Na função 2.4, **N** representa o *batch size*[4], θ representa o vetor que possui os pesos e as inclinações, *biases* em inglês, da rede neuronal e y_i é o valor que a ANN tenta aproximar, sendo assim considerado o valor alvo a atingir. Este valor alvo não se encontra explícito

[3] *Atari 2600* é uma consola lançada nos anos 80, que veio revolucionar a indústria dos vídeojogos, trazendo jogos como *PacMan*, *Breakout*, entre muitos outros.

[4] *Batch size* é um termo utilizado em *machine learning*, que se refere ao número de exemplos de treino utilizados numa iteração.

nos dados recolhidos através da experiência, como tal é necessário aproximar este valor para que seja possível treinar a ANN, sendo que quando o valor alvo é aproximado os dados recolhidos durante a experiência podem ser utilizados. Quando t representa a unidade temporal, o valor alvo $y_i = y_t$, pode ser representado pela equação 2.5.

$$y_t \equiv r_{t+1} + \gamma \max_a q(s_{t+1}, a; \theta). \tag{2.5}$$

O algoritmo que utiliza a equação 2.5 apresenta algumas desvantagens. Primeiramente, se os dados utilizados forem retirados das atualizações mais recentes, existe uma grande correlação entre eles e como tal, os dados provenientes das atualizações mais antigas perderão impacto com as novas amostras, fazendo com que a rede perca a sua propriedade de generalização, [26]. Em segundo lugar, se *s* e *s+1* forem iguais ou similares a rede tornar-se-á instável, isto porque a rede prevê o valor do estado seguinte, *s+1*, ao mesmo tempo que está a ser atualizada, [22]. Por último, segundo [47], o algoritmo DQN sobrevaloriza os estados, ou seja, quando a ação correspondente ao maior valor de Q no estado *s+1* é escolhida, é provável que esta seja uma sobrevalorização dessa mesma ação. Este fenómeno é originado devido à utilização da rede principal na avaliação da ação ótima correspondente ao estado seguinte, *s+1*.

O primeiro problema referido, é resolvido através do armazenamento das transições mais antigas numa *replay memory*. Esta tem como função guardar a estrutura de dados (s_t, a_t, r_t, s_{t+1}), onde a_t representa a ação tomada pelo agente, s_t representa o estado atual, r_t retrata a recompensa e s_{t+1} corresponde ao estado seguinte. Utilizando esta memória, é possível utilizar-se experiências não correlacionadas para treinar a rede, sendo isto possível através da amostragem das transições armazenadas. Esta, como todas as memórias, é limitada, no entanto caso esteja perto da sua capacidade máxima elimina a experiência mais antiga, para que seja possível armazenar novas experiências. Em suma, a *replay memory* permite guardar de forma ordenada e sistemática toda a experiência adquirida pelo agente, aumentando a quantidade de dados que poderá ser utilizada e como consequência possibilitar o aumento da qualidade de treino da rede para menos experiências, [21].

Introduzindo uma segunda rede neuronal, denominada de **rede alvo**, cópia da primeira rede, **rede principal**, é possível resolver a segunda desvantagem acima referida. Segundo [22], a rede alvo é unicamente utilizada para prever qual o valor da ação ótima do próximo estado, *s+1*, ser escolhida quando a rede principal é atualizada. Através da utilização deste método, a rede principal torna-se mais estável, visto que, a rede alvo é congelada enquanto a rede principal é atualizada. A rede alvo, por sua vez, é atualizada com uma certa frequência, copiando os pesos da rede principal. Utilizando os parâmetros de fixação, denotados de θ^-, a expressão y_t fica conforme a expressão apresentada na equação 2.6.

$$y_t \equiv r_t + \gamma q(s_{t+1}, \arg\max_a q(s_{t+1}, a; \theta); \theta^-). \tag{2.6}$$

À semelhança dos algoritmos de *Q-learning*, em baixo encontra-se em pseudo-código, o algoritmo de Double-DQN com uma política $\epsilon - greedy$.

Algoritmo de *Double* DQN com uma política $\epsilon - greedy$

Inicialização de um novo episódio e do estado s
Enquanto $s \neq s_{terminal}$:
Realizar uma ação a no estado atual de acordo com a política $\epsilon - greedy - Q$
Observar a recompensa r e o próximo estado s+1
Armazena (s,a,r,s+1) na *replay memory*
$NAT \leftarrow NAT + 1$, sendo NAT o número de ações tomadas
Se $NAT \geq a_{starttraining}$
Atualiza os pesos da rede neuronal θ
num conjunto de transições amostradas da *replay memory*:
$y \leftarrow r_t + \gamma q(s_{t+1}, \underset{a}{argmax} q(s_{t+1}, a; \theta); \theta^-)$
$\theta \leftarrow \theta + Adam(y, q(s, a; \theta))$
Se $NAT(mod f_{atualizado}) == 0$
$\theta^- \leftarrow \theta$
$s \leftarrow s + 1$

2.5 Escalonamento da Produção

Primeiramente é necessário entender o que é o escalonamento na manufatura e o porquê de ser um problema. Segundo [29], o escalonamento em manufatura é uma subsecção do produto cartesiano de três conjuntos, **o quê**, **quando** e **onde**, respondendo a três perguntas:

- **O que**, corresponde ao conjunto de tarefas a serem cumpridas;
- **Quando**, corresponde ao conjunto de intervalos de tempo, $< iniciar, parar >$, que estão associados com o tempo que uma tarefa demora a ser realizada;
- **Onde**, corresponde à localização geográfica do conjunto de máquinas, matérias-primas e outros recursos numa fábrica.

O escalonamento é uma subsecção do conjunto **O quêXQuandoXOnde**, especificando os recursos que cada tarefa necessita e quando necessita deles. [29] define W=**OquêXQuandoXOnde**, implicando a existência de uma função de avaliação, para a seleção de um escalonamento. A função de avaliação $F : (2^W) \rightarrow R$ tem como objetivo transformar o conjunto W em valores reais, de modo a criar um resultado para cada escalonamento, possibilitando a escolha de um escalonamento ótimo.

[29] também defende que existem cinco desafios que tornam o escalonamento difícil, sendo estes a **estocasticidade**, a **tratabilidade**, a **desejabilidade**, o **caos** e a **decidibilidade**. A **estocasticidade** e a **desejabilidade** estão presentes em todos os problemas de

escalonamento. A **tratabilidade** encontra-se em praticamente todos os problemas realistas de escalonamento. Por fim, o caos e a **decidibilidade** encontram-se em ambientes de manufatura flexíveis.

A **desejabilidade** de um escalonamento é originada quando, na prática, queremos resolver o problema começando com o espaço W e uma função de avaliação F, que reflete os objetivos económicos e através destes dois fatores, retirar um escalonamento S que minimize o valor de F. A **estocasticidade** em manufatura, por sua vez, consiste no desvio imprevisível dos parâmetros de operação, comparativamente àqueles assumidos na altura de computação de um escalonamento. Por outro lado, a **tratabilidade** de um sistema carateriza a dificuldade computacional de analisar o seu comportamento. Um **sistema caótico** é um sistema determinístico e simples, cuja trajetória no espaço de fase nunca se repete, no entanto ocupa uma região consideravelmente menor do que o espaço total. Isto significa que sistemas caóticos idênticos, cujas condições iniciais diferem por pequenas quantidades, podem divergir exponencial durante a sua operação, fazendo com que seja impossível realizar um diagnóstico mais detalhado. A **decidibilidade** caracteriza a possibilidade de analisar o comportamento de um sistema, querendo isto dizer que se não existir nenhum algoritmo computacional capaz resolver o comportamento de uma certa máquina, faz com que esta seja indeterminável. Quando uma fábrica apresenta uma configuração de máquinas com estas caraterísticas, torna-se impossível de realizar o escalonamento de uma forma preditiva, [29].

Um dos problemas de escalonamento é o problema de encaminhamento dos produtos, de uma forma dinâmica e eficiente. [53] defendem que para ser possível obter um encaminhamento flexível, é necessário que o escalonamento, do sistema em questão, seja responsivo e robusto. Para tal os autores alegam que a utilização de um SMA, traria as caraterísticas necessárias para a implementação de uma política de encaminhamento flexível e dinâmica. Segundo [48], é benéfica a utilização de um SMA para a otimização de políticas de encaminhamento, pois existe cooperação entre os agentes. No entanto, os autores defendem que a cooperação entre agentes, bem como a inteligência coletiva do sistema, não são suficientes para uma implementação bem sucedida. Para tal é necessário atribuir a cada agente inteligência individual, de modo a que, juntamente com a comunicação com os outros agentes, estes tomem as decisões da forma mais eficiente possível. A inteligência individual pode ser adquirida através de técnicas de *machine learning*, como por exemplo RL ou DRL, sendo que estas permitem que a aprendizagem do agente seja realizada através de tentativa e erro, sem conhecimento prévio do ambiente de produção.

Na literatura existem vários problemas de manufatura referentes a diferentes tipologias de sistemas de produção. Nesta revisão de literatura, bem como nesta dissertação, é estudado o problema de uma *job-shop*. Segundo [8], o problema de uma *job-shop* consiste num sistema em que existem **N** operaçãos, iguais ou diferentes, presentes em **M** recursos, onde a matéria-prima tem de passar, de modo a ser manipulada para produzir o produto final, sempre visando a diminuição do tempo de produção, assumindo os seguintes fatores:

- cada recurso só pode realizar apenas uma operação de cada vez,
- o processamento de uma operação num recurso é chamado de operação,
- uma operação não pode ser interrompida,
- a sequência de recursos é desconhecida e tem de ser determinada, de modo a diminuir o tempo de produção de um produto.

2.5.1 Heurísticas Tradicionais na Resolução de Problemas de Escalonamento

Em 2001, [55] utilizaram um algoritmo genético heurístico híbrido para resolver o problema de escalonamento numa *job-shop*, bem como regras heurísticas de escalonamento para gerar novos indivíduos, com o objetivo de encaminhar a evolução genética. Proveniente de uma analogia com a evolução biológica, o algoritmo genético consiste numa técnica de pesquisa aleatória para otimização baseada no mecanismo de seleção natural. Segundo os autores, o objetivo de escalonamento numa *job-shop* é o de encontrar a sequência ótima na qual o produto passa por todas as máquinas, necessárias à sua produção, seguindo as seguintes propriedades:

- compatibilidade com as limitações tecnológicas,
- otimização baseada em critérios específicos para desempenho.

Os autores concluíram que com a utilização deste algoritmo, foi possível obter resultados superiores comparativamente a outras abordagens populares, no entanto apesar do algoritmo genético ter provado que é eficiente na resolução de problemas de escalonamento numa *job-shop*, à data deste artigo, ainda era necessário melhorar a sua eficiência. Foi também concluído que a utilização de regras heurísticas e técnicas de pesquisa local, juntamente com o algoritmo genético, permitia acelerar a convergência deste algoritmo.

[10] propõem um modelo matemático juntamente com abordagens heurísticas, de modo a resolver o problema de escalonamento numa *Flexible Job-Shop Problem* (FJSP). Nesta abordagem, o modelo matemático é utilizado com o propósito de alcançar uma solução ótima para uma *job-shop* de tamanho reduzido, enquanto as abordagens heurísticas, sendo estas abordagens integradas ou hierárquicas, são utilizadas para resolver os problemas presentes numa *job-shop* de tamanho real. Os autores defendem que o problema de uma FJSP encontra-se dividido em dois subproblemas, sendo estes o **subproblema de encaminhamento** e o **subproblema de escalonamento**. O **subproblema de encaminhamento** consiste na atribuição de uma operação a uma máquina, presente num conjunto de máquinas capazes de realizar essa operação, enquanto o **subproblema de escalonamento** consiste em criar uma sequência de todas as operações, de modo a se obter um escalonamento viável, minimizando a função objetivo predefinida. Neste estudo foram utilizadas duas abordagens heurísticas, como referido anteriormente, para resolver o problema de uma *job-shop* real. Na abordagem hierárquica a atribuição de operações a máquinas e a sequência de operações são tratadas de forma independente, enquanto na abordagem integrada a atribuição de operações e a sequência de operações são utilizadas simultaneamente. Os autores concluíram que no caso de uma *job-shop* de pequena dimensão, o algoritmo **HSA/TS** obtém o melhor rendimento, enquanto no caso de uma *job-shop* de maior dimensão, os algoritmos **HTS/SA** e **HTS/TS** são os que apresentam um melhor desempenho.

Em [24], os autores defendem que atualmente estamos presente grandes alterações relativamente à estrutura e topologia de pequenas e médias empresas, onde a manufatura está a enfrentar novos desafios, sendo necessário diminuir o custo de produção, bem como o tempo que o produto demora a ser produzido e a chegar ao mercado. Para tal neste estudo, é referido a criação de ramificações de uma empresa em países sub-desenvolvidos, substituindo a tradicional produção numa única fábrica para uma manufatura distribuída. Primeiramente, foram criados dois modelos matemáticos, sendo o objetivo destes representar *job-shops* distribuídas e os respetivos problemas. A avaliação destes modelos matemáticos, foi realizada com base na sua complexidade computacional, concluindo que o modelo baseado numa sequência é mais eficiente, visto que, apresenta um menor número de restrições. Em seguida os autores propuseram uma regra de atribuição para o sistema em questão, bem como três heurísticas gananciosas, de modo a resolver de uma forma mais eficiente o problema apresentado. As heurísticas apresentadas **SPT**, **LPT** e **LRPT** são baseadas em certas regras, sendo que estão relacionadas com o conceito de inserção numa vizinhança. O objetivo dos autores era adicionar operações numa sequência, de modo a construir uma permutação de operações. Os autores concluíram que, apesar de apresentar inúmeras vantagens, o esquema de permutações era redundante, sendo assim necessário uma análise cuidada para que fosse possível eliminar as permutações redundantes.

2.5.2 *Reinforcement Learning* na Resolução de Problemas de Escalonamento

RL começou a ser utilizado para resolver problemas de escalonamento de produção no contexto de ambientes dinâmicos. É possível encontrar na literatura vários autores que utilizaram RL para resolver paradigmas de escalonamento de produção, como o problema de escalonamento numa *job-shop* parcialmente flexível ou o problema de escalonamento num sistema estocástico.

Em 1999, [35] utilizaram o algoritmo de *Q-learning* para atribuir inteligência a agentes, de modo a que estes tivessem a capacidade de aprender e otimizar uma política de execução local dinâmica para resolver problemas de escalonamento numa *job-shop*. Estes agentes possuíam a aptidão de aprender, de forma autónoma, uma política de execução através do *feedback* do comportamento dinâmico do sistema de produção, sendo capazes de lidar com uma maior quantidade de informação e assim alcançar políticas de execução mais complexas e sofisticadas, comparativamente às heurísticas tradicionais. Os agentes foram treinados utilizando uma regra de aprendizagem, adaptada ao processo de decisão local, baseada em *Q-learning*, de modo a que a aprendizagem permitisse uma melhoria constante da política de decisão, no que toca à otimização dos custos globais. Os autores aplicaram dois casos experimentais, sendo que no primeiro utilizaram um único recurso, de modo a demonstrarem o modo de funcionamento e as capacidades de desempenho do agente e num segundo utilizaram múltiplos recursos, demonstrando a capacidade do agente de trabalhar num ambiente multi-agente. Ambos os casos revelaram a capacidade do agente de aprender políticas locais, que permitissem otimizar o comportamento global do sistema. Estes concluíram que as políticas dos agentes podem ser generalizadas para situações desconhecidas sem serem treinados novamente e, através dos resultados, concluíram também que as políticas dinâmicas são mais vantajosas do que as heurísticas tradicionais.

[4] propõem um sistema de escalonamento dinâmico baseado num agente inteligente, onde um ambiente simulado realiza certas atividades segundo uma regra de prioridade, sendo que esta é escolhida pelo agente e constituída pelas condições atuais do sistema. Os autores referem várias desvantagens do algoritmo de *Q-learning*, sendo que a falta de generalização é a mais problemática neste contexto. Como tal, desenvolveram um algoritmo, baseado em *Q-learning*, denominado de *Q*-III. Este é a combinação do algoritmo *Q*-II, desenvolvido por [28], e o algoritmo de *Hard c-Means*, [36], onde *Q*-II permite que o agente atualize as suas experiências temporais para todas as ações relacionadas, enquanto o algoritmo *Hard c-Means* converte estas experiências, de modo a generalizá-las. De forma a analisarem o desempenho de escalonamento do agente, os autores executaram vários exemplos do ambiente simulado, onde em cada um aplicaram uma regra de prioridade diferente, durante um ciclo de simulação. Por sua vez, o agente foi treinado com diferentes parâmetros, nos exemplos supracitados. Os resultados experimentais revelaram que em maior parte dos casos, comparativamente às heurísticas tradicionais, esta abordagem obteve melhores resultados.

Em [30] é estudada uma solução, baseada em RL e multi-agentes, para resolver o problema de escalonamento num sistema estocástico, ou *Stochastic Economic Lot Scheduling Problem* (SELSP) em inglês. Este problema ocorre quando é necessário realizar o escalonamento de inúmeros produtos presentes numa única instalação, cujo ambiente de produção é aleatório. As políticas geradas pelo algoritmo de RL implementado neste estudo, tinham a inconveniência de serem de difícil compreensão, como tal foi necessário implementar um classificador baseado numa árvore de decisão, de modo a permitir uma melhor compreensão do desempenho do agente. Os autores concluíram que a solução de multi-agentes com RL utilizada, foi capaz de reduzir os níveis de *stock* base, enquanto manteve um baixo nível de produtos atrasados, sendo considerada uma abordagem vantajosa para sistemas dinâmicos. A abordagem proposta pelos autores possui também a capacidade de ser aplicada numa enorme variedade de sistemas de produção que envolvam políticas de controlo dinâmicas.

Em [6], é criada uma distribuição controlada por CPPS, que utiliza produtos inteligentes, ou *Intelligent Product* (IP) em inglês treinados com RL, mais especificamente com o algoritmo de *Q-learning*, para resolver o problema de escalonamento numa *job-shop* parcialmente flexível, ou *Partially-Flexible Job-Shop Problem* (P-FJSP) em inglês. No problema de escalonamento de uma FJSP, uma operação pode ser realizado por mais do que um recurso, ou *Service Product* (SP) em inglês, sendo que para se obter uma solução eficiente é necessário que a atribuição de operaçãos para cada recurso seja realizada à priori. A resolução deste problema consiste na otimização de vários parâmetros, como a sequência de serviços, a atribuição de recursos para cada serviço e a escolha dos tempos de início e paragem. FJSP pode ser classificado como *Total Flexible Job-Shop Problem* (T-FJSP), em inglês ou P-FJSP. Nesta abordagem, o IP identifica o estado atual e através da matriz Q escolhe qual ação deve de realizar a seguir. Após ter sido feita a escolha da ação, é realizada uma seleção de regras e a função recompensa é calculada, tendo como base a eficiência da decisão passada. Através desta função é possível reajustar os valores da matriz Q, possibilitando uma aprendizagem gradual. Os autores concluíram que com a utilização de RL, foi possível melhorar o desempenho global do CPPS, bem como fazer com que o IP resolvesse os problemas de escalonamento de uma maneira eficaz e descentralizada.

[48] utilizam um SMA e RL, especificamente o algoritmo *Q-learning*, para atribuir inteligência coletiva e inteligência individual aos agentes, de modo a criar uma política de encaminhamento flexível, dinâmica e eficiente. Os autores utilizam os agentes para representar cada recurso e cada produto. Nesta abordagem são utilizados protocolos de comunicação, utilizados em SMA, para permitir a comunicação entre os recursos e os respetivos produtos, sendo assim possível a atribuição de inteligência coletiva ao sistema, permitindo caminhar ao encontro de um escalonamento e encaminhamento flexível e dinâmico. Após a criação do SMA, os autores utilizaram o algoritmo *Q-learning* para fornecer inteligência individual aos agentes, possibilitando a criação de uma política de encaminhamento otimizada e eficiente. Esta é utilizada pelo agente para tomar decisões baseado no seu estado inicial, sendo que no caso específico deste estudo, a política de

encaminhamento de um agente produto, é um mapeamento de todos os agentes recursos que o agente pode escolher, baseado no seu estado atual. Neste estudo é concluído que o algoritmo *Q-learning* é uma abordagem promissora, no que toca à otimização de políticas de encaminhamento de produtos dinâmicos, num SMA. Os autores também concluíram que este algoritmo obteve melhores resultados, comparativamente às heurísticas tradicionais, quando o sistema está a operar sob pressão, ou seja, com datas de entrega mais curtas. No caso das datas de entrega mais espaçadas a abordagem não foi tão benéfica, visto que, existem vários níveis de penalidades, no entanto só existe um nível de recompensa, ou seja, a função de recompensa não faz uma diferenciação dos valores quando a decisão é boa, apenas quando a decisão é má.

2.5.3 *Deep Reinforcement Learning* na Resolução de Problemas de Escalonamento

Analisando os artigos da sub-subsecção 2.5.2, é possível concluir que a utilização de RL é benéfica para problemas de escalonamento da produção. O próximo passo será utilizar algoritmos de DRL como solução para estes problemas, no entanto, à data desta dissertação, a aplicação destes algoritmos na resolução de problemas de escalonamento de manufatura ainda é escassa, fazendo com que a quantidade de artigos e estudos acerca deste tema seja limitada.

[50] aplicaram o algoritmo de DQN da *Google DeepMind*, para resolver o problema de escalonamento, referente à produção de semicondutores. Neste artigo é utilizado um ambiente de RL, para treinar agentes de DQN, ou seja, agentes que utilizam DANN para tomar decisões. Nesta abordagem, durante a fase de pré-treino, os agentes de DQN aprendem as estratégias de escalonamento dinâmico existentes, de modo a obterem experiência sobre estas e tornarem-se mais versáteis e dinâmicos. Os autores defendem que com a aplicação de DRL estamos cada vez mais perto da I4.0, onde será possível obter um sistema descentralizado que seja capaz de se otimizar, aprender e organizar por ele próprio. Os autores concluíram que com a utilização de agentes DQN num ambiente RL é possível que o sistema desenvolva um escalonamento global ótimo sem a intervenção humana ou conhecimento prévio, que os agentes sejam versáteis de modo a serem utilizados noutros ambientes numa questão de horas, que o sistema adquira a capacidade de se adaptar a cada conjunto de objetivos e que com a utilização deste método a quantidade de produtos atrasados diminua significativamente.

2.6 Conclusões Gerais

Através deste capítulo, é possível constatar que estamos perante um mercado cada vez mais individualizado, onde existe uma enorme procura por produtos personalizados, sendo necessário criar sistemas descentralizados, cada vez mais inteligentes e autónomos.

Um dos problemas intrínsecos às linhas de produção referidas neste capítulo, é o escalonamento da produção, mais especificamente o encaminhamento dos produtos, onde existe a constante procura por soluções que sejam capazes de encontrar a política de encaminhamento ótima para uma linha de produção. Uma tecnologia utilizada nestas linhas de produção são os sistema multi-agente, porém como já foi referido a atribuição de inteligência individual é também importante.

Inicialmente começou-se por utilizar algoritmos heurísticos, no entanto com o desenvolvimento tecnológico a utilização de inteligência artificial tornou-se uma solução cada vez mais viável. RL foi utilizado em diversos estudos, visto que, permite a aprendizagem através de tentativa e erro, não sendo necessário qualquer conhecimento prévio do sistema, porém este apresenta algumas desvantagens, como por exemplo não ser viável para ambientes contínuos ou de elevada complexidade.

O desenvolvimento tecnológico veio tornar possível treinar redes neuronais cada vez mais rapidamente. Com isto, foi possível juntar a tecnologia das redes neuronais com o algoritmo de RL, possibilitando a criação de um único agente de IA versátil, capaz de aprender vários ambientes, com diferentes objetivos. Este novo algoritmo não se encontra muito presente na literatura, no que toca à resolução de problemas de encaminhamento, tornando-se uma oportunidade de trabalho e estudo.

Capítulo 3

Desenho do Sistema

Neste capítulo, primeiramente, é abordada a definição de RL, onde é desmistificado o conceito de *Markov Decision Process*, MDP, visto ser fulcral para a implementação de qualquer algoritmo de RL, sendo também explicadas as caraterísticas que um ambiente, ou simulação, necessita de apresentar para que seja possível implementar e treinar um algoritmo de RL. Na secção referente aos algoritmos inteligentes, são também apresentados os conceitos chave para a implementação dos algoritmos *Q-learning* e *Double-DQN*, seguida de uma explicação do funcionamento genérico do algoritmo heurístico *A**.

Por fim, são abordados, de uma forma genérica, os pontos fulcrais para a implementação de uma simulação, através de um SMA, capaz de representar uma linha de produção do tipo *job-shop*, apresentando as caraterísticas necessárias para servir de ambiente para um algoritmo de RL. Nesta última secção é também proposta uma solução genérica, de como implementar um algoritmo inteligente num cenário real.

3.1 Algoritmos Inteligentes

Nesta dissertação é proposta a utilização de IA, mais especificamente um algoritmo de DRL denominado de *Double Deep Q-Network*, *Double-DQN*, para fornecer inteligência individual a agentes.

Como forma de comparação, foram desenvolvidos outros dois algoritmos, utilizados para a resolução deste tipo de problema, sendo estes o algoritmo *Q-Learning*, proveniente de RL e o algoritmo heurístico *A**.

3.1.1 *Reinforcement Learning*

Reinforcement Learning, RL, consiste na aprendizagem de um agente, através de tentativa e erro. Um agente de RL durante o seu treino realiza ações num certo ambiente, de modo a

maximizar uma recompensa, que é acumulada ao longo do treino. Essas ações vão alterar o ambiente e como tal o agente vai receber uma recompensa consoante a alteração que a ação escolhida teve no ambiente. Após diversas iterações de treino, através de tentativa e erro, o agente começa a compreender qual o conjunto de ações adequadas, de modo a otimizar a política de recompensas, que por sua vez faz com que consiga cumprir o seu objetivo, num dado ambiente.

3.1.1.1 *Markov Decision Process*

Na subsecção 3.1.1, é referido que RL consiste na observação de um ambiente, por parte de um agente, que por sua vez realiza ações num estado atual desse mesmo ambiente. Este agente aprende através de recompensas, que variam consoante a consequência que a sua ação teve no próximo estado do ambiente. Em suma, o processo de um agente realizar uma ação num estado atual de um ambiente e, posteriormente, receber uma recompensa consoante o estado seguinte desse mesmo ambiente, denomina-se de *Markov Decisions Processes*, ou MDP. Este processo é utilizado para descrever um ambiente totalmente observável, ou seja, um ambiente que permite a implementação de um algoritmo de RL.

Um MDP é descrito pelo tuplo (S, A, P, R, γ). O espaço de estados, **S**, engloba todos os estados em que o agente se pode encontrar. Por sua vez, o espaço de ações, **A**, contém todas as ações possíveis, que podem ser realizadas nos diferentes estados. O conjunto de probabilidades, relativamente às transações, **P**, consiste nas probabilidades de o agente se mover para um estado específico, após escolher uma certa ação, num certo estado atual. As recompensas, **R**, correspondem aos valores que o agente recebe após transitar de um estado atual para um estado seguinte. Por fim, o fator de desconto, γ, é um escalar compreendido entre $[0, 1]$, cuja função é averiguar a importância da recompensa imediata, comparativamente à recompensa acumulada pelo agente, [9].

A figura 3.1 exemplifica um MDP. Neste exemplo, o agente encontra-se no estado atual **E**, presente no nó superior, e pode escolher duas ações, sendo estas a ação **A1** ou a ação **A2**. Cada ação vai levar o agente para um estado seguinte diferente, sendo estes **E1**, **E2**, **E3** e **E4**, onde **P1**, **P2**, **P3** e **P4** correspondem às probabilidades de determinada ação levar o agente para determinado estado seguinte. Por fim, **r1**, **r2**, **r3** e **r4** correspondem às recompensas que o agente vai adquirir, caso escolha uma certa ação e vá para um determinado estado seguinte.

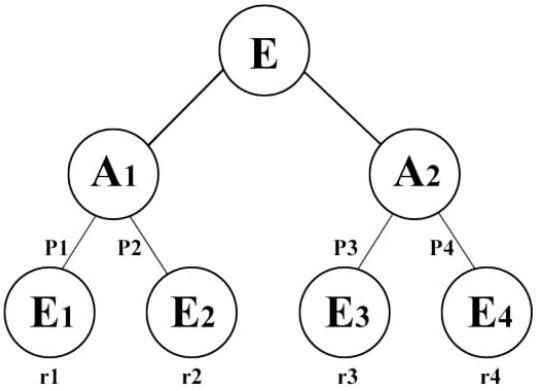


Figura 3.1: Exemplo de um MDP.

Como referido anteriormente, após o agente executar uma ação, é-lhe atribuída uma recompensa consoante o estado seguinte do ambiente. A recompensa acumulada ao longo de um episódio pode ser representada pela equação 3.1.

$$R_t = \sum_{k=0}^{T-t} \gamma^k r_t + k, t \in N \tag{3.1}$$

Nesta equação, é considerado que R_t consiste na recompensa que o agente recebe após escolher uma ação num dado estado. Por sua vez, **t** corresponde ao tempo em que aconteceu esta transição, enquanto **T** corresponde ao tempo em que o episódio termina.

Como referido na subsecção 3.1.1, o objetivo de um algoritmo de RL é encontrar uma política ótima para o objetivo de um dado ambiente. Esta política, π, pode ser definida através da equação 3.2.

$$\pi(a|s) = P[a_t = a|s_t = s], a \in A, s \in S \tag{3.2}$$

Em suma para que um ambiente seja considerado um MDP, este necessita de ser totalmente observável pelo agente em questão, bem como possuir certas caraterísticas como estados que são alterados pelas ações do agente, ações que o agente pode executar e recompensas por cada alteração no ambiente, originada por uma ação executada pelo agente.

3.1.1.2 Algoritmo *Q-Learning*

No algoritmo *Q-learning* é criada uma tabela, denominada de tabela *Q*. O objetivo desta tabela, ou matriz, é de armazenar os valores *Q*, valores estes que correspondem à probabilidade do agente escolher uma certa ação num dado estado, com o objetivo de maximizar uma política de recompensas. Uma das vantagens da utilização desta técnica, é o facto de não ser necessário criar previamente uma política otimizada, isto porque, o próprio agente possui a capacidade de melhorar a tabela *Q*, à medida que treina, e criar a sua própria política. Na figura 3.2, é possível observar-se o modo de funcionamento do algoritmo *Q-learning*.

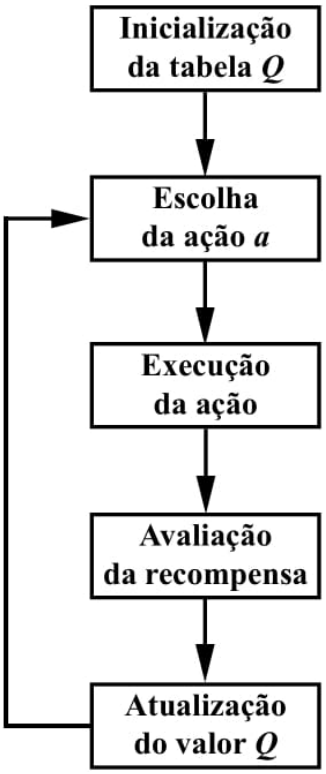


Figura 3.2: Esquemático acerca do funcionamento do algoritmo *Q-Learning*.

Na tabela *Q* as linhas correspondem aos possíveis estados do ambiente, enquanto as colunas correspondem às diferentes ações que o agente pode escolher, ou seja, uma tabela de tamanho **NxM**, corresponde a um ambiente com **N** estados, onde o agente pode realizar **M** ações. Inicialmente, todas as posições da tabela são inicializadas com o valor **zero**. Após o início do treino, quando o agente escolhe uma certa ação, num determinado estado, a posição da tabela correspondente é preenchida com um valor, denominado de valor *Q* e calculado através da equação de Bellman (equação 3.3). À medida que o agente explora o ambiente, a tabela é preenchida e o agente aprende qual a melhor ação num certo estado, consoante o valor *Q*, ou seja, para o estado **X** o agente escolhe a ação que corresponde ao maior valor *Q*.

$$Q'(s_t, a_t) = Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \tag{3.3}$$

Na equação 3.3, as presentes variáveis possuem o seguinte significado:

- $Q'(s_t, a_t)$, representa o valor *Q* a ser calculado,
- $Q(s_t, a_t)$, representa o valor *Q* atual,
- α, representa a taxa de aprendizagem,
- r_t, representa a recompensa dada ao agente por escolher uma ação num determinado estado,

- γ, representa a taxa de desconto,

- $\underset{a}{max}Q(s_{t+1}, a_{t+1})$, representa o valor máximo de recompensa futura que o agente pode receber, dado um estado futuro s_{t+1} e todas as ações possíveis para esse estado.

É de notar que os algoritmos que utilizam a equação de Bellman são algoritmos gananciosos, querendo isto dizer que vão realizar continuamente as ações que lhe garantem uma recompensa positiva, visto que, o seu objetivo é otimizar uma política de recompensas. Como tal a atribuição de recompensas positivas faz com que o agente repita essa ação o maior número de vezes possível, enquanto uma recompensa negativa faz com que o agente evite realizar essa ação, num dado estado. Com isto, a atribuição de recompensas negativas de baixo valor, é vantajoso caso se queira que o agente explore o ambiente de forma mais rápida possível. Como tal, a função recompensa atribuída a um agente, durante o seu treino, é extremamente importante, pois é devido a esta que o agente entende como cumprir a sua tarefa. Porém, se esta não for adequada, o agente pode apresentar um comportamento inesperado.

Atualmente, uma das desvantagens dos algoritmos de RL baseados em funções recompensas, é o facto de que o treino do agente depende de como as recompensas são atribuídas, ou seja, o agente pode apresentar um comportamento indesejado face ao ambiente, devido às recompensas que lhe são atribuídas, sendo que existem ambientes extremamente complexos onde é muito complicado o desenvolvimento de uma função recompensa adequada.

No início do treino, as posições da tabela Q possuem todas o valor **zero**, pelo que o agente não sabe qual ação escolher. Para ultrapassar este problema, é criado um rácio de exploração ϵ, cujo objetivo é representar a probabilidade do agente escolher uma ação aleatória. Inicialmente $\epsilon = 1$, **ponto A** da figura 3.3, visto que as posições da tabela Q encontram-se com o valor **zero**, ou seja, não se possui qualquer informação acerca do ambiente. À medida que o agente vai treinando, o valor de exploração ϵ vai diminuindo e o agente vai ganhando confiança na estimativa que faz relativamente aos valores Q. Por fim, quando ϵ chega a um valor mínimo, representado pelo **ponto B** da figura 3.3, previamente estipulado, o agente já sabe qual a melhor ação para cada estado, e como tal sai do estado de exploração aleatória, *exploration* em inglês, e entra no estado de exploração por si próprio, *exploitation* em inglês.

É agora apresentado um simples exemplo. Imaginemos que o nosso ambiente é uma linha de produção extremamente simples, representada na figura 3.4. Esta linha de produção possui as seguintes caraterísticas:

- produz um único produto,

- uma estação de entrada, representada pelo quadrado verde,

- uma estação de saída, representada pelo quadrado vermelho,

Figura 3.3: Gráfico da política *epsilon – greedy*.

- dois recursos **TA** e **TB**, representados pelos quadrados laranja e azul respetivamente.

Nesta linha de produção, a matéria-prima, representada pela caixa castanha, necessita de passar pelo recurso **TA**, de modo a produzir um produto e posteriormente deslocar-se para a estação de saída.

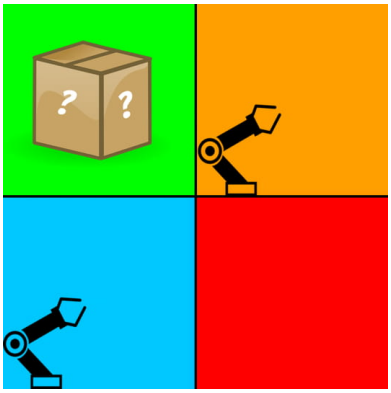


Figura 3.4: Exemplo de uma linha de produção simples, para a explicação do algoritmo *Q-Learning*.

Como referido anteriormente é necessário conceder ao agente *Q-learning* recompensas pelas suas ações, sendo estas as seguintes:

- 0,1 quando se deslocar para o recurso **TA**,
- -1 quando se deslocar para o recurso **TB**,
- -1 quando se deslocar para uma estação que já tenha estado previamente,
- 1 quando se deslocar para a estação de saída, proveniente do recurso **TA**.

Como é possível observar na figura 3.4, o produto possui quatro ações, tais como, mover-se para a **direita**, **esquerda**, **cima** ou **baixo**. Por sua vez, o ambiente possui quatro estados, sendo estes, a estação inicial, o recurso **TA**, o recurso **TB** e a estação final.

A execução do algoritmo *Q-Learning*, começa pela inicialização da tabela *Q*, sendo esta representada pela tabela 3.1.

Tabela 3.1: Inicialização da tabela *Q*.

	Cima	**Baixo**	**Direita**	**Esquerda**
Estação Inicial (EI)	0	0	0	0
Recurso **TA**	0	0	0	0
Recurso **TB**	0	0	0	0
Estação Final (EF)	0	0	0	0

O próximo passo do algoritmo passa pela escolha de uma ação por parte do agente. No entanto, como o agente se encontra na primeira iteração do seu treino, $\epsilon = 1$, o agente vai escolher uma ação aleatória, como por exemplo, **mover-se para a direita**. O resultado desta ação encontra-se presente na figura 3.5.

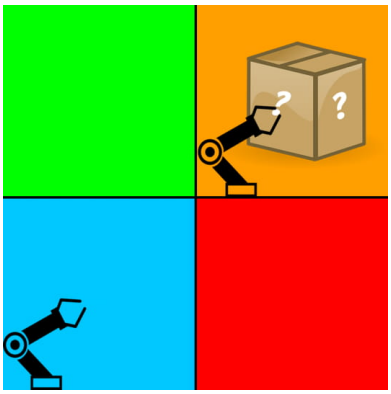


Figura 3.5: Primeira ação do agente no exemplo do algoritmo *Q-Learning*.

Como é possível observar na figura 3.5, o agente movimentou-se para o recurso **TA**, recebendo uma recompensa no valor de 0,01. Após receber a recompensa, o próximo passo do algoritmo é descobrir o novo valor Q, através da equação de Bellman (equação 3.3), considerando $\alpha = 0,1$ e $\gamma = 0,9$, **EI** como abreviatura para Estação Inicial, **dir.** para direita e **esq.** para esquerda.

$$Q'(EI, dir.) = Q(EI, dir.) + \alpha\,[R(EI, dir.) + \gamma max Q((TA, esq.), (TA, baixo)) - Q(EI, dir.)]$$
$$Q'(EI, dir.) = 0 + 0,1 * [0,1 + 0,9 * (0 - 0)] = 0,01$$

Observando a equação acima apresentada, é possível concluir que o valor Q, relativamente à ação **movimentar para a direita** na **estação inicial** é igual a 0,01.

Tabela 3.2: Alteração da tabela Q, após a primeira ação.

	Cima	**Baixo**	**Direita**	**Esquerda**
Estação Inicial	0	0	0,01	0
Recurso **TA**	0	0	0	0
Recurso **TB**	0	0	0	0
Estação Final	0	0	0	0

Os passos supracitados voltam-se a repetir até que o agente tenha encontrado uma política ótima e seja capaz de se movimentar no ambiente sem cometer erros.

A junção do algoritmo *Q-Learning* com uma simulação ou ambiente está representada na figura 3.6.

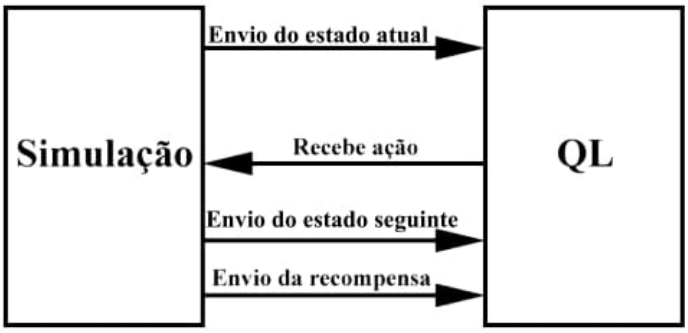


Figura 3.6: Representação da ligação entre uma simulação e o algoritmo *Q-learning*.

Apesar de ser benéfico para problemas simples, este algoritmo torna-se obsoleto em problemas mais complexos ou ambientes contínuos, visto que, nem sempre é possível deduzir todos os cenários possíveis num ambiente, como por exemplo num videojogo ou numa linha de produção complexa. Outra desvantagem deste algoritmo, é o facto de não ser versátil, ou seja, um algoritmo *Q-learning* treinado para um certo ambiente, não é flexível o suficiente para atuar sobre um ambiente diferente.

3.1.2 *Deep Reinforcement Learning*

Na sub-subsecção 3.1.1.2 são referidas algumas desvantagens do algoritmo *Q-learning*, como tal, para ultrapassar estas desvantagens [22] propuseram o algoritmo *Double-DQN*. Este algoritmo é semelhante ao anterior, no entanto os diferentes estados do ambiente não necessitam de ser previamente implementados, visto que, são utilizadas imagens da simulação, o que é benéfico para a resolução de problemas mais complexos, bem como atribui versatilidade ao algoritmo. Estas imagens por sua vez, são enviadas para uma *Convolutional Neural Network* (CNN), que vai armazenar os valores Q nos seus pesos[1], em vez de numa tabela. A equação de Bellman utilizada neste algoritmo (equação 3.4), sofre algumas alterações comparativamente à equação de Bellman (equação 3.3), utilizada na sub-subsecção 3.1.1.2. Tal como o algoritmo *Q-learning*, o algoritmo *Double-DQN* não necessita de conhecimento prévio do sistema, sendo que através de tentativa e erro, consegue encontrar a política ótima para o objetivo proposto, apresentando uma grande capacidade de generalização.

$$Q'(s_t, a_t) = r + \gamma \max_a Q(s_{t+1}, a_{t+1}) \tag{3.4}$$

3.1.2.1 *Double Deep Q-Network*

O algoritmo *Double-DQN* utiliza uma rede neuronal convolucional, para prever qual a melhor ação que o agente pode tomar num determinado estado de um ambiente. Geralmente o modelo de uma CNN é constituído por um grupo de camadas específicas, ordenadas por uma certa ordem, como por exemplo:

- camadas convolucionais, sendo que a primeira é o *input* da rede,
- entre cada camada convolucional encontra-se uma camada *pooling* e uma camada *activation*,
- camada *flatten*,
- camadas *dense*, sendo que a última é o *output* da rede,
- entre cada camada *dense* encontra-se uma camada *activation*.

No algoritmo *Double-DQN*, o *input* do modelo, ou seja, da primeira camada convolucional, consiste numa imagem do sistema, previamente processada de modo a diminuir o custo computacional. O tamanho do *output* da última camada *dense*, equivale ao número de ações que o agente pode realizar num determinado ambiente. O *output* corresponde a um vetor, cujo número de posições é igual ao número de ações que o agente pode escolher.

Na figura 3.7, é possível observar um esquema correspondente ao funcionamento do algoritmo *Double-DQN*.

[1] Os pesos de um rede neuronal, são o conjunto de valores alterados, pela CNN, ao longo do seu treino que permitem realizar uma estimativa cada vez melhor do valor a prever.

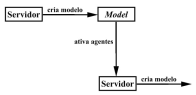


Figura 3.7: Esquemático acerca do funcionamento genérico do algoritmo *Double Deep Q-Network*.

No algoritmo *Double-DQN*, é utilizada uma rede alvo, [13]. O algoritmo DQN[2] tende em sobrestimar os valores Q correspondentes a uma ação, num dado estado. A superestimação de uma certa ação faz com que a probabilidade desta ser escolhida nos próximos estados vá aumentando, fazendo com que o agente não consiga explorar o ambiente de uma forma otimizada e encontrar a melhor política. Esta desvantagem pode ser ultrapassada, através da utilização de uma segunda rede, chamada **rede alvo**. A rede alvo é uma cópia da rede principal, porém só é atualizada de τ em τ iterações. Após esta alteração, na fase de treino, a rede principal prevê todos os valores Q para o estado atual, enquanto a rede alvo prevê todos os valores Q para o estado seguinte, sendo só utilizado o maior valor, correspondente à melhor ação, para o cálculo da equação de Bellman. Posteriormente, é utilizada a equação de Bellman, presente em 3.5, para calcular os novos valores Q.

[2]O algoritmo DQN é a versão do algoritmo *Double-DQN* que não utiliza uma rede alvo no seu treino.

$$Q'(s_t, a_t) = \gamma Q(s_{t+1}, (Q(s_{t+1}, a, \theta), \theta')) \tag{3.5}$$

Uma ferramenta fulcral para a implementação do algoritmo *Double-DQN* é a utilização de uma *replay memory*. A *replay memory* consiste numa memória finita, que guarda informações importantes para o treino da CNN, sendo estas:

- estado atual,
- ação escolhida pelo agente nesse estado,
- recompensa que o agente recebeu,
- estado seguinte,
- sinal cuja função é indicar se o agente realizou alguma ação incorreta, ou se chegou ao estado final. Este sinal vai influenciar o treino do agente, visto que, vai definir se o valor Q é calculado através da equação de Bellman, 3.5, ou se toma o valor da recompensa.

Esta memória é extremamente importante no treino da rede, porque permite com que esta aprenda através de experiências passadas, fazendo com que seja capaz de convergir. É importante referir que convém que esta memória seja grande o suficiente para armazenar todos, ou quase todos, os estados do treino, de modo a que a rede treine com estados mais recentes, bem como com estados mais antigos.

Tal como no algoritmo *Q-learning*, neste também é utilizada uma política $\epsilon - greedy$. Inicialmente o algoritmo é executado durante **X** iterações, com $\epsilon = 1$, onde o agente escolhe ações aleatórias, de modo a preencher uma parte da memória. Após isso, o valor de ϵ começa a decair a um ritmo definido, até chegar ao valor final de ϵ. Por fim, nas últimas iterações, tal como no algoritmo *Q-learning*, o agente entra num modo de exploração por si próprio e executa as ações provenientes da CNN.

A junção do algoritmo *Double-DQN* com ambas as simulações, encontra-se representada na figura 3.8.

Na implementação deste algoritmo, é necessário ter em mente o conceito de transferência de conhecimento, bem como duas fases fundamentais, sendo estas a fase de treino e a fase de testes. A fase de treino, como o próprio nome indica, consiste no estágio em que o algoritmo se encontra a ser treinado, ou seja, é quando este é colocado na simulação, sem conhecimento prévio sobre esta, e através de tentativa e erro encontra a política ótima para o objetivo proposto. Por sua vez, a fase de testes é importante para averiguar se o algoritmo aprendeu a política ótima de um determinado ambiente e encontra-se totalmente treinado. Por fim, o conceito de transferência de conhecimento consiste em treinar uma única instância do algoritmo e utilizar o conhecimento adquirido, para tornar todas as outras instâncias iguais, não treinadas, inteligentes. Sendo assim, na fase de testes os pesos da rede treinada são transferidos para as redes, presentes nas restantes instâncias do algoritmo, tornando-as inteligentes.

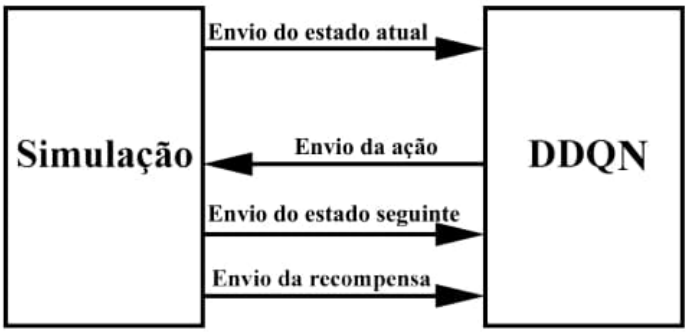


Figura 3.8: Representação da ligação entre uma simulação e o algoritmo *Double-DQN*.

3.1.3 Algoritmo *A**

O algoritmo heurístico *A** consiste num algoritmo cuja função é encontrar o caminho mais curto de um ponto **A** até um ponto **B**. Contrariamente a outros algoritmos heurísticos, este algoritmo utiliza uma equação **F** (equação 3.6), que permite escolher o melhor nó em cada iteração de procura. Esta equação utiliza um parâmetro heurístico **H** que calcula a distância entre o ponto atual e o ponto final, através do teorema de Pitágoras [3], diminuindo o tempo de procura, ao invés de calcular todos os caminhos possíveis desde a origem até ao ponto final e só depois escolher o melhor.

$$F = G + H \tag{3.6}$$

As variáveis presentes na equação 3.6 possuem o seguinte significado:

- **F**, representa o custo total de um nó,
- **G**, representa a distância entre o nó atual e o nó inicial,
- **H**, representa a heurística, sendo esta a distância estimada entre o nó atual e o nó final.

O algoritmo *A**, segue uma série de etapas até encontrar o caminho mais rápido para chegar ao ponto final. Este procedimento consiste no seguinte:

- adicionar o primeiro nó, ao conjunto de nós a serem testados,
- procurar o nó que possui o menor valor de **F** no conjunto de nós a serem testados, ou seja, o nó com menor custo,
- colocar o nó encontrado anteriormente, no conjunto dos melhores nós,

[3] Teorema de Pitágoras, consiste num teorema criado por Pitágoras, que permite calcular a hipotenusa de um triângulo retângulo, através da soma dos catetos ao quadrado.

- para cada posição adjacente, ignorar as posições onde não é permitido o movimento do agente, ou os nós que já se encontram no conjunto dos melhores nós,
- caso contrário, se o nó não se encontrar no conjunto dos nós a serem testados, adicionar e guardar os custos deste nó, nomeadamente os valores de **F**, **G** e **H**,
- se o nó já se encontrar no conjunto de nós a serem testados, averiguar se existe um nó que origina um caminho melhor, utilizando a variável **G** como custo,
- o processo termina quando o nó final é adicionado ao conjunto dos melhores nós, tendo encontrado o melhor caminho, ou não ser possível encontrar o nó final.

3.2 Simulação de uma *Job-Shop*

Uma linha de produção do tipo *job-shop* é constituída por produtos, transportadoras, recursos e pontos de decisão.

Na secção 2.5 do capítulo 2, [8] defende que uma *job-shop* cumpre os seguintes requisitos:

- cada recurso só pode realizar apenas uma operação de cada vez,
- o processamento de uma operação num recurso é chamado de operação,
- uma operação não pode ser interrompida,
- a sequência de recursos é desconhecida e tem de ser determinada, de modo a diminuir o tempo de produção de um produto.

3.2.1 Sistema Multi-Agente

Um Sistema Multi-Agente (SMA) é composto por diferentes agentes inteligentes, que interagem entre si. Estes agentes são entidades fulcrais na implementação dos algoritmos, visto que, estes algoritmos é que vão atribuir caraterísticas como tomada de decisão, inteligência e dinamismo aos agentes.

3.2.1.1 Agentes

Os agentes são entidades importantes, visto que são estes que são controlados pelos algoritmos inteligentes. Estes agentes podem representar todos os componentes presentes numa linha de produção do tipo *job-shop*, sendo estes:

- produtos,
- transportadoras,
- recursos,

- pontos de decisão.

Como um dos objetivos do algoritmo proposto é encaminhar os produtos pela linha de produção, é mais vantajoso se este controlar os agentes que permitem o produto mudar de direção, como tal este algoritmo deve ser implementado nos agentes produto, transportadoras ou pontos de decisão, dependendo da linha e simulação em questão.

Podem existir vários tipos de produtos numa determinada linha de produção, sendo assim necessário encaminhá-los consoante a sua necessidade. É proposto o exemplo de uma simulação que consegue produzir no máximo **dezasseis** produtos simultaneamente, sendo que existem **quatro** tipos de produtos distintos. Neste exemplo, o algoritmo inteligente é instanciado nos agentes produto, sendo necessário criar e treinar quatro algoritmos para cada tipo de produto, porém é necessário um grande poder computacional para treinar simultaneamente dezasseis algoritmos. Para ultrapassar esta desvantagem, é proposto que, na fase de treino, seja criada uma instância do algoritmo para um agente produto de cada tipo, diminuindo o número de algoritmos a treinar simultaneamente, de dezasseis para quatro, e que os restantes agentes produto tenham um encaminhamento pré-definido.

Após a fase de treino, são instanciados os restantes algoritmos, de modo a tornarem inteligentes os agentes com encaminhamento pré-definido. Para tal, transfere-se o conhecimento obtido pelos algoritmos treinados para todos os outros, originando, numa fase de testes, dezasseis agentes produto inteligentes, só necessitando de treinar quatro algoritmos simultaneamente.

3.3 Desenho da Solução

Uma solução, visando a junção de algoritmos inteligentes, como o referido nesta dissertação, com um sistema de produção real, consiste na utilização de um CPPS. Como referido na secção 2.2, um CPPS, de uma forma genérica, consiste num sistema, utilizado em manufatura, que engloba entidades físicas e entidades virtuais, visando a sua interação e comunicação, através de uma interface de comunicação. Sendo assim, é proposta a criação de uma simulação, cujo objetivo é servir de *digital twin* de uma linha de produção real, permitindo a utilização de algoritmos inteligentes em ambientes reais, de modo a que não seja necessário parar a sua produção, fazendo com que não existam alterações ou prejuízos.

Após treinar, testar e validar o algoritmo, é possível colocá-lo a controlar o cenário real através de um CPPS, como supracitado. Os sensores presentes na entidade física originam informação acerca dos diferentes componentes e através da interface de comunicação esta é enviada para a entidade virtual, simulação. A simulação fornece a informação ao algoritmo inteligente, previamente treinado, de modo a que este tome uma decisão. Por sua vez, o algoritmo envia esta decisão para o SMA, que através da interface de comunicação, envia para a entidade física. Esta, através de atuadores controla os diferentes

componentes físicos. Na figura 3.9, encontra-se um esquema que exemplifica a situação descrita.

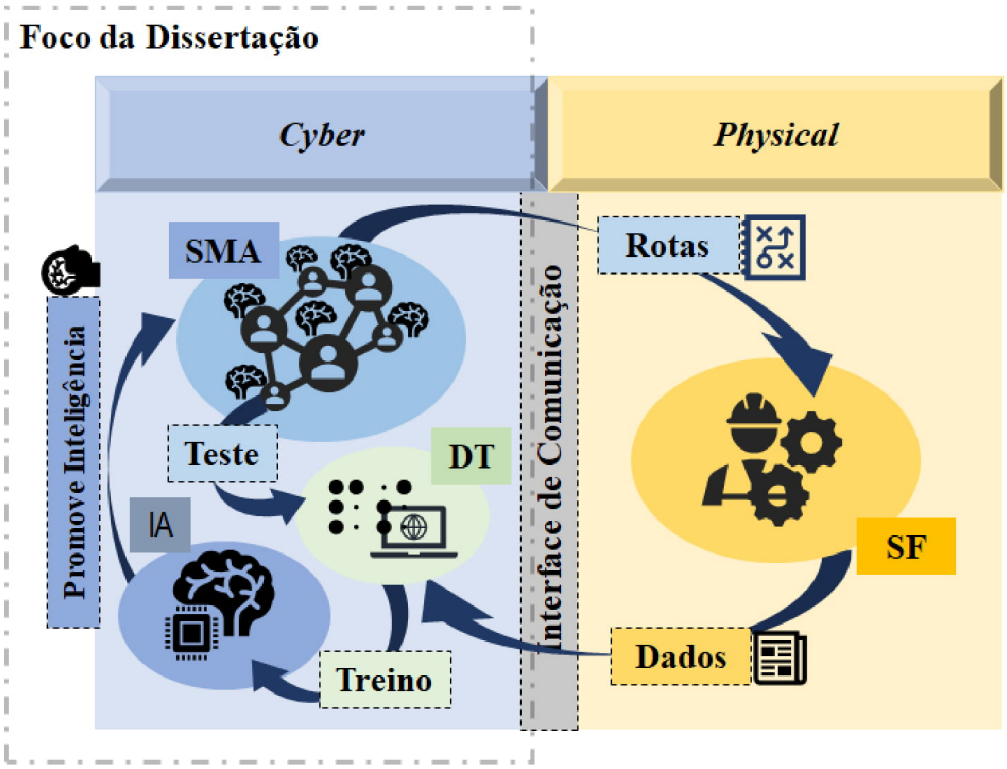


Figura 3.9: Esquemático representativo do CPPS proposto.

As siglas presentes na figura 3.9 possuem o seguinte significado:

- **SMA** significa **sistema multiagente**,
- **IA** significa **inteligência artificial**,
- **DT** significa ***digital twin***,
- **SF** significa **Sistema Físico (SF)**.

A simulação desenvolvida necessita de apresentar as características necessárias para implementar um algoritmo de RL, sendo estas ser totalmente observável, apresentar um espaço de ações, um espaço de estados e uma função recompensa. Para além disto, esta também necessita de ser discreta para que em cada passo o agente consiga escolher uma ação e posteriormente, ser adquirido o estado atual, o estado seguinte e a recompensa, de modo a treinar o agente.

Capítulo 4

Implementação

A implementação das simulações propostas nesta dissertação, foi realizada utilizando a linguagem de programação *Python*, recorrendo à *framework* de simulação baseado em agentes, *Mesa*, [44]. De notar que todas as listagens referidas neste capítulo encontram-se em anexo.

4.1 Ambiente de Simulação *Mesa*

O ambiente de simulação *Mesa* consiste numa *framework*, baseada na linguagem de programação *Python*, que permite a criação de modelos baseados em agentes, utilizando componentes pré-programados, onde é possível simular a interação de vários agentes e testar vários cenários, como o cenário das ovelhas e dos lobos, ou até recriar a interação de células cancerígenas e analisar o comportamento de tumores, [43]. Através das ferramentas de análise de dados da linguagem de programação *Python*, é possível analisar os resultados obtidos, sendo também possível visualizar a simulação e os seus agentes através de uma interface *web browser*, [44].

A linguagem de programação *Python* possui bibliotecas de criação de modelos de IA muito úteis, bem como bibliotecas pertinentes para manipulação de dados. Sendo assim, foi escolhido este ambiente de simulação, por apresentar um baixo custo computacional, bem como ser baseado na linguagem de programação *Python*.

A *framework Mesa* apresenta uma ordem de execução específica, presente na figura 4.2, sendo esta extremamente importante para o desenvolvimento dos algoritmos *Q-learning* e *Double-DQN*, neste ambiente.

Primeiramente o modelo, *model* em inglês, os agentes, *agents* em inglês, e o servidor, *server* em inglês, são criados, originando um *URL*[1] que permite aceder ao ambiente de simulação através de um *web browser* (figura 4.1).

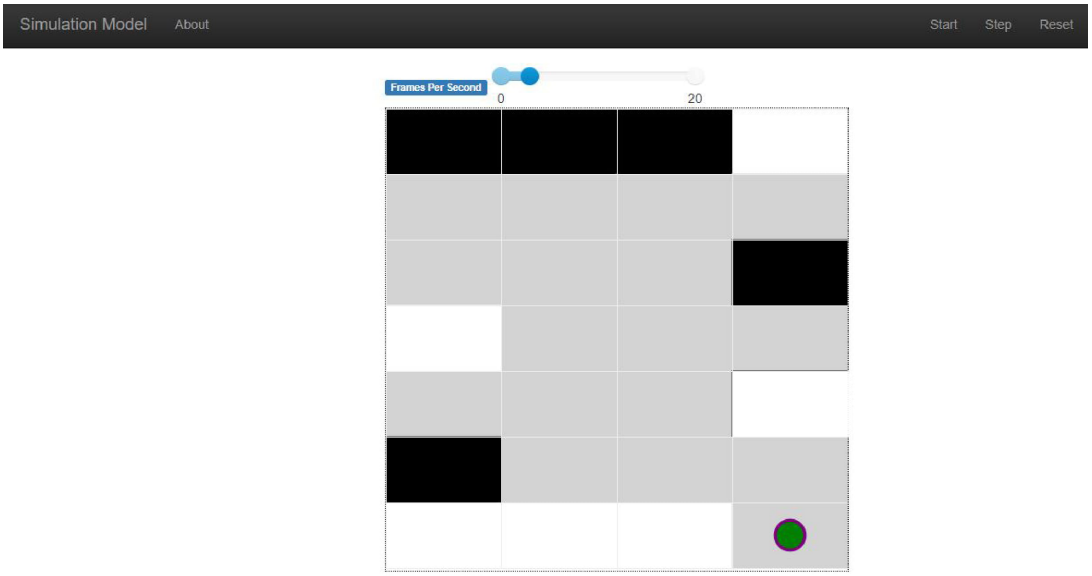


Figura 4.1: Exemplo da simulação 2 no *browser*.

Em seguida, quando é pressionado o botão *Start*, presente na figura 4.1, o método *step* da classe *Model* é executado, onde são ativados os respetivos agentes e executados os respetivos métodos *step* de cada um, onde se encontra a ação que cada agente necessita de executar. Posteriormente, a componente *ModularVisualization* altera, visualmente, o ambiente de simulação presente no *web browser* e o método *step* da classe *Model* é executado novamente. Este procedimento é repetido até que a variável *running*, proveniente da classe *Model*, seja **falsa**, ou que o utilizador pressione o botão *Stop*, presente no *web browser* (figura 4.1).

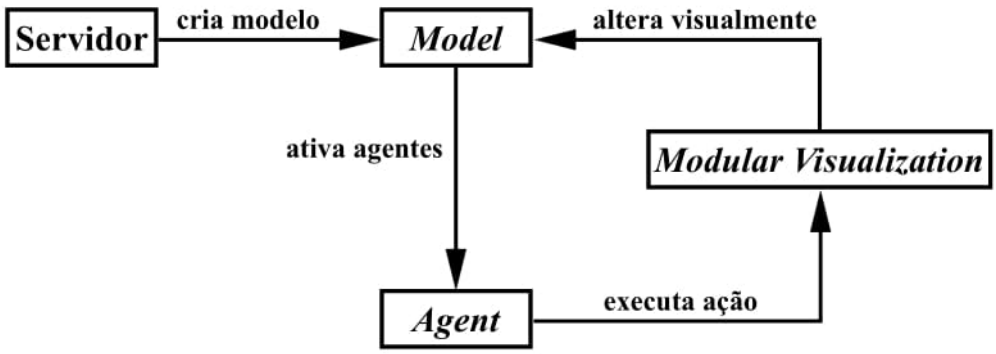


Figura 4.2: Fluxograma referente ao funcionamento da *framework Mesa*.

[1] *Uniform Resource Locator (URL)*, ou *Uniform Resource Locator*, é o endereço que permite aceder a qualquer *webpage*, servidor local, entre outros, através de um *web browser*.

A *framework* é constituída por três módulos principais, sendo estes, o **módulo de modelação**, utilizado para desenvolver tanto o ambiente como os agentes, o **módulo de análise**, que faculta ferramentas para armazenar os dados provenientes da simulação, bem como executar diferentes simulações em paralelo com diferentes parâmetros e o **módulo de visualização**, que permite visualizar o ambiente através de um servidor com interface baseada na linguagem *javascript*, [44]. Nesta dissertação são utilizados os **módulos de modelação** e **visualização**, visto que, os dados provenientes da simulação foram armazenados ao longo das iterações de treino em ficheiros *.pkl* e *.csv*, permitindo interromper o treino sempre que necessário e voltar a executá-lo a partir da última iteração, sem perder informação.

4.1.1 Módulo de Modelação

O módulo de modelação é constituído por duas classes, sendo estas a classe *Model* e a classe *Agent* e dois sub-módulos, sendo estes, o sub-módulo *Time* e o sub-módulo *Space*. A classe *Model* é utilizada para desenvolver o modelo da simulação orquestrando toda a simulação. Nesta classe são inicializados as seguintes componentes:

- a **componente temporal**, que define a ordem pelo qual os agentes são chamados para realizarem a sua ação,
- a **componente espacial**, que define o tipo de espaço em que os agentes vão estar presentes, bem como o seu tamanho,
- a **componente dos agentes**, onde estes são inicializados, ordenados para serem executados e posicionados numa posição específica do ambiente.

Ambas as classes, *Model* e *Agent*, possuem o método *step*. O método *step* da classe *Model* tem como função avançar a componente temporal por um passo, enquanto o método *step* de cada agente, faz com que este realize a sua ação. No código presente na listagem I.1, encontra-se um código exemplo onde o agente escreve a frase *Agent Stop* no ecrã. Para tal, a classe *Model* avança um passo na simulação, através do método *step*, este por sua vez chama o método *step* do agente, que neste caso, escreve a frase *"Agent Stop"*.

O sub-módulo *Time*, permite definir e agendar a ordem de execução dos agentes, existindo as seguintes alternativas:

- *BaseScheduler*, consiste na forma mais simples de agendamento de agentes, sendo que a ordem de ativação dos agentes é baseada na ordem em que os agentes são adicionados à componente temporal, ativando um agente de cada vez,
- *RandomActivation*, é semelhante ao *BaseScheduler*, porém a ordem de ativação dos agentes é aleatória,
- *SimultaneousActivation*, consiste num agendamento que ativa todos os agentes em simultâneo,

- *StagedActivation*, consiste num agendamento que permite dividir a ativação dos agentes em vários estágios.

Por sua vez, o sub-módulo *Space*, permite adicionar uma componente espacial ao modelo, como por exemplo uma grelha, ou *grid* em inglês. A *framework Mesa* faculta diferentes ambientes, como por exemplo:

- *ContinuousSpace*, consiste num espaço contínuo onde cada agente possui uma posição arbitrária,
- *Grid*, consiste numa grelha quadrada,
- *HexGrid*, consiste numa grelha hexagonal,
- *MultiGrid*, consiste numa grelha onde cada célula pode conter mais do que um agente. Este espaço é um caso particular da *Grid*.
- *NetworkGrid*, consiste numa grelha em rede, onde cada nó contém zero ou mais agentes,
- *SingleGrid*, consiste numa grelha quadrada, onde cada célula contém no máximo um agente. Consiste num caso particular da *Grid*.

É importante referir que cada opção dos sub-módulos *Time* e *Space*, possuem as suas próprias caraterísticas, bem como os seus próprios métodos. Com isto, a escolha da melhor opção dos sub-módulos *Time* e *Space*, necessita de ser feita com base no objetivo da simulação, bem como os métodos que cada opção oferece.

4.1.2 Módulo de Visualização

O módulo de visualização permite observar o ambiente de simulação, bem como as interações entre os agentes, através de um servidor *web* local, utilizando a linguagem de programação *JavaScript*. Este módulo possui duas classes importantes, sendo estas a classe *ModularVisualization* e a classe *Modules*. A primeira classe, permite criar e executar o servidor *web* local, enquanto a segunda permite personalizar, o ambiente de simulação através de uma *CanvasGrid*.

Para além dos motivos presentes na secção 4.1, esta *framework* também foi escolhida, pois apresenta a opção de aumentar a velocidade da simulação (figura 4.3) possibilitando incrementar a velocidade até 20 *Frames Per Second* (FPS)[2]. Esta caraterística é uma vantagem, visto que, permite acelerar a simulação e diminuir o tempo de treino dos agentes. Todas as informações supracitadas encontram-se em [44].

[2]FPS, ou *frames per second* em inglês, consiste no número de imagens passadas durante um segundo.

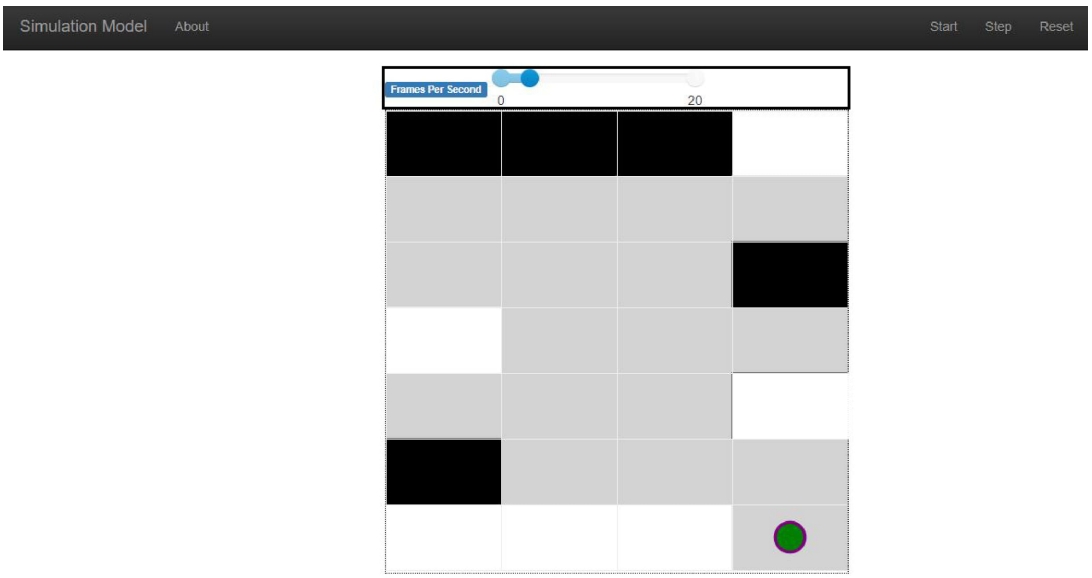


Figura 4.3: *Slider* que permite alterar a velocidade da simulação.

4.2 Simulações

Nesta dissertação são apresentadas duas simulações, sendo estas a **simulação 1** e a **simulação 2**. Estas simulações, representam uma linha de produção do tipo *job-shop*. Ambas as simulações foram desenvolvidas utilizando a *framework Mesa* , são baseadas num ambiente multi-agente e compostas por dez tipos de agentes.

Estes agentes têm o objetivo de representar as transportadoras, a estação inicial, a estação final, os diferentes recursos, as gruas, os pontos de decisão e os diferentes tipos de produtos, de uma linha de produção. Devido ao funcionamento da *framework* utilizada, é necessário criar dois grupos de agentes, os **Agentes Produto**, ou **Agentes Ativos**, e os **Agentes Inativos**.

Os **Agentes Produto**, ou **Agentes Ativos**, consistem nos agentes que se movimentam na simulação e são controlados pelos três algoritmos referidos nesta dissertação, de modo a que lhes seja atribuída inteligência individual. Contrariamente ao que acontece num cenário real, devido à *framework* utilizada, estes agentes representam os diferentes produtos desenvolvidos numa linha de produção, e não as transportadoras, ou os pontos de decisão. Estes agentes fornecem informações essenciais, sendo estas:

- nome/ID,
- quantidade de tempo que demorou a ser produzido, em pontos,
- posição.

Por sua vez, **Agentes Inativos** são todos aqueles que não sofrem alterações durante a execução do sistema, ou seja, que se encontram **estacionários**. A principal função destes agentes é representarem diferentes componentes presentes numa linha de produção, de

modo a tornar as simulações realistas. Para além desta função, estes também facultam informações essenciais ao funcionamento da simulação, como a sua posição ou o tempo necessário para a manipulação de matéria-prima em cada recurso.

Um dos objetivos desta dissertação é demonstrar que ao aplicar algoritmos inteligentes em ambientes de produção, é possível encontrar o caminho mais rápido para a produção de vários produtos. Sendo assim, é necessário desenvolver uma métrica capaz de contabilizar o tempo que cada produto demora a ser produzido. Esta métrica consiste na quantidade de pontos que os **Agentes Produto** acumulam até chegarem à estação de saída, ou seja, por cada iteração que cada **Agente Produto** se encontra na simulação é considerado um ponto, sendo que o agente recebe pontos em três situações distintas, sendo estas:

- ao movimentar-se,
- ao ficar parado à espera que a próxima estação se encontre livre,
- ao se encontrar em cada um dos recursos presentes na linha de produção.

Para tornar as simulações mais realistas, estes pontos podem ser convertidos em unidades de tempo, como por exemplo segundos, onde se pode considerar **1 ponto = 1 segundo**. Como tal, é possível concluir quanto tempo demorou um certo tipo de produto a ser produzido. De notar que na **simulação 1**, o menor tempo possível que qualquer agente demora a ser produzido é de **53 pontos**, ou **53 segundos**.

4.2.1 Simulação 1

Nesta subsecção, numa primeira parte, é explicado o modo de funcionamento da **simulação 1**. Após isso, é apresentada a implementação da mesma, bem como toda a lógica de baixo nível relativamente aos agentes.

O propósito desta simulação, é demonstrar que o algoritmo *Double-DQN* é capaz de definir o encaminhamento ótimo de produtos tendo em conta os restantes produtos presentes no ambiente de produção, visando uma produção no menor tempo possível. Para tal, este algoritmo é comparado, através de cenários de teste e métricas, com outros dois algoritmos, sendo estes o algoritmo *Q-learning* e o algoritmo heurístico *A**.

4.2.1.1 Agentes Produto

A **simulação 1** tem a capacidade de produzir três tipos de produtos diferentes, onde cada tipo de produto corresponde a um algoritmo, ou seja:

- **Agente Produto Tipo 1** corresponde ao algoritmo *Double-DQN*,
- **Agente Produto Tipo 2** corresponde ao algoritmo *Q-learning*,
- **Agente Produto Tipo 3** corresponde ao algoritmo heurístico *A**.

Nesta simulação, existem quatro tipos de **Agentes Produto**, presentes na tabela 4.1, isto porque o quarto tipo de agente é unicamente utilizado na fase de treino dos algoritmos *Double-DQN* e *Q-learning*, sendo que nesta fase ambos os algoritmos vão controlar este agente, no seu respetivo treino. O **Agente Produto Tipo 4** tem como objetivo representar um agente de qualquer um dos três tipos, possuindo uma cor diferente, para que seja possível identificá-lo durante os treinos. Como tal, durante os treinos dos dois algoritmos referidos, encontram-se dois **Agentes Produto** do **Tipo 1** ou do **Tipo 2** e um **Agente Produto Tipo 4**.

Na fase de testes da **simulação 1**, o **Agente Produto Tipo 4** não é utilizado, sendo que cada algoritmo vai estar responsável pelo movimento dos restantes **Agentes Produto**, como por exemplo, o algoritmo *Double-DQN* vai estar responsável pelo movimento dos **Agentes Produto Tipo 1**.

Tabela 4.1: Agentes Produto e as suas caraterísticas para a simulação 1.

Tipo	Nome	Quantidade	Cor	Algoritmo
Agentes Produto Tipo 1	$PA1_{0\ldots2}$	3	Verde	*Double-DQN*
Agentes Produto Tipo 2	$PA2_{0\ldots2}$	3	Cinzento Escuro	*Q-learning*
Agentes Produto Tipo 3	$PA3_{0\ldots2}$	3	Preto	A^*
Agente Produto Tipo 4	$PA4_0$	1	Múltiplas	

Cada **Agente Produto** possui uma ordem de recursos, previamente estipulada, pelo qual necessita de passar, presente na tabela 4.2.

Tabela 4.2: Agentes Produto e a sua ordem de produção para a simulação 1.

Tipo	Ordem
Agentes Produto Tipo 1	RA, RC, RB
Agentes Produto Tipo 2	RB, RA, RC
Agentes Produto Tipo 3	RA, RB, RC

4.2.1.2 Agentes Inativos

Na tabela 4.3 encontram-se os **Agentes Inativos** presentes na **simulação 1**, bem como as suas caraterísticas.

Tabela 4.3: Agentes Inativos presentes na simulação 1 e as suas caraterísticas.

Tipo	Nome	Quantidade	Cor
Agentes Transportadora	$CA_{0\ldots64}$	65	Cinzento Claro
Agente Estação Inicial	$Begin_0$	1	Verde Escuro
Agente Estação Final	End_0	1	Azul Bebé
Agentes Recurso Tipo A	$RA_{0\ldots5}$	6	Vermelho
Agentes Recurso Tipo B	$RB_{0\ldots5}$	6	Azul
Agentes Recurso Tipo C	$RC_{0\ldots5}$	6	Azul Escuro

Os recursos da **simulação 1** têm como função executar as operaçãos presentes na tabela 4.4. É de notar que na **simulação 1**, cada recurso só tem a capacidade de executar uma única operação.

Tabela 4.4: Operações executadas pelos recursos da simulação 1.

Tipo	Operação	Tempo de Produção
Agentes Recurso Tipo A	Operação A	10 segundos
Agentes Recurso Tipo B	Operação B	5 segundos
Agentes Recurso Tipo C	Operação C	20 segundos

4.2.1.3 Funcionamento

A **simulação 1**, presente na figura 4.4, consiste numa grelha de tamanho **11x13**, cuja função é representar uma linha de produção constituída por **nove** transportadoras, **dezoito** recursos e **cinco** pontos de decisão. Esta linha de produção tem a capacidade de produzir **três** tipos de produtos diferentes, sendo possível produzir três produtos de cada tipo simultaneamente. Cada tipo de produto, tem de passar obrigatoriamente por **três** recursos, de forma ordenada, antes de sair da linha de produção. Nesta simulação os **Agentes Produto** podem movimentar-se em três direções diferentes, sendo estas, **direita**, **esquerda** e **baixo**.

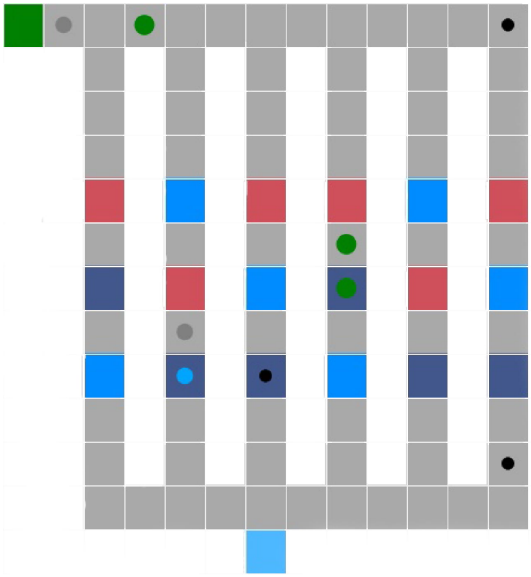


Figura 4.4: Simulação 1.

As transportadoras estão organizadas em paralelo, existindo dois caminhos possíveis para cada tipo de produto. Estas encontram-se presentes, inicialmente, numa configuração horizontal (figura 4.5), permitindo que os produtos se desloquem até aos diferentes pontos de decisão.

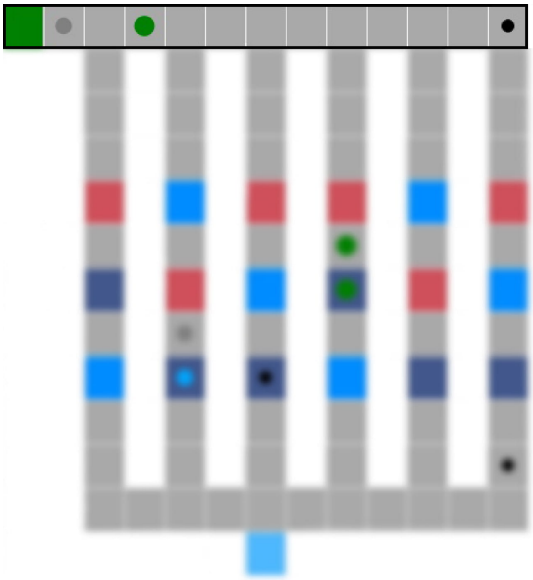


Figura 4.5: Transportadora horizontal superior, presente na simulação 1.

Nos pontos de decisão, presentes na figura 4.6, é possível decidir se os produtos são encaminhados para a primeira transportadora vertical do seu tipo, ou se continuam na transportadora horizontal até chegarem à próxima transportadora vertical do seu tipo. É nos pontos de decisão presentes na primeira transportadora horizontal (figura 4.5), que o agente, a ser treinado pelos diferentes algoritmos, toma a decisão consoante cada cenário.

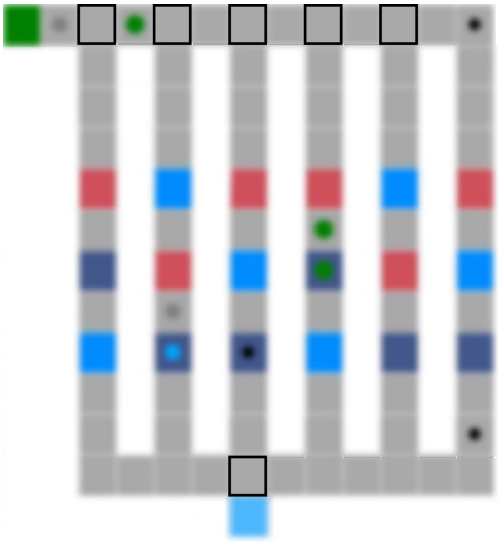


Figura 4.6: Pontos de decisão, presentes na simulação 1.

Após a decisão, o produto em questão é deslocado para uma das transportadoras verticais (figura 4.7), onde passa pelos recursos necessários.

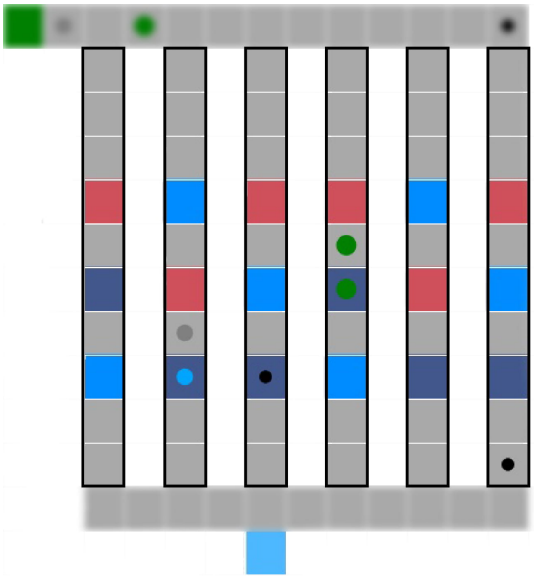


Figura 4.7: Transportadoras verticais, presentes na simulação 1.

Como é possível observar na figura 4.7, existem seis transportadoras verticais. Estas foram criadas, de modo a existirem duas transportadoras para cada tipo de produto, com o objetivo de criar pontos de decisão onde os algoritmos têm a possibilidade de tomar decisões. As transportadoras correspondentes ao **Agente Produto Tipo 1** encontram-se na figura 4.8, enquanto as transportadoras correspondentes ao **Agente Produto Tipo 2** encontram-se na figura 4.9 e por fim as correspondentes ao **Agente Produto Tipo 3** encontram-se na figura 4.10.

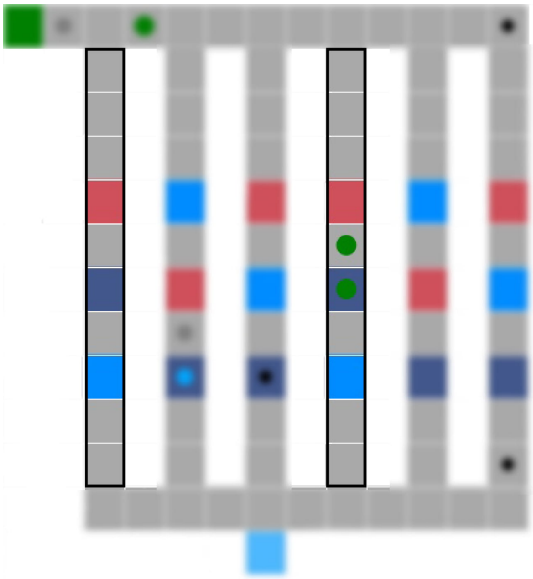


Figura 4.8: Transportadoras verticais para o Agente Produto Tipo 1, presente na simulação 1.

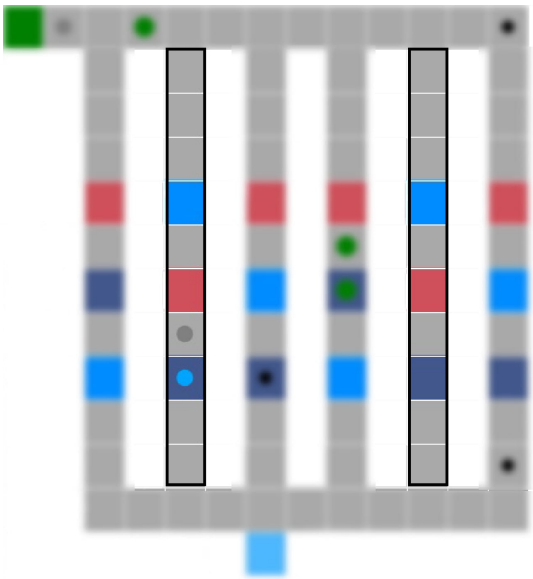


Figura 4.9: Transportadoras verticais para o Agente Produto Tipo 2, presente na simulação 1.

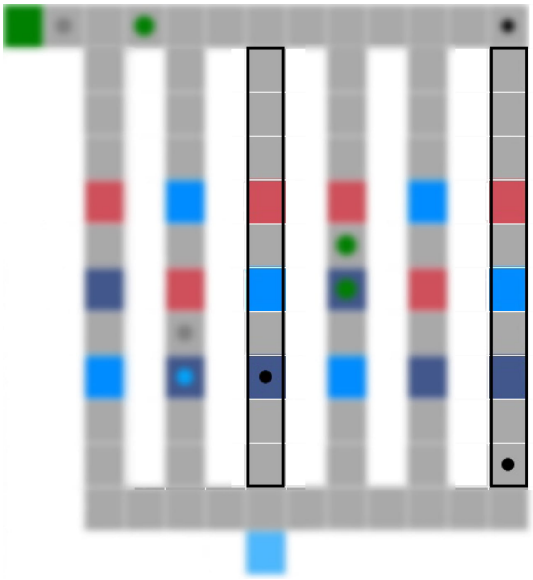
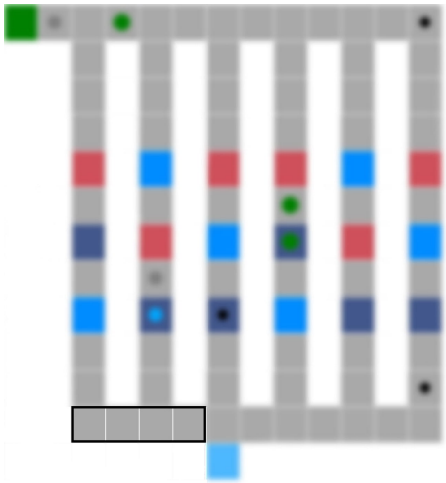
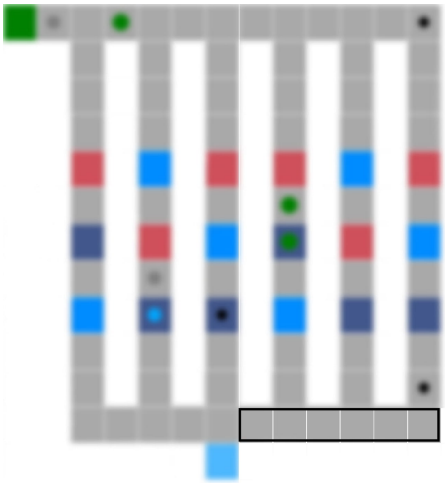


Figura 4.10: Transportadoras verticais para o Agente Produto Tipo 3, presente na simulação 1.

Por fim, os produtos são enviados para uma das transportadoras horizontais inferiores, figura 4.11a e figura 4.11b, de modo a finalizar um ciclo de produção.



a Transportadora horizontal inferior do lado esquerdo, presente na simulação 1.



b Transportadora horizontal inferior do lado direito, presente na simulação 1.

Figura 4.11: Transportadoras horizontais inferiores, presentes na simulação 1.

Como referido anteriormente, cada tipo de produto tem de passar por uma ordem específica de recursos, presente na tabela 4.2. Os recursos encontram-se ordenados nas transportadoras verticais presentes na simulação (figura 4.12).

Nesta simulação, o menor número de pontos possível é **53 pontos**, pelo que acontecem quando qualquer **Agente Produto** escolhe a primeira transportadora vertical do seu tipo e esta se encontra vazia.

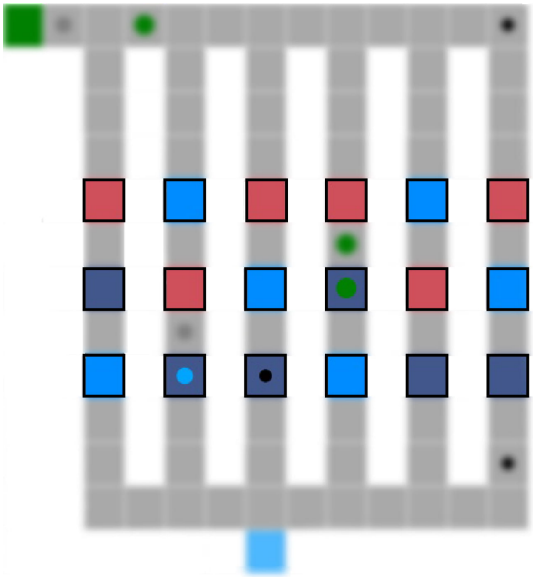


Figura 4.12: Diferentes tipos de recursos, presentes na simulação 1.

De notar que cada transportadora só possui uma única direção de funcionamento, estando estas presentes na figura 4.13 em forma de setas.

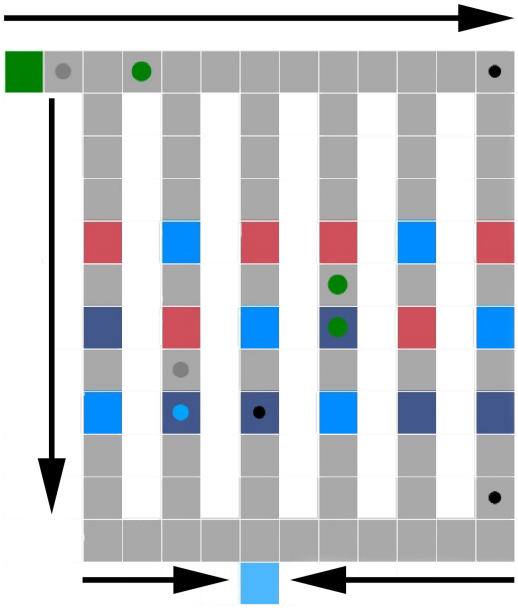


Figura 4.13: Direções de funcionamento das transportadoras, presentes na simulação 1.

4.2.1.4 Implementação e Lógica

Nesta simulação, é utilizada a classe *RandomActivation*, do sub-módulo *Time* presente na *framework Mesa* , tanto na fase de treino, como na fase de testes, de modo a tornar a

simulação mais dinâmica, visto que, esta classe permite a ativação aleatória dos agentes. Por sua vez, é utilizada a classe *MultiGrid*, proveniente do sub-módulo *Space*, visto ser a classe mais adequada para esta simulação, isto porque, permite mais do que um agente em cada célula e possui métodos extremamente úteis para, de uma forma rápida, retornar informações acerca do ambiente, como por exemplo a vizinhança [3] dos agentes. O conceito **vizinhança** é importante no desenvolvimento das duas simulações, pois permite averiguar se a posição seguinte, neste caso transportadora, ponto de decisão ou recurso, se encontra ocupada por outro agente, ou não, facilitando assim o desenvolvimento da lógica correspondente à interação dos agentes.

Primeiramente, foi implementada a classe *server*, ou seja, o servidor. Nesta classe encontra-se o método *agent_portrayal*, cujo excerto se encontra na listagem I.3 (em anexo), responsável pela implementação das caraterísticas visuais dos agentes presentes na **simulação 1**. De notar que na listagem I.3, o agente *ProductAgent1* corresponde ao **Agente Produto Tipo 1**, o agente *ConveyorAgent* corresponde aos **Agentes Transportadora**, o agente *ResourceAgentA* corresponde ao **Agente Recurso Tipo A** e por fim o agente *BeginAgent* corresponde ao **Agente Estação Inicial**.

Em seguida foram desenvolvidos os métodos referentes à instanciação dos agentes, onde são adicionados à lista de agendamento e colocados nas suas posições, formando o ambiente de simulação. Antes do desenvolvimento da simulação, foi realizada uma reflexão acerca da sua estrutura, visto ser imprescindível estipular as posições corretas dos agentes.

Após a instanciação dos agentes, foi criada a classe *Model*, denominada de *SimulationModel*, possuindo os seguintes atributos:

- *width*, ou seja, o comprimento da grelha,
- *height*, ou seja, a altura da grelha,
- *cnn_model* e *model_target*, ou seja, a rede principal e a rede alvo, no caso do algoritmo *Double-DQN*.

Foram também criadas as variáveis **ra_pos_list_A**, **ra_pos_list_B** e **ra_pos_list_C**, de modo a armazenar as posições dos **Agentes Recurso** para serem utilizadas no algoritmo de movimentação dos agentes.

Relativamente à interação e movimentação dos agentes, foi desenvolvido um método, denominado de *stop_move*, que permite com que os **Agentes Produto** fiquem parados nos **Agentes Recurso** durante um tempo previamente estipulado. O estado natural dos agentes na *framework Mesa* é estar parado, como tal foi preciso criar uma variável que define o tempo que o agente tem de estar parado. Esta variável é decrementada em cada *step* da *framework* até atingir o valor zero, sendo que enquanto for diferente de zero o

[3] É considerada a vizinhança do agente as células adjacentes a este. Com este conceito é possível desenvolver comportamentos, para os agentes, baseados nos agentes que se encontram ao seu redor.

método *stop_move* retorna o valor *true*, ou verdadeiro em português, fazendo com que o agente não se mova devido ao operador condicional *if* presente no seu método *step*. Caso contrário, o método *stop_move* retorna o valor lógico *false*, ou falso em português, tornando o operador condicional *if* verdadeiro e permitindo com que o agente se movimente para a próxima posição. A figura 4.14 representa um fluxograma acerca do funcionamento do método *stop_move*, presente na listagem I.7 (em anexo).

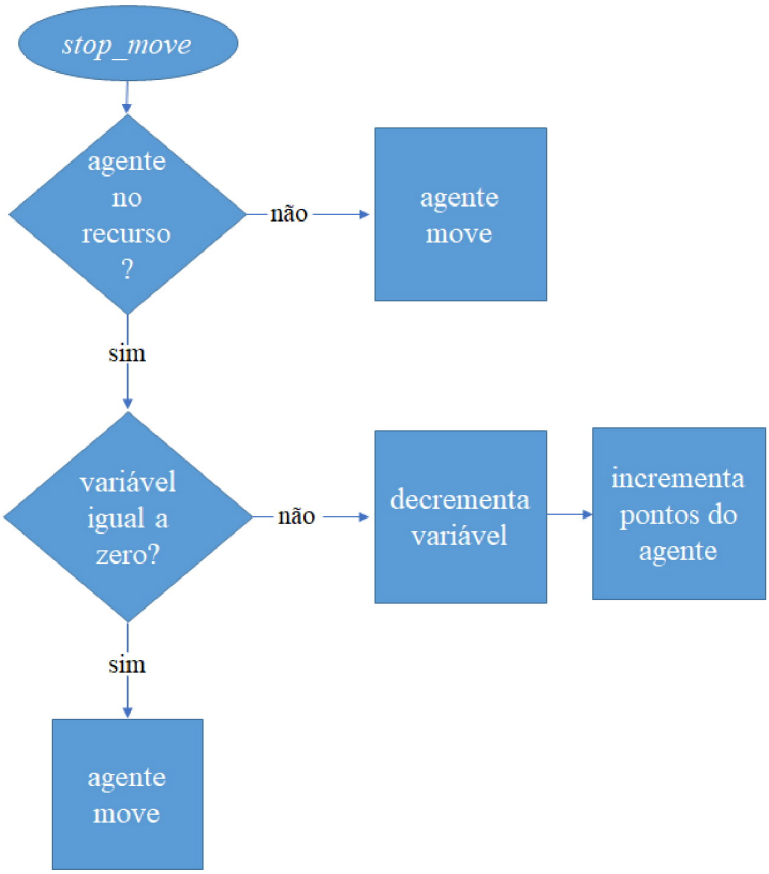


Figura 4.14: Fluxograma referente ao funcionamento do método *stop_move* da simulação 1.

Em seguida, foram implementados os métodos que permitem com que o agente se mova. Nestes, é guardada a posição atual do agente em questão, sendo posteriormente incrementadas ou decrementadas, consoante o movimento pretendido, as coordenadas **X** ou **Y** correspondentes à sua posição. O método que permite o movimento aleatório, no primeiro ponto de decisão de cada tipo, é semelhante ao método mencionado, no entanto a escolha de o agente se mover para baixo ou para a direita é realizada aleatoriamente através da biblioteca *random* e do operador condicional *if*.

Posteriormente foram desenvolvidos os métodos que averiguam se a posição, para o qual o agente se quer mover, se encontra ocupada por um outro **Agente Produto**, ou não, sendo utilizado o método *get_neighbors*, fornecido pela classe *MultiGrid*. Este método retorna todos os agentes que se encontram na vizinhança do **Agente Produto** em

questão na forma de uma lista, consoante certos parâmetros, como por exemplo o raio de procura e se a posição atual é contada ou não. Se o tamanho da lista retornada for igual a um, significa que a posição para o qual o **Agente Produto** se quer mover, encontra-se desocupada. Por outro lado, se o tamanho da lista for igual a dois, significa que a posição desejada encontra-se ocupada por um **Agente Recurso** ou um **Agente Transportadora**, bem como por um **Agente Produto**.

Seguidamente foi desenvolvido um método cuja função é retirar os **Agentes Produto** tanto do agendamento, como do ambiente de simulação, quando este chega à estação final. Este método utiliza os métodos *remove* e *remove_agent*, facultados pelas classes *RandomActivation* e *Multigrid* respetivamente. Usando o **Agente Produto Tipo 1** como exemplo, após removê-lo do ambiente de simulação e da lista de agendamento, este é novamente adicionado a estes dois, através do método *add_agent_1*, presente na listagem I.4 (em anexo), para que se encontrem sempre nove **Agentes Produto** no ambiente de simulação.

Após desenvolver os métodos acima mencionados, foram criados métodos específicos a cada **Agente Produto**, que englobam todos os métodos acima referidos. A figura 4.15 representa um fluxograma acerca do funcionamento do método *move_pa1*, referente ao **Agente Produto Tipo 1**, presente na listagem I.11 (em anexo).

O método *move_pa4*, referente à movimentação e lógica do **Agente Produto Tipo 4**, é semelhante aos métodos dos restantes **Agentes Produto**, porém, durante o treino, caso este agente se encontre no primeiro ponto de decisão, correspondente ao tipo de **Agente Produto** a ser treinado, a sua ação é decidida pelo algoritmo em questão, recebendo a recompensa correspondente.

Por fim, os métodos correspondentes a cada **Agente Produto**, foram aglomerados no método *get_state*, presente na listagem I.12 (em anexo), sendo este chamado no método *step* de cada **Agente Produto**.

Através de todos os métodos supracitados, foi possível desenvolver a **simulação 1** e toda a lógica inerente à interação e movimentação dos diferentes **Agentes Produto**, de modo a criar um cenário dinâmico e viável para o treino dos algoritmos inteligentes.

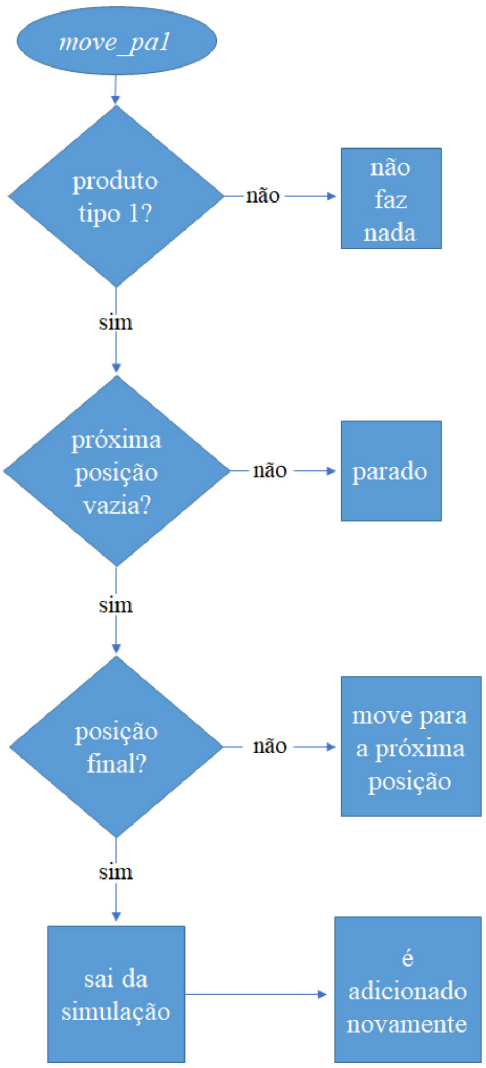


Figura 4.15: Fluxograma referente ao funcionamento do método *move_pa1*, relacionado com a movimentação do Agente Produto Tipo 1 na simulação 1.

4.2.2 Simulação 2

A **simulação 2** apresenta dois propósitos, sendo estes demonstrar a facilidade de implementar o algoritmo *Double-DQN* em ambientes de produção distintos, certificando-se que este é genérico e capaz de aprender novos ambientes com poucas alterações ao seu código, bem como servir de *digital twin* para um cenário real, de modo a expor uma possível utilização deste algoritmo num contexto real. Ao recriar um ambiente real através de uma simulação, ou *digital twin*, é adicionada a capacidade de treinar o algoritmo proposto em modo *offline*, ou seja, não é necessário utilizar o cenário real durante o treino do algoritmo, sem parar a sua produção, não originando prejuízos, nem alterações físicas.

Após o treino do algoritmo, a simulação pode ser ligada à linha de produção real, através de um *Programmable Logic Controller* (PLC) [4], para possibilitar a comunicação

[4] *Programmable logic controller*, ou PLC, é um computador utilizado em automação industrial, cuja função

entre a linha de produção e o computador onde se encontra a simulação. De uma forma mais detalhada, o PLC recebe informação proveniente da linha de produção física, através de sensores, e por sua vez, envia essa informação para a simulação. O algoritmo inteligente, neste caso o algoritmo *Double-DQN*, através da informação recebida e do SMA, controla a simulação, que por sua vez envia informação para o PLC, de modo a controlar o ambiente real.

4.2.2.1 Agentes Produto

Contrariamente à **simulação 1**, a **simulação 2** só possui a capacidade de produzir dois tipos de produtos diferentes, onde só um é controlado pelo algoritmo *Double-DQN*. As caraterísticas dos **Agentes Produto** presentes nesta simulação, encontram-se na tabela 4.5.

Tabela 4.5: Agentes Produto e as suas caraterísticas para a simulação 2.

Tipo	Nome	Quantidade	Cor	Algoritmo
Agentes Produto Tipo 1	$PA1_0$	1	Roxo	*Double-DQN*
Agentes Produto Tipo 2	$PA2_0$	1	Verde	

A **simulação 2** possui quatro tipos de **Agentes Recurso**, sendo estes o **Agente Recurso Tipo A**, o **Agente Recurso Tipo B**, o **Agente Recurso Tipo C** e o **Agente Recurso Tipo D**. Cada **Agente Recurso** consegue executar uma única operação, à exceção do **Agente Recurso Tipo D** que tem a capacidade de executar as mesmas operaçãos que os anteriores.

Os **Agentes Produto** podem passar pelos **Agentes Recurso** supracitados, de modo a produzir um produto de cada tipo. A ordem pelo qual os **Agentes Produto** têm de passar pelos **Agentes Recurso**, de modo a produzir um produto de cada tipo, encontra-se presente na tabela 4.6.

Tabela 4.6: Agentes Produto e a sua ordem de produção para a simulação 2.

Tipo	Ordem
Agentes Produto Tipo 1	RA ou RD, RC ou RD
Agentes Produto Tipo 2	RA ou RD, RB ou RD

4.2.2.2 Agentes Inativos

Na tabela 4.7, encontram-se os **Agentes Inativos** presentes na **simulação 2**, bem como as suas caraterísticas.

é automatizar processos ou até mesmo linhas de produção.

Tabela 4.7: Agentes Inativos presentes na simulação 2 e as suas caraterísticas.

Tipo	Nome	Quantidade	Cor
Agentes Pontos de Decisão	$PD_{0\ldots2}$	3	Cinzento
Agentes Grua	$G_{0\ldots16}$	17	Cinzento
Agente Estação Inicial	$Begin_0$	1	Branco / Preto
Agentes Estação Final	$End_{0\ldots3}$	4	Branco / Preto
Agentes Recurso Tipo A	RA_0	1	Branco / Preto
Agentes Recurso Tipo B	RB_0	1	Preto
Agentes Recurso Tipo C	RC_0	1	Branco / Preto
Agentes Recurso Tipo D	RD_0	1	Branco / Preto

Como é possível observar na tabela 4.7, na **simulação 2** não existem **Agentes Transportadora**, porque são substituídos por **Agentes Pontos de Decisão**, ou seja, estações onde são tomadas as decisões pelos **Agentes Produto** sobre qual caminho seguir.

Para além dos **Agentes Pontos de Decisão**, devido à arquitetura do *kit*, é necessário acrescentar **Agentes Grua**. Estes agentes são uma representação da grua presente no ambiente real, pois como é impossível simular uma grua na *framework* utilizada, esta foi representada por uma área com uma caraterística especial, sendo esta, o movimento dos **Agentes Produto**. Outra caraterística da **simulação 2**, é o facto de que os **Agentes Recurso Tipo A, Tipo C** e **Tipo D** possuem a capacidade de alterarem a sua cor, consoante a acessibilidade e ordem de produção para o agente *Double-DQN*. Após o agente *Double-DQN* ter passado por todos os **Agentes Recurso**, pela ordem correta, os **Agentes Estação Final** também alteram de cor.

As operaçãos executadas pelos recursos da **simulação 2**, encontram-se presentes na tabela 4.8. Contrariamente à **simulação 1**, na **simulação 2** o **Agente Recurso Tipo D**, como referido anteriormente, tem a capacidade de executar mais do que uma operação, pelo que quando um certo recurso se encontra ocupado, o **Agente Produto** que se encontra à espera pode deslocar-se para este recurso, se este se encontrar disponível, de modo a proceder à execução da operação necessária.

Tabela 4.8: Operações executadas pelos recursos da simulação 2.

Tipo	Operação	Tempo de Produção
Agentes Recurso Tipo A	Operação A	10 segundos
Agentes Recurso Tipo B	Operação B	10 segundos
Agentes Recurso Tipo C	Operação C	10 segundos
Agentes Recurso Tipo D	Operação A, B e C	10 segundos

4.2.2.3 Funcionamento

A **simulação 2**, presente na figura 4.16, consiste numa grelha de tamanho **5x7**, constituída por **três** transportadoras, **quatro** recursos e **três** pontos de decisão. Um requisito da simulação utilizada, é que, tanto a simulação, como a linha de produção, só possuem a capacidade de produzir dois tipos de produto, sendo só possível produzir um produto de cada tipo, simultaneamente. Contrariamente à **simulação 1**, nesta simulação os produtos têm de passar, de forma ordenada, por dois recursos. Nesta simulação, os **Agentes Produto** podem movimentar-se em quatro direções diferentes, sendo estas **direita**, **esquerda**, **cima** e **baixo**.

A alteração das cores dos **Agentes Recurso** e **Agente Estação Final**, pode ser realizada através de sensores presentes no *kit*, consoante a sua disponibilidade, ou ordem.

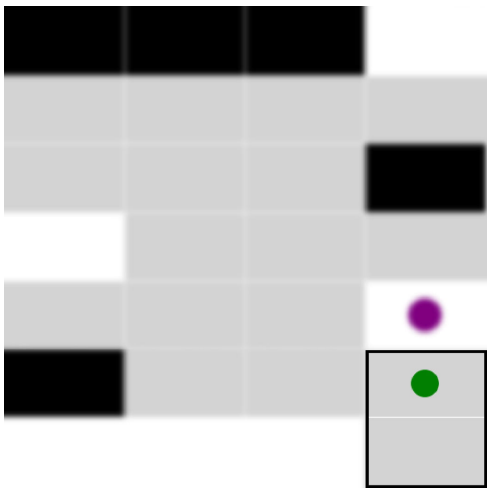


Figura 4.16: Simulação 2.

Este *kit* é constituído por **quatro** transportadoras, **uma** grua, cujo objetivo é transportar a matéria-prima da transportadora para os recursos **RC** ou **RD**, **dois** pontos de decisão, **um** *buffer* de saída, que apresenta o objetivo de armazenar os produtos após serem fabricados e **quatro** recursos diferentes, sendo estes os recursos **RA**, **RB**, **RC** e **RD**. Como presente na tabela 4.8, enquanto os recursos **RA**, **RB** e **RC** têm a capacidade de executar uma só operação, o recurso **RD** consegue executar as três operações executadas por cada um dos outros recursos acima referidos. É de notar que só se pode encontrar um **Agente Produto** na grua de cada vez e que, tanto a simulação como o *kit*, apresentam transportadoras bidirecionais.

Apesar da **simulação 2** ser semelhante ao *kit*, esta apresenta duas diferenças, nomeadamente:

- apresentar três pontos de decisão em vez de dois,
- apresentar três transportadoras em vez de quatro.

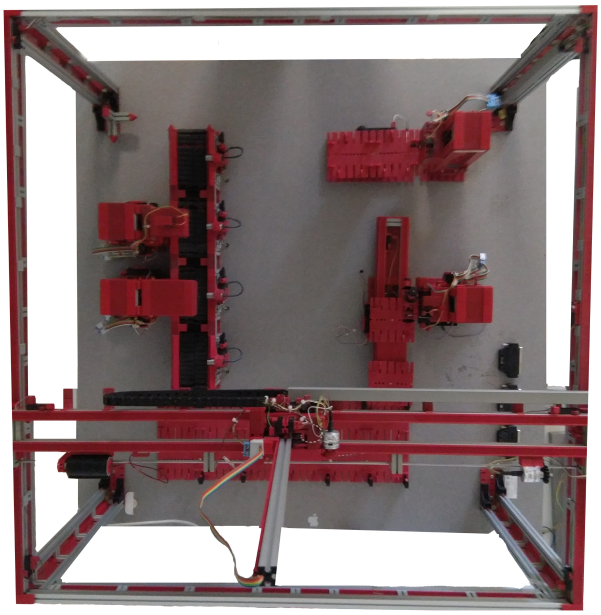


Figura 4.17: Cenário real utilizado nesta dissertação.

A primeira diferença supracitada foi implementada, de modo a tornar o ambiente de simulação mais dinâmico, visto que é acrescentado mais um ponto de decisão onde o agente *Double-DQN* pode deslocar-se para a grua.

Inicialmente todos os **Agentes Produto** começam na estação inicial (figura 4.18). De notar, que o **Agente Produto Tipo 2** possui um encaminhamento pré-programado, enquanto o **Agente Produto Tipo 1**, também denominado de agente *Double-DQN*, é controlado pelo algoritmo *Double-DQN*, sendo assim capaz de escolher o encaminhamento que permite produzir o produto da forma mais rápida possível, passando pelos recursos necessários.

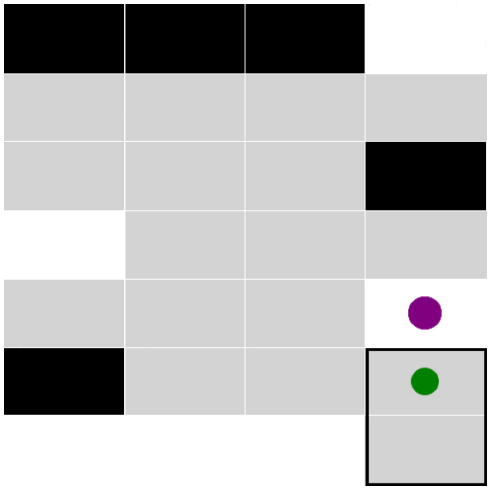


Figura 4.18: Primeira transportadora, presente na simulação 2.

Após saírem do **Agente Estação Inicial**, os **Agentes Produto** deslocam-se para o primeiro **Agente Ponto de Decisão**. Neste, existe a possibilidade de se deslocarem para o **Agente Recurso Tipo C** ou para o **Agente Recurso Tipo D**, através dos **Agentes Grua**, ou para o **Agente Recurso Tipo A**, utilizando a primeira transportadora. Os agentes supracitados encontram-se presentes na figura 4.19.

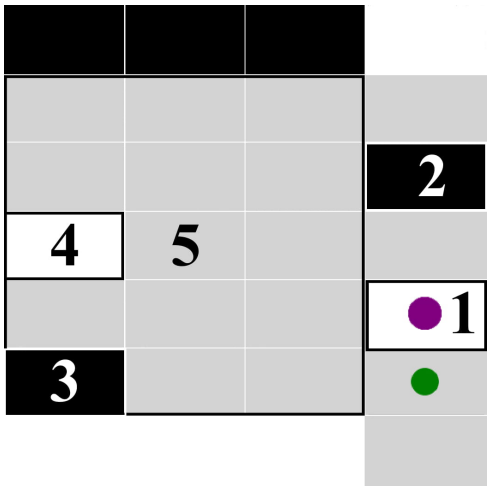
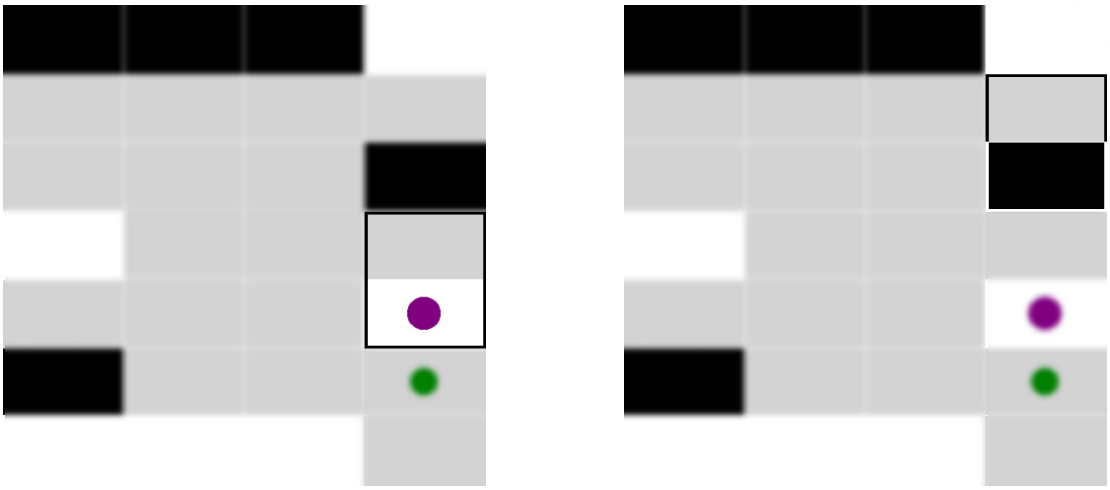


Figura 4.19: Agentes Grua e Agentes Recurso, presentes na simulação 2.

De notar que a numeração presente na figura 4.19, possui o seguinte significado:

- o **número um**, representa o **Agente Recurso Tipo A**,
- o **número dois**, representa o **Agente Recursos Tipo B**,
- o **número três**, representa o **Agente Recurso Tipo C**,
- o **número quatro**, representa o **Agente Recurso Tipo D**,
- o **número cinco**, representa os **Agentes Grua**.

A única maneira de um **Agente Produto** se mover até ao **Agente Recurso Tipo C** ou até ao **Agente Recurso Tipo D** é através dos **Agentes Grua**, sendo que também tem de os utilizar caso queira regressar para as transportadoras. No caso do **Agente Produto** ter sido manipulado pelo **Agente Recurso Tipo A**, este desloca-se para o segundo **Agente Ponto de Decisão**, onde tem novamente a possibilidade de se deslocar para os **Agentes Grua**, ou seguir para a próxima transportadora até ao **Agente Recurso Tipo B**.



a Segunda transportadora e respetivo recurso e ponto de decisão, presentes na simulação 2.

b Terceira transportadora e respetivo recurso e ponto de decisão, presentes na simulação 2.

Figura 4.20: Restantes transportadoras, presentes na simulação 2.

No final da produção, todos os **Agentes Produto** são encaminhados pelos **Agentes Grua** até ao *buffer* de saída, representado pelos **Agentes Estação Final**, presente na figura 4.21.

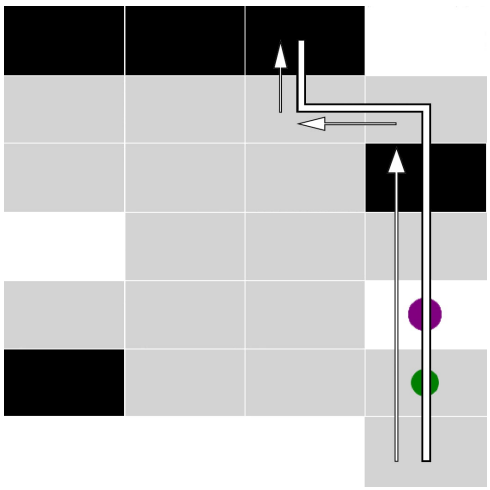


Figura 4.21: *Buffer* final presente na simulação 2, representado pelos Agentes Estação Final.

Como foi referido anteriormente, o **Agente Produto Tipo 2** possui um encaminhamento definido, presente na figura 4.22.

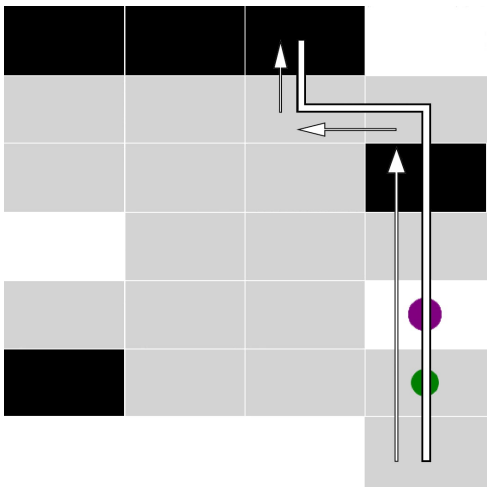


Figura 4.22: Rota de encaminhamento do Agente Produto Tipo 2, presente na simulação 2.

Nesta simulação, o agente *Double-DQN* tem duas opções de encaminhamento, consoante a posição do **Agente Produto Tipo 2**, sendo estas:

- caso o agente *Double-DQN* seja o primeiro a sair da estação inicial, a opção de encaminhamento mais rápida é deslocar-se ao **Agente Recurso Tipo A** e, seguidamente, ao **Agente Recurso Tipo D**, perfazendo **28 pontos**,
- caso o agente *Double-DQN* seja o segundo a sair da estação inicial, e o **Agente Produto Tipo 2** se encontre no **Agente Recurso Tipo A**, a opção de encaminhamento mais rápida é deslocar-se ao **Agente Recurso Tipo D** e, seguidamente, ao **Agente Recurso Tipo C**, perfazendo **33 pontos**.

4.2.2.4 Implementação e Lógica

Nesta simulação, é utilizada a classe *RandomActivation* durante a fase de treino, e a classe *BaseScheduler* durante a fase de testes, ambas presentes no sub-módulo *Time*. Na **simulação 2**, é também utilizada a classe *MultiGrid*, proveniente do sub-módulo *Space*, tal como na **simulação 1**. De notar que nesta simulação, o **Agente Produto Tipo 2**, ou seja, o **Agente Produto** que não está a ser treinado, está programado de forma a que nunca aconteça o caso de ambos os **Agentes Produto** se encontrarem na mesma posição.

A maioria dos métodos e classes utilizados na **simulação 2** são baseados nos métodos presentes na **simulação 1**, como por exemplo os métodos utilizados na instanciação dos agentes, ou a classe *Model*, presente na listagem I.5 (em anexo).

A **simulação 2** começou a ser desenvolvida pela classe do servidor, ou seja, a classe *server*. Apesar do método *agent_portrayal*, presente na classe *server* da **simulação 2** ser semelhante ao método *agent_portrayal* da classe *server* da **simulação 1**, este apresenta a lógica da alteração de cor dos **Agentes Recurso** e **Agentes Estação Final**. Para tal foram criadas as seguintes variáveis:

- *change_color_a*, cujo objetivo é alterar a cor do **Agente Recurso Tipo A**,
- *change_color_c*, cujo objetivo é alterar a cor do **Agente Recurso Tipo C**,
- *change_color_d*, cujo objetivo é alterar a cor do **Agente Recurso Tipo D**,
- *change_color_end*, cujo objetivo é alterar a cor dos **Agentes Estação Final**.

Estas variáveis tomam os valores *true* ou *false*, de modo a alterarem a cor do respetivo agente consoante o estado da simulação.

A **simulação 2** possui uma lógica de movimentação e interação dos **Agentes Produto** mais simples do que a **simulação 1**, visto que, só se encontram dois **Agentes Produto** na simulação.

Esta simulação apresenta uma regra particular, sendo esta o facto de só se poder encontrar um **Agente Produto** nos **Agentes Grua** de cada vez. Como tal foi necessário criar uma variável, *grua_on*, cujo valor é *true* ou *false*, consoante se um dos **Agentes Produto** se encontra num **Agente Grua** ou não.

À semelhança do método *check_right_avail* da **simulação 1**, presente na listagem I.9 (em anexo), o método *move_pa2*, correspondente ao movimento do **Agente Produto Tipo 2** e presente na listagem I.14 (em anexo), também averigua a vizinhança do agente através do método *get_neighbors* da classe *MultiGrid*, de modo a verificar se a posição seguinte se encontra ocupada ou não.

Como referido anteriormente, na **simulação 2**, o **Agente Produto Tipo 2** possui um caminho de produção previamente programado, sendo que este foi armazenado num vetor de tuplos[5], ou *tuple* em inglês.

Caso o **Agente Produto Tipo 2** esteja no **Agente Recurso Tipo A**, este último tem de ficar com a cor preta, logo uma variável comum aos dois agentes, *resource_occupied* é igualada a *true*, indicando que o **Agente Recurso Tipo A** se encontra ocupado. Por sua vez, a variável *change_color_a* é igualada a *false*, de modo a mudar a cor deste agente para preto.

Por fim, quando o **Agente Produto** alcança um **Agente Estação Final**, este é removido do agendamento e do ambiente, de forma idêntica à **simulação 1**, sendo adicionado novamente, de modo a realizar mais um ciclo de produção.

Contrariamente à **simulação 1**, a **simulação 2** não apresenta o método *stop_move*. Como tal, os **Agentes Produto** não necessitam de ficar parados nos **Agentes Recurso**, porém, sempre que passam por um, a variável correspondente ao número de pontos de cada **Agente Produto** é incrementada por **dez**, sendo este o tempo de manipulação de cada **Agente Recurso** presente na **simulação 2**. Esta alteração foi realizada, de modo a diminuir o tempo do treino do agente *Double-DQN*, visto que desta forma, este encontra-se em constante movimento.

[5]Um tuplo consiste numa sequência de elementos, como por exemplo coordenadas cartesianas.

4.2.3 Armazenamento de Dados

O armazenamento dos dados adquiridos ao longo dos treinos e dos cenários de teste é extremamente importante, pois com estes é possível comparar o desempenho dos diferentes algoritmos, averiguar se os algoritmos *Q-learning* e *Double-DQN* treinaram de forma correta e armazenar dados essenciais ao treino dos agentes, de modo a ser possível parar a execução do treino, retomando, posteriormente, a partir da etapa anterior.

Os dados foram armazenados através de um método, denominado de *save_model*, presente na listagem I.15 (em anexo). Este foi executado de **10000** em **10000** iterações de treino, ou no final dos cenários de teste, ou seja, após **1000** iterações de teste.

Os dados armazenados estão divididos em dois grupos, sendo estes:

- os dados necessários para averiguar o desempenho dos agentes durante o treino e comparar os diferentes algoritmos nos cenários de teste,
- os dados necessários para o treino dos agentes, de modo a retomar o treino após uma paragem.

O primeiro grupo de dados foi armazenado em ficheiros *excel*, de extensão *.csv*, através da biblioteca *panda*. Nas iterações onde o método *save_model* não é executado, estes dados são armazenados em *dataframes* provenientes da biblioteca supracitada. Foi também utilizada a biblioteca *os*, de modo a averiguar se os ficheiros *excel* já tinham sido previamente criados, ou era necessário criar ficheiros novos.

O segundo grupo de dados foi armazenado em ficheiros de extensão *.pkl*, cujo objetivo é armazenar objetos através do método *save_obj*. Por sua vez, o carregamento do conteúdo destes ficheiros para os respetivos objetos, é realizado através do método *load_obj*. Estes métodos usam a biblioteca *pickle*, que permite serializar [6] e deserializar objetos na linguagem de programação *Python*.

No início do treino de cada agente, foi necessário criar os ficheiros supracitados, como tal foi desenvolvido o método *init_cache*, cujo objetivo é criar os diferentes ficheiros *.pkl* necessários para o treino dos agentes. É importante referir que a *replay memory* foi armazenada num objeto da coleção *deque*, proveniente da biblioteca *collections*.

4.3 Algoritmos

Nesta dissertação é proposta a utilização de IA para fornecer inteligência individual a agentes. É proposta a utilização de DRL, mais especificamente o algoritmo *Double-DQN*.

Como forma de comparação, são desenvolvidos dois algoritmos utilizados para a resolução deste tipo de problema, sendo estes o algoritmo *Q-Learning*, proveniente de RL, e o algoritmo heurístico *A**. Posteriormente ao treino de cada agente, são realizados

[6] Serialização consiste na tradução uma estrutura de dados, ou um objeto, para um formato que possa ser armazenado.

diferentes testes, sendo que os algoritmos são avaliados com base em métricas adquiridas durante o treino dos agentes, bem como métricas baseadas no desempenho dos agentes em cada um dos cenários de teste.

4.3.1 Algoritmo *Double Deep Q-Network*

O modelo da CNN utilizada no algoritmo *Double-DQN*, desenvolvido nesta dissertação, é semelhante ao modelo desenvolvido em [22]. A CNN utilizada nesta dissertação possui as seguintes caraterísticas:

- três camadas convolucionais, uma de tamanho 32 e duas de tamanho 64,
- entre cada camada convolucional encontra-se uma camada *pooling* e uma camada *activation*,
- uma camada *flatten*,
- duas camadas *dense*, sendo que a primeira camada um *output* de tamanho **256**, enquanto a segunda camada tem um *output* de tamanho igual ao número de ações que o agente pode realizar,
- entre cada camada *dense* encontra-se uma camada *activation*,
- o otimizador utilizado foi o otimizador *RMSprop*, com um *learning rate* = $0,00025$, $\rho = 0,95$ e $\epsilon = 0,01$,
- a função de erro, ou *loss function* em inglês, utilizada foi a função *logcosh*.

O *input* da primeira camada convolucional, ou seja, o *input* da rede neuronal, é uma imagem do ambiente de tamanho **80x80x4**. No caso da primeira simulação, o *output* da última camada *dense*, ou seja o *output* da rede neuronal, é igual a **três**, visto que, o agente pode realizar as seguintes ações: movimentar-se para a **direita**, para **baixo** ou **ficar parado**.

Por sua vez, no caso da **simulação 2**, o agente tem a possibilidade de se mover para a **direita**, para a **esquerda**, para **cima** e para **baixo**, pelo que o *output* da última camada *dense* é igual a **quatro**. De notar, que o tamanho da última camada *dense* é a única alteração necessária na rede, de ambiente para ambiente, visto que, cada ambiente pode apresentar o seu conjunto de ações.

A última camada *dense* retorna um vetor que contém os valores Q para o estado em que o agente se encontra, sendo que a ação escolhida corresponde à posição do vetor que possui o maior valor Q.

Nesta arquitetura são adicionadas duas otimizações ao algoritmo DQN para que este se torne no algoritmo *Double-DQN*, baseadas nas otimizações utilizadas em [22]. A primeira otimização, consiste na utilização de uma rede alvo, [13], explicada na sub-subsecção 3.1.2.1.

A segunda otimização consiste na utilização da função de erro, ou *loss function* em inglês, *logcosh*. Em [22], é utilizada a função de erro *huber loss*, que consiste na junção das funções de erro *Mean Squared Error (MSE)* e *Mean Absolute Error (MAE)*, ou seja, utiliza o erro MSE para os valores pequenos e o erro MAE para os valores maiores. Este erro é definido pelo seguinte sistema de equações:

$$Huber(a) = \begin{cases} \frac{1}{2}a^2, & \textbf{para} |a| \leq 1, \\ (|a| - \frac{1}{2}), & \textbf{caso contrário}. \end{cases} \tag{4.1}$$

Nesta dissertação não é utilizada a função de erro *huber loss* especificamente, mas sim uma função semelhante, denominada de *logcosh*.

Os parâmetros utilizados no treino do agente *Double-DQN* nas duas simulações, foram baseados nos parâmetros utilizados em [22]. É importante referir que os parâmetros utilizados na **simulação 1** para o agente *Double-DQN* são semelhantes aos utilizados no agente *Q-learning* (tabela 4.15) para não existir discrepância no treino dos dois algoritmos. Por sua vez, o treino do agente *Double-DQN* na **simulação 2** foi mais demorado, visto que, para além da segunda simulação apresentar uma complexidade superior à primeira, nesta o agente tem total liberdade no ambiente.

4.3.1.1 Recompensas e Parâmetros - Simulação 1

No caso da primeira simulação, o agente *Double-DQN* começa a escolher ações desde a estação inicial até ao ponto de decisão, correspondente ao **Agente Produto Tipo 1**. Como tal as ações que o agente pode escolher são movimentar-se para a **direita**, para **baixo** ou ficar **parado**. Com isto, o agente aprende que tem de se movimentar dentro da primeira transportadora horizontal e quando alcança o ponto de decisão escolher a melhor transportadora vertical, consoante a posição dos restantes **Agentes Produto Tipo 1**. A partir do momento em que o agente toma a decisão, o resto do movimento é programado, visto que, não afeta o tempo de produção e torna o treino mais rápido.

As recompensas utilizadas no treino do agente *Double-DQN* para a **simulação 1**, encontram-se no intervalo de valores de $[-1, 1]$. Estas encontram-se presentes na tabela 4.9.

Tabela 4.9: Recompensas do algoritmo *Double-DQN* para a simulação 1.

Casos	Recompensas
A	-1
B	-0,1
C	-0,01
D	0,01
E	R(t)

De notar que os casos englobados pelas letras **A**, **B**, **C** e **D** são os seguintes:

- **casos A**, consistem no agente sair da linha de produção ou deslocar-se para uma posição ocupada por outro **Agente Produto**, após o primeiro ponto de decisão,
- **caso B**, consiste na situação em que o agente *Double-DQN* se desloca para uma posição onde se encontra um **Agente Produto**, antes do primeiro ponto de decisão,
- **caso C**, consiste na situação em que o agente *Double-DQN* fica parado, não estando um **Agente Produto** na próxima posição,
- **caso D**, consiste na situação em que o agente *Double-DQN* fica parado quando se encontra um **Agente Produto** na próxima posição,
- **caso E**, consiste no caso em que o agente *Double-DQN*, toma uma decisão no primeiro ponto de decisão e não se encontra nenhum **Agente Produto** na posição seguinte.

A recompensa do **caso E** é calculada com base na função 4.2. Esta relaciona a recompensa dada ao agente com o tempo que este demorou a ser produzido, ou seja, a pontuação que fez, sendo que a variável t representa o tempo de produção.

$$R(t) = \frac{100 - t}{47} \tag{4.2}$$

De notar que o menor valor de t para a **simulação 1** é de **53 pontos**, ou segundos, correspondendo ao maior valor de R, ou seja, **um**. Após alguma observação foi possível concluir, que a realização de **100 pontos**, ou mais, é pouco frequente. Como tal, de modo a colocar a recompensa num intervalo de $[-1, 1]$, foi necessário realizar as operações aritméticas presentes em 4.2, visto que, $100 - 53 = 47$ e $\frac{47}{47} = 1$, sendo esse o maior valor que o agente pode receber como recompensa. Caso o valor de pontos seja maior que **100**, o agente recebe pontuação negativa, sendo benéfico para a sua aprendizagem.

Tabela 4.10: Parâmetros de treino do algoritmo *Double-DQN* para a simulação 1.

Parâmetro	**Valor**
Ações	[0, 1, 2]
Número de Ações	3
$\epsilon_{inicial}$	1
ϵ_{final}	0,1
Exploração	20000
Observação	1000
τ	500
Replay Memory	500000
Batch	32
γ	0,99
Epochs	100000

Os parâmetros presentes na tabela 4.10, possuem o seguinte significado:

- **ações**, são representadas por um vetor que armazena as possíveis ações que o agente pode realizar,
- $\epsilon_{inicial}$, representa o valor inicial da política ϵ-*greedy*, sendo este um ou 100%,
- ϵ_{final}, representa o valor final da política ϵ-*greedy*, sendo este 0,1 ou 10%,
- **exploração**, representa o valor que define o rácio da política ϵ-*greedy*,
- **observação**, representa o número de iterações em que o agente só escolhe ações aleatórias, de modo a popular a *replay memory*,
- τ, representa a frequência com que a rede alvo é atualizada,
- *replay memory*, representa o tamanho da *replay memory*,
- ***batch size***, representa o número de exemplos de treino facultados à rede em cada iteração de treino,
- γ, representa a taxa de aprendizagem do agente,
- ***epochs***, representa o número total de iterações realizadas no treino do agente.

4.3.1.2 Recompensas e Parâmetros - Simulação 2

Contrariamente à **simulação 1**, na **simulação 2** o agente tem total liberdade sobre o ambiente, querendo isto dizer que todos os movimentos são decididos pelo agente *Double-DQN*. Como os produtos têm obrigatoriamente de passar por uma ordem fixa de recursos, esta tem de ser ensinada ao agente. Porém, nesta arquitetura esta ordem não é fornecida à CNN como *input*, como tal necessita de ser demonstrada através da simulação. Sendo assim, a cor dos **Agentes Recurso**, pelo qual o agente *Double-DQN* necessita de passar ou a cor dos **Agentes Estação Final**, é alterada.

Desta forma, o agente aprende que, quando certo **Agente Recurso** se encontra com uma cor diferente, este vai receber uma recompensa positiva caso se desloque para essa posição. De notar que o agente *Double-DQN* só pode passar uma única vez em cada **Agente Recurso**. Após o agente *Double-DQN* passar pelos três **Agentes Recurso** necessários, os **Agentes Estação Final** alteram de cor.

Nesta simulação, as recompensas encontram-se no intervalo de $[-1, 1]$, onde a recompensa negativa é atribuída quando o agente faz algo incorreto, enquanto a recompensa positiva é atribuída quando o agente faz algo correto. As recompensas utilizadas no treino encontram-se na tabela 4.11.

Tabela 4.11: Recompensas do algoritmo *Double-DQN* para a simulação 1.

Casos	Recompensas
A	-1
B	-0,01
C	0,5
D	1

Os casos englobados pelas letras **A**, **B**, **C** e **D** são os seguintes:

- **casos A**, consistem em circunstâncias onde o agente *Double-DQN* se desloca para uma posição de cor preta, ou seja, para um **Agente Recurso** fora de ordem, ou já utilizado, ou para um **Agente Estação Final**, sem ter passado pelos **Agentes Recurso** necessários,

- **casos B**, consistem nos casos onde o agente *Double-DQN* realiza qualquer movimento não englobado nos outros casos,

- **caso C**, consiste na situação em que o agente *Double-DQN* se desloca para um **Agente Recurso** de cor branca, ou seja, um **Agente Recurso** livre,

- **caso D**, consiste na situação em que o agente *Double-DQN* se desloca para um **Agente Estação Final** de cor branca.

Relativamente à segunda simulação, foram utilizados os seguintes parâmetros no treino do agente:

Tabela 4.12: Parâmetros de treino do algoritmo *Double-DQN* para a simulação 2.

Parâmetro	**Valor**
Ações	[0, 1, 2, 3]
Número de Ações	4
$\epsilon_{inicial}$	1
ϵ_{final}	0,1
Exploração	50000
Observação	10000
τ	2000
Replay Memory	50000
Batch	32
γ	0,99
Epochs	200000

Os parâmetros presentes na tabela 4.12 possuem o mesmo significado que os parâmetros presentes na tabela 4.10. De notar que a rede aprendeu a melhor política de recompensas em menos *Epochs* do que os indicados, no entanto o treino apresenta um número superior de *Epochs* para garantir a aprendizagem da rede.

O agente só finalizou o seu treino quando conseguiu alcançar **5000 ciclos de produção**.

4.3.1.3 Processamento de Imagem

A rede neuronal recebe como *input* uma pilha de **quatro** imagens referentes ao ambiente. Para tal é necessário adquirir essas imagens e posteriormente realizar algum processamento sobre as mesmas, de modo a diminuir o custo computacional.

Tal como referido na subsecção 4.1.2, a *framework Mesa* utiliza um servidor local para representar o ambiente de simulação. Este servidor local pode ser utilizado com qualquer *web browser* existente, no entanto não é viável adquirir imagens do ambiente de forma manual, ou utilizar um *software* externo dedicado a este propósito, visto que, iria aumentar o custo computacional e inviabilizar a utilização do computador. Como tal, é utilizado um conjunto de ferramentas, denominadas de *Selenium*, cuja função é automatizar *web browsers*, através de bibliotecas e métodos específicos, desenvolvidos nas linguagens de programação *Python* e *Java*.

Nesta dissertação é utilizado o executável *chromedriver*, proveniente do conjunto de ferramentas *Selenium*, referente ao *web browser Google Chrome*. Através da classe *webdriver*,

proveniente da biblioteca *selenium*, é possível instanciar o executável *chromedriver*, sendo que primeiramente, é necessário definir a diretoria onde este se encontra.

Listagem 4.1: Inicialização do executável *chromedriver*

```
1 from selenium import webdriver
2
3 driver = webdriver.Chrome(chromeDriver_path)
```

Após a instanciação do executável, cada vez que o programa é executado é aberta uma página do *web browser Google Chrome*, pronta a ser automatizada.

Este executável é utilizado para adquirir as imagens do ambiente de simulação e pausar/retomar, de uma forma automatizada, a simulação, sempre que o método *save_model*, presente na listagem I.15 (em anexo), é chamado. Desta forma, foi necessário analisar o código *Hypertext Markup Language* (HTML)[7] da página referente à simulação, de modo a encontrar o código que representa o ambiente de simulação, bem como o código referente ao botão *play/stop*. A variável que representa a imagem do ambiente de simulação em linguagem HTML e o método *play_pause_model*, que permite pausar/retomar a simulação de uma forma automática, encontram-se na listagem 4.2.

Listagem 4.2: Variável HTML do ambiente e método *play_pause_model*

```
1 from selenium import webdriver
2
3 getbase64Script =
4 "canvasRunner=document.getElementsByClassName('world-grid');"
5                   "return canvasRunner[0].toDataURL().substring(22)"
6
7 def play_pause_model():
8     play_pause = driver.find_element_by_id("play-pause")
9     play_pause.click()
```

Em seguida, é necessário converter a imagem proveniente do código HTML, imagem do tipo *base64*, para um *numpy array*[8], utilizando a biblioteca *numpy*. Esta conversão é realizada no método *html_image*, presente na listagem I.22 (em anexo), onde primeiramente é necessário abrir a imagem *base64* como *BytesIO*, utilizando as classes *Image* e *BytesIO*, provenientes das bibliotecas *PIL* e *io* respetivamente, e a biblioteca *base64*.

A imagem proveniente do ambiente é uma imagem de tamanho **500x500**, presente na escala de cores *RGB*[9], esta origina um vetor de tamanho $500 * 500 * 3 = 750000$, sendo que quando as imagens são empilhadas, estas originam um vetor de tamanho $750000 * 4 = 3000000$, apresentando um elevado custo computacional. Como tal, é necessário utilizar algumas técnicas de processamento de imagem, como por exemplo converter a imagem

[7] HTML ou *Hypertext Markup Language* em inglês, é a linguagem utilizada para documentos, cujo propósito é serem exibidos numa página *web*.

[8] *Numpy array* é um vetor, ou *array* em inglês, proveniente da biblioteca *numpy*, extremamente útil na manipulação de dados na linguagem de programação *Python*.

[9] *RGB* é um modelo de cor constituído por três canais de cores, sendo estas vermelho, ou *red* em inglês, verde, ou *green* em inglês, e azul, ou *blue* em inglês.

para escala de cinzentos[10] e reduzir o tamanho da imagem de **500x500** para **80x80**. Sendo assim, através da biblioteca *cv2*, correspondente à biblioteca *opencv* noutras linguagens de programação, é possível aplicar estas técnicas de processamento de imagem, reduzindo o tamanho do vetor final de **3000000** para **25600**, visto que, $80*80*1*4 = 25600$, reduzindo imenso o custo computacional. De notar que este processamento só é possível porque uma CNN tem a capacidade de processar imagens de pequena dimensão e com um só canal de cor.

Por fim, o método *image_treatment*, presente na listagem I.24 (em anexo), aglomera as duas funções supracitadas, originando uma imagem do ambiente em escala de cinzentos de tamanho **80x80**, pronta a ser empilhada e enviada para a rede. De notar que na listagem I.24, também se encontram os métodos *grab_image* e *image2video*. O primeiro método, utiliza as bibliotecas *cv2* e *os* para armazenar imagens do treino, ou teste, de modo a ser realizado um vídeo da simulação, enquanto o segundo método utiliza as imagens, adquiridas pelo método anterior, e converte-as num vídeo, sendo que neste método são utilizadas as bibliotecas *glob* e *cv2*, onde a primeira serve para armazenar as imagens num objeto, e organizá-las por data, e a segunda para, através dessas imagens, criar um vídeo.

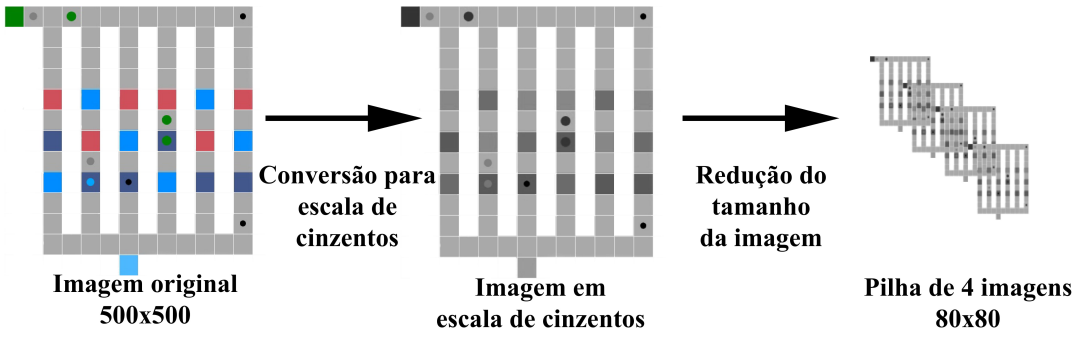


Figura 4.23: Exemplo de uma imagem após sofrer as transformações necessárias para o treino do algoritmo.

4.3.1.4 Implementação do Algoritmo *Double Deep Q-Network*

O modelo da rede neuronal utilizada nesta dissertação, foi desenvolvido utilizando os métodos facultados pela biblioteca *keras* onde, primeiramente, foi desenvolvido o método *buildmodel*, cuja função é criar o modelo da CNN. Como referido anteriormente, esta é constituída por duas camadas convolucionais, intercaladas por camadas *pooling* e *activation relu*, seguidas de uma camada *flatten* e duas camadas *dense* intercaladas por uma camada *ativation relu*. Na subsecção 4.3.1, encontra-se uma explicação mais detalhada da rede, sendo que na listagem I.19 (em anexo) encontra-se o método *buildmodel*, utilizado para desenvolver a CNN.

[10]Escala de cinzentos, ou *gray scale* em inglês, consiste num modelo de cor constituído por um único canal.

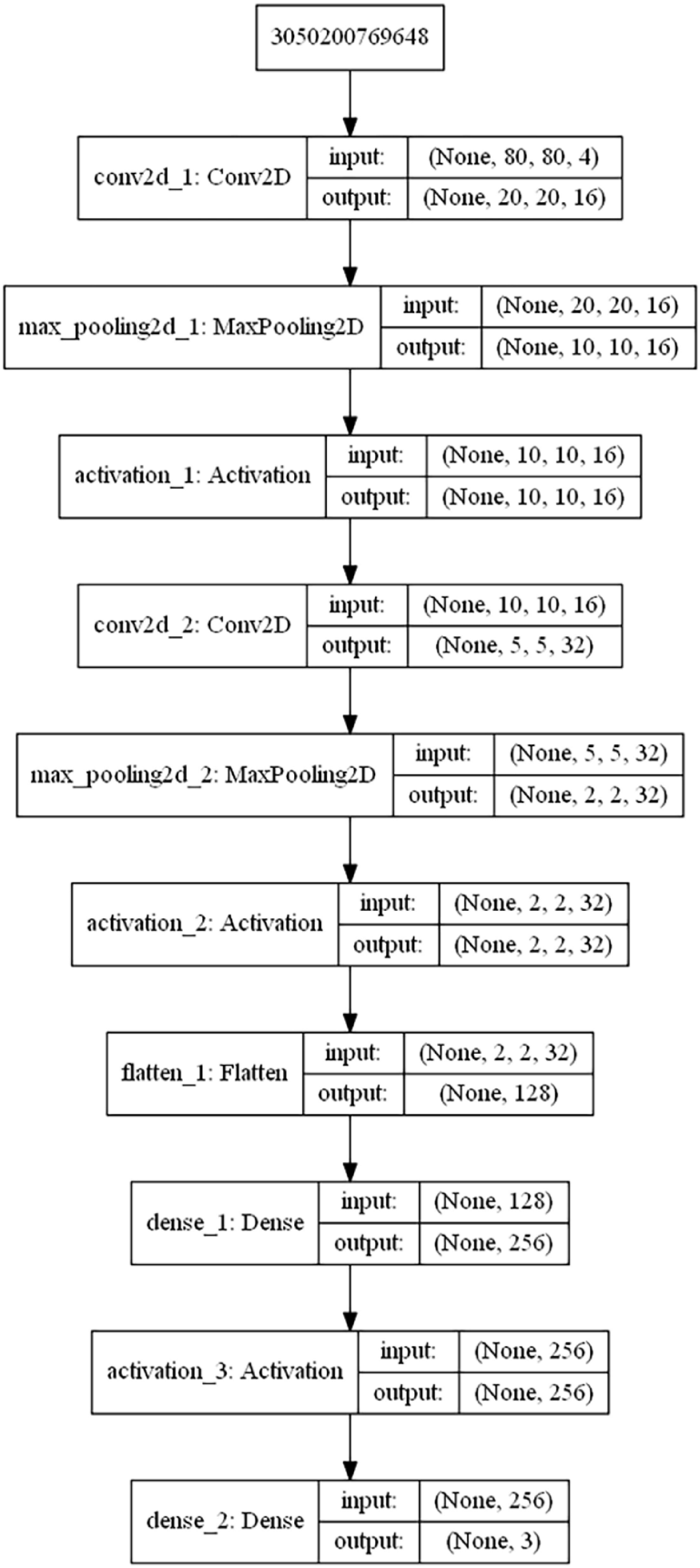


Figura 4.24: Esquemático da CNN utilizada.

Na figura 4.24 encontra-se um esquemático da CNN utilizada nesta dissertação.

Após a criação do modelo da rede, este é armazenado num ficheiro *.json*, através do método *model_from_json* proveniente da biblioteca *keras*, enquanto os pesos da rede são armazenados num ficheiro *.h5*, através do método *save_weights* também proveniente da biblioteca *keras*.

Na subsecção 4.3.1, é referido que uma das otimizações aplicadas no algoritmo *Double-DQN* presente nesta dissertação, é a utilização de uma função de erro *logcosh*, similar à função de erro *huber loss* utilizada em [22]. A justificação para se utilizar a função de erro *logcosh*, ao invés da função de erro *huber loss*, é porque a biblioteca *keras*, utilizada nesta dissertação para a criação do modelo da rede neuronal, não possui a função de erro *huber loss*.

Após a criação da rede neuronal, foi necessário desenvolver um método, cuja função é carregar o modelo e os respetivos pesos da rede neuronal, caso o treino seja novamente executado após uma paragem. Para tal foi utilizado o método *model_from_json* proveniente da biblioteca *keras* para carregar o modelo, juntamente com o método *load_weights*, também proveniente da biblioteca *keras*, para carregar os pesos.

Como referido na sub-subsecção 3.1.2.1, durante o treino foi utilizada uma rede alvo, de modo a auxiliar o treino da rede principal e ajudar a sua convergência. Esta rede consiste num clone da rede principal, sendo que existe uma discrepância nos pesos de ambas, ou seja, os pesos da rede principal são alterados em cada iteração de treino, enquanto os pesos da rede alvo só são alterados de τ em τ iterações. De modo a clonar o modelo da rede principal, foi utilizado o método *clone_model* proveniente da biblioteca *keras*, enquanto para atribuir os pesos da rede principal à rede alvo, foi utilizado o método *set_weights* também proveniente desta biblioteca.

No início do treino existem algumas variáveis e objetos que necessitam de ser carregados, bem como a criação da pilha de quatro imagens do estado inicial. Para tal foi criado o método *init_train_network*, presente na listagem I.25 (em anexo), cujo objetivo é carregar as variáveis e objetos necessários, bem como criar a pilha supracitada. Os objetos e variáveis a serem carregados consistem nos seguintes:

- *replay memory*, carregada para a coleção *deque*,
- τ, carregado para a variável *tau* e representa o número de iterações em que a rede alvo é atualizada,
- *t_finish*, representa o número de iterações de treino em modo de exploração pelo agente,
- ϵ, carregado para a variável *epsilon* e representa o valor de exploração do agente,
- *t*, representa o número de iterações total de treino.

Por sua vez, é necessário adquirir uma imagem do estado inicial e posteriormente criar uma pilha inicial, através da biblioteca *numpy*, com a imagem, correspondente ao estado inicial, e três cópias.

Em seguida, foi desenvolvido um método que permite o agente escolher uma ação. Como tal foi criado o método *find_max_q*, presente na listagem I.26 (em anexo), cuja função é averiguar se o valor de ϵ é maior que um valor aleatório, gerado pelo método *rand*, proveniente da biblioteca *numpy*, entre **zero** e **um**. Neste caso, se ϵ for maior que o valor aleatório, o agente realiza uma ação aleatória, através da biblioteca *random*. Caso contrário, a rede neuronal prevê a melhor ação para o estado atual, através do método *predict*. Este método recebe como parâmetro de entrada uma pilha de quatro imagens, referentes ao estado atual, e retorna um vetor de valores *Q*, proveniente da previsão da rede neuronal, sendo que a melhor ação corresponde ao indíce do vetor que possui o maior valor *Q*.

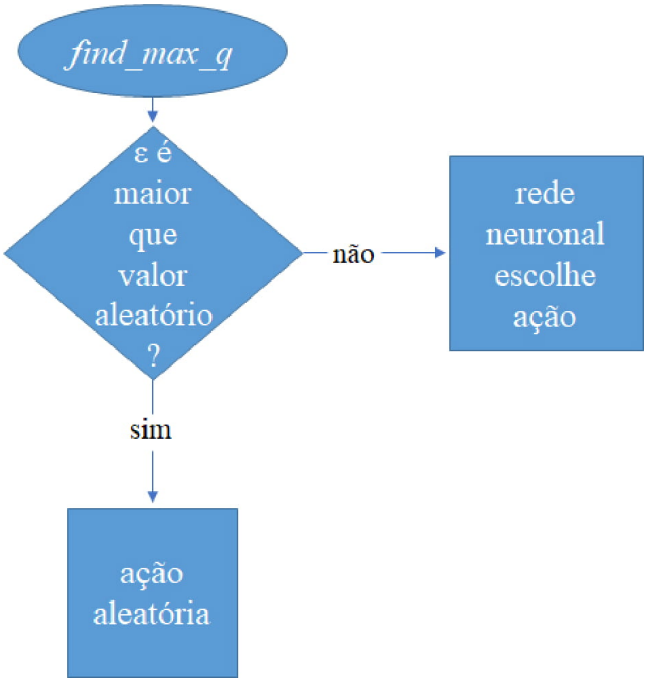


Figura 4.25: Fluxograma referente ao funcionamento do método *find_max_q*, relacionado com a escolha de ações pelos algoritmos *Q-learning* e *Double-DQN*.

O método *find_max_q* utilizado para o treino do agente *Double-DQN* na **simulação 2** apresenta algumas alterações. Para diminuir o tempo de treino do agente na **simulação 2**, é retirada a opção de este sair da linha de produção, ou seja, nas posições do ambiente onde o agente tem a possibilidade de sair da linha de produção, as suas ações são filtradas e consideradas inválidas. Para tal, averiguou-se quais as posições do vetor, proveniente da previsão da rede neuronal, que levariam o agente a manter-se dentro da linha de produção, ou seja, a cometer uma ação válida, sendo posteriormente escolhido o maior valor *Q* e a respetiva ação, de entre os valores *Q* correspondentes às ações válidas. Caso a ação seja aleatória, a(s) ação ou ações inválida(s) são excluídas do vetor de ações.

O treino da CNN foi realizado no método *train_memory_batch*, presente na listagem I.27 (em anexo). Neste método é criada uma *replay memory* auxiliar que contém 32 amostras aleatórias da *replay memory* principal, sendo estas utilizadas para treinar a rede principal. Cada posição da *replay memory* auxiliar possui um conjunto de informações fulcrais, como a pilha atual, a ação que o agente escolheu, a recompensa obtida, a pilha com o estado seguinte e se é considerado um estado terminal.

A rede principal prevê todos os valores Q para o estado atual, enquanto a rede alvo prevê os valores Q para o próximo estado, sendo utilizado o maior valor correspondente à melhor ação. Caso o estado não seja terminal, através das informações referidas anteriormente e das previsões das duas redes, é possível calcular o valor Q utilizando a equação de Bellman (equação 3.4). Por sua vez, se o estado for terminal, o valor Q é igual à recompensa recebida pelo agente.

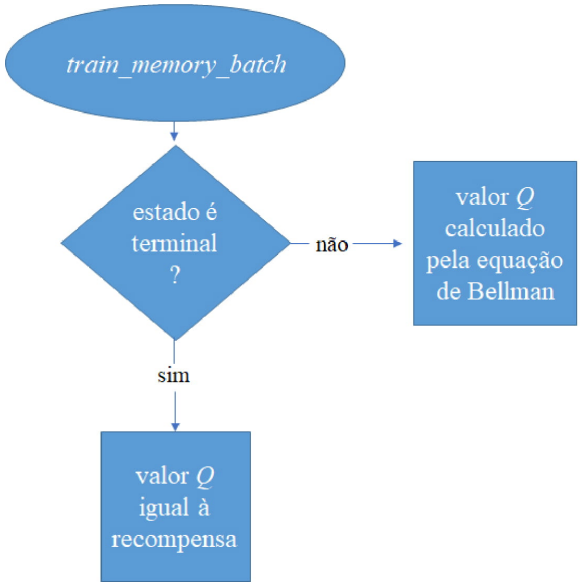


Figura 4.26: Fluxograma referente ao cálculo do valor Q, presente no método *train_memory_batch*.

O treino da rede principal foi realizado através do método *train_on_batch*, proveniente da biblioteca *keras*, onde as pilhas, correspondentes ao estado atual, presentes na *replay memory* auxiliar, são utilizadas como dados de treino, enquanto os valores Q calculados, juntamente com as respetivas ações, são utilizados como dados alvo, ou também denominados de *label* em inglês.

No método *train_memory_batch*, é possível averiguar que, quando uma ação é terminal, o valor Q correspondente é igual à recompensa obtida pelo agente. Contudo, no caso do estado terminal onde o agente acaba a sua produção, ou seja, alcança o seu objetivo, o valor Q necessita de ser maior do que o valor Q de uma ação terminal errada, de modo a que o agente se queira deslocar para a estação final correspondente à finalização da produção.

Foram averiguadas duas soluções para resolver este problema, sendo que a primeira

consiste em atribuir uma recompensa, muito superior às restantes, ao estado em que o agente alcança a estação final, após ter realizado os seus objetivos, para que o valor Q neste estado seja superior a todos os outros, sendo que neste caso, como o estado é terminal o valor Q só é igual à recompensa obtida, enquanto a segunda solução consiste em não considerar o estado mencionado como terminal, visto que, desta forma o valor Q é originado a partir da equação Bellman (equação 3.4). O valor Q é superior aos restantes, porque a recompensa dada é superior às restantes.

Nesta dissertação, é implementada a segunda solução, porque na primeira solução é necessário que a recompensa seja maior do que o maior valor Q, sendo este difícil de encontrar, visto que, os valores Q variam. Como tal, a segunda solução é mais fácil de implementar, pois qualquer recompensa superior às restantes, faz com que esta solução funcione.

Para implementar a segunda solução, foi criada uma segunda variável indicadora de estado terminal. Desta forma a primeira variável indicadora de estado terminal, denominada de *IS_OVER*, indica que o agente cometeu uma ação errada e a segunda variável indicadora de estado terminal, denominada de *IS_OVER_FINAL*, indica que o agente *Double-DQN* encontra-se num **Agente Estação Final** e como tal necessita de retornar ao **Agente Estação Inicial**, de modo a começar um novo ciclo de produção.

Por fim, o método *train_network*, presente na listagem I.28 (em anexo), foi desenvolvido, de modo a aglomerar os métodos supracitados. Primeiramente é averiguado se $\epsilon_{atual} > \epsilon_{final}$ e se $t > n_{observacao}$, de modo a verificar se o agente ainda se encontra em modo de exploração aleatória, ou não. Se o agente ainda se encontrar em modo de exploração aleatória, o valor de ϵ altera-se através da seguinte equação:

$$\epsilon = \epsilon - \frac{(\epsilon_{inicial} - \epsilon_{final})}{E} \tag{4.3}$$

De notar que a variável **E** na equação 4.3, corresponde ao número de iterações de exploração aleatória.

Em seguida são adquiridas as informações necessárias para popular a *replay memory*, sendo estas a ação escolhida pelo agente, a recompensa, se é um estado terminal, a pilha atual e a pilha com o estado seguinte. Após isto, é averiguado se $t > n_{observacao}$ e se for o caso, é chamado o método *train_memory_batch*, de modo a treinar a rede, sendo também averiguado se é necessário atualizar os pesos da rede alvo. Por fim, a pilha atual é igualada à pilha com o estado seguinte, de modo a atualizar a pilha atual. Na figura 4.27, encontra-se um fluxograma referente ao funcionamento genérico do método *train_network*.

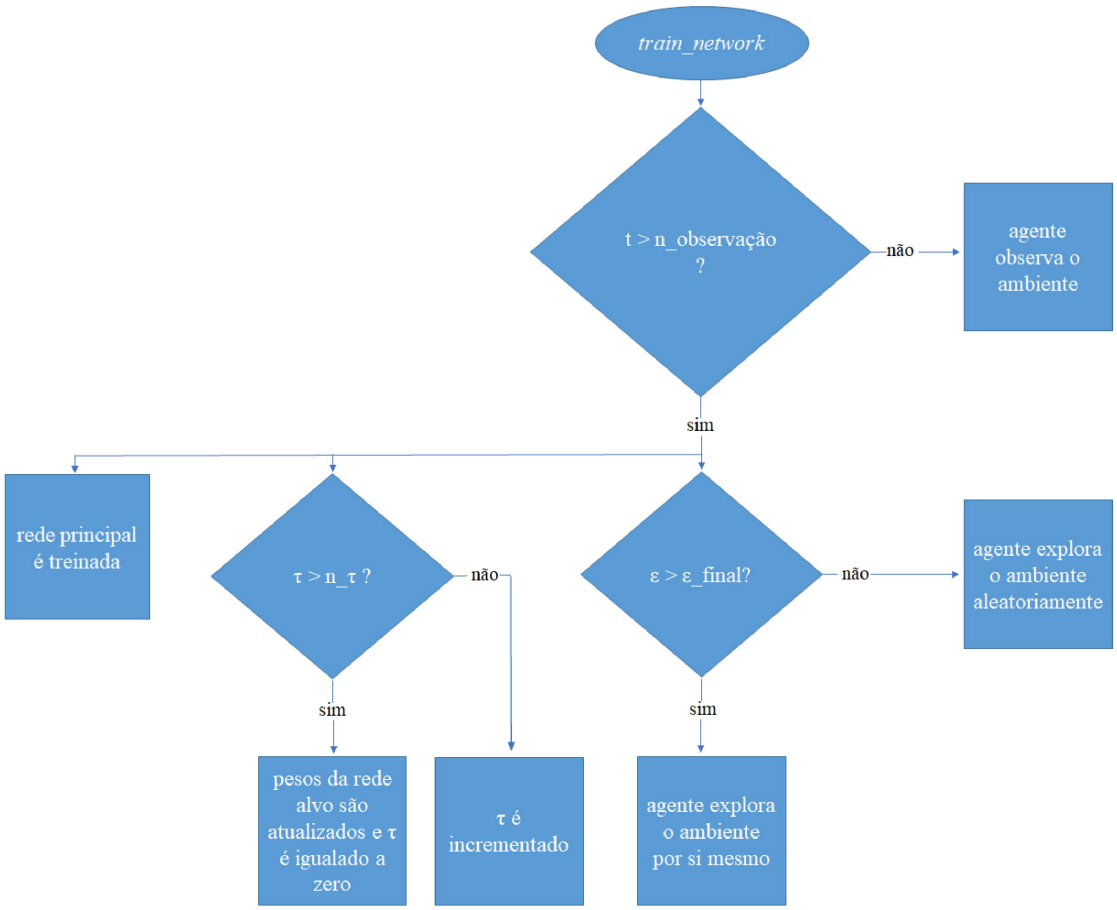


Figura 4.27: Fluxograma referente ao funcionamento do método *train_network*, relacionado com o treino do algoritmo *Double-DQN*.

Como é demonstrado na figura 4.2, a *framework Mesa* tem uma sequência própria de funcionamento, pelo que foi necessário distribuir os métodos referentes ao treino do agente pelas várias classes da simulação, ou seja, o método *init_train_network*, bem como o método *train_network*, foram colocados no método *step* da classe *model*, isto porque quando este método é executado já foi realizada a alteração visual correspondente à ação do agente, sendo assim possível adquirir a imagem do próximo estado. De notar que o método *init_train_network* só é executado na primeira iteração de treino. Por sua vez o método *find_max_q* foi colocado no método correspondente ao agente a ser treinado, de modo a alterar a posição do agente consoante a previsão da CNN.

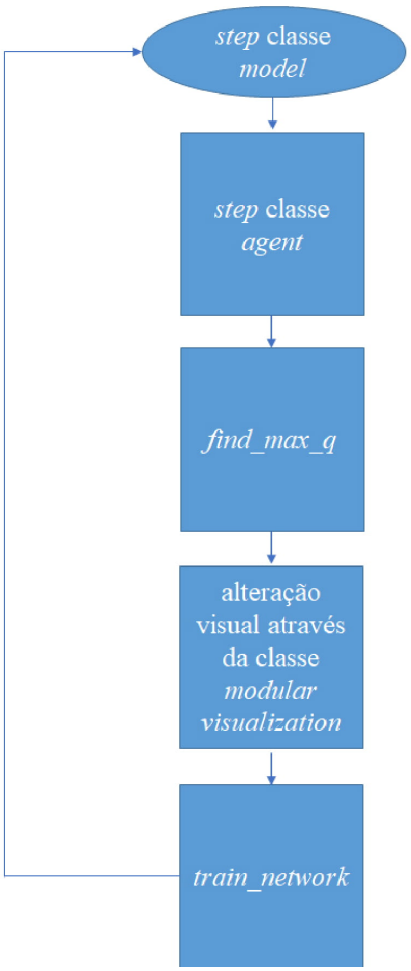


Figura 4.28: Fluxograma referente à ligação entre os diferentes métodos da simulação e do algoritmo *Double-DQN*.

Os métodos utilizados nos cenários de teste são semelhantes aos utilizados no treino, no entanto não utilizam o parâmetro ϵ, a *replay memory* e o método correspondente ao treino da rede.

4.3.2 Algoritmo *Q-learning*

Na **simulação 1**, o algoritmo *Q-Learning* é aplicado no ponto de decisão referente ao **Agente Produto Tipo 2**, agente *Q-learning*. Como referido no capítulo 3, é necessário construir uma tabela *Q* (tabela 4.13). Apesar de na **simulação 1** os **Agentes Produto** terem a possibilidade de se movimentarem em três direções diferentes, o agente *Q-learning* só tem a possibilidade de se movimentar em duas destas, ou seja, só pode escolher entre duas ações, sendo estas, movimentar-se para a **direita** ou para **baixo**. O agente *Q-learning* só tem acesso a estas duas ações para tornar o seu treino mais rápido, visto que, este só possui a capacidade de escolher uma ação na estação em que a sua decisão afeta o seu tempo de produção, sendo esta o ponto de decisão que corresponde ao **Agente Produto Tipo 2**.

4.3.2.1 Parâmetros e Recompensas - Simulação 1

Os diferentes estados presentes na tabela 4.13, correspondem às diferentes possibilidades em que os restantes **Agentes Produto Tipo 2** se podem encontrar, quando o agente *Q-learning* se encontra no ponto de decisão supracitado.

Tabela 4.13: Inicialização da tabela *Q* utilizada para treinar o agente *Q-learning* na simulação 1.

	Direita	**Baixo**
0	0	0
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0
6	0	0
7	0	0
8	0	0
9	0	0
10	0	0
11	0	0
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0

Os estados presentes na tabela 4.13 possuem o seguinte significado:

- ambas as transportadoras verticais, correspondentes aos **Agentes Produto Tipo 2**, vazias,
- ambos os **Agentes Produto Tipo 2** na primeira transportadora vertical ou ambos na segunda transportadora vertical,
- um dos **Agentes Produto Tipo 2** na primeira transportadora horizontal, atrás ou à frente do ponto de decisão, e o outro na primeira ou segunda transportadora vertical e vice-versa,
- um **Agente Produto Tipo 2** em cada uma das transportadoras verticais, onde se podem encontrar na mesma posição, relativamente ao eixo de coordenadas Y, ou

um dos **Agentes Produto** numa posição superior relativamente ao outro, tendo em conta o mesmo eixo de coordenadas, e vice-versa.

Como são estipulados, de uma forma genérica, os estados em que os agentes, não treinados, se podem encontrar, é possível deduzir qual das transportadoras verticais originam um caminho de produção mais rápido, sendo assim atribuída uma recompensa positiva, neste caso **um**, caso o agente escolha o caminho mais rápido num dado estado, ou uma recompensa negativa, de valor **menos um**, caso o agente escolha o caminho mais demorado num dado estado.

O agente treinou durante **100000 iterações**, onde **50000** foram focadas na fase de exploração aleatória do agente e as últimas **50000** foram focadas na fase de exploração por si próprio. No final do treino a tabela *Q*, presente em 4.13, fica preenchida com os valores *Q*, provenientes da equação 3.3, que melhor se aplicam a cada posição da tabela. A tabela *Q*, preenchida pelos valores *Q*, após o treino do agente *Q-learning*, encontra-se na tabela 4.14.

Tabela 4.14: Tabela *Q* preenchida após o treino do agente RL.

	Direita	**Baixo**
0	-0,1	0,1
1	1	-1
2	-1	1
3	-1	1
4	-1	1
5	-1	1
6	-1	1
7	1	-1
8	1	-1
9	-1	1
10	-1	1
11	0	0
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0

Tabela 4.15: Parâmetros de treino do algoritmo *Q-learning*

Parâmetro	Valor
Ações	[0, 1]
Número de Ações	2
$\epsilon_{inicial}$	1
ϵ_{final}	0,1
Exploração	50000
γ	0,6
α	0,1
Epochs	100000

Os parâmetros presentes na tabela 4.15, possuem o seguinte significado:

- **ações**, são representadas por um vetor que armazena as possíveis ações que o agente pode realizar, onde **zero** representa movimentar para baixo e **um** movimentar para a direita,
- $\epsilon_{inicial}$, representa o valor inicial da política ϵ-*greedy*, sendo este um ou 100%,
- ϵ_{final}, representa o valor final da política ϵ-*greedy*, sendo este 0,1 ou 10%,
- **exploração**, representa o valor que define o rácio da política ϵ-*greedy*, ou seja, quando o número de iterações alcança os **50000** significa que o valor de $\epsilon = 0,1$,
- γ, representa a taxa de desconto,
- α, representa a taxa de aprendizagem,
- ***epochs***, representam o número total de iterações realizadas no treino do agente *Q-learning*, sendo que neste caso, nas primeiras **50000** iterações o agente encontra-se numa fase de exploração aleatória, enquanto nas últimas **50000** iterações o agente encontra-se numa fase de exploração por si próprio.

4.3.2.2 Implementação do Algoritmo *Q-learning*

O algoritmo *Q-learning* utiliza uma tabela, denominada tabela *Q*, onde as linhas correspondem aos diferentes estados do ambiente e as colunas correspondem às ações que o agente pode tomar nesses mesmos estados. Com isto, é necessário averiguar quais os estados possíveis presentes na **simulação 1**, de modo a construir a tabela. Porém, apesar do ambiente ser simples, este apresenta inúmeros estados, pois existem inúmeras combinações de posições dos três **Agentes Produto Tipo 2**. Como tal, foram considerados alguns estados genéricos, presentes na sub-subsecção 4.3.2.1. A tabela *Q* foi criada através da biblioteca *numpy* e encontra-se na listagem 4.3.

Listagem 4.3: Criação da tabela Q

```
1 import numpy as np
2
3 observation_space = 17
4 action_space = 2
5 q_table = np.zeros([observation_space, action_space])
```

A implementação do algoritmo *Q-learning* utiliza métodos muito semelhantes ao descritos na sub-subsecção 4.3.1.4, como por exemplo o método *init_train_network*. Porém, apesar dos métodos *find_max_q* e *train_network* serem muito similares aos presentes no algoritmo *Double-DQN*, estes apresentam algumas diferenças.

No método *find_max_q*, utilizado no algoritmo *Q-learning*, o agente prevê a ação baseando-se no maior valor Q presente num certo estado da tabela, tal como presente na listagem I.29 (em anexo).

Por sua vez, o método *train_network* utilizado no algoritmo *Q-learning* apresenta duas diferenças comparado ao método *train_network* utilizado no algoritmo *Double-DQN*. A primeira diferença consiste na utilização de uma equação de Bellman diferente, sendo que no algoritmo *Double-DQN* é utilizada a equação 3.5, enquanto no algoritmo *Q-learning* é utilizada a equação 3.3. A segunda diferença encontra-se no facto de que as informações necessárias para a equação de Bellman são obtidas através da tabela.

Tal como no algoritmo *Double-DQN*, o método *train_network* é chamado no método *step* da classe *model*, enquanto o método *find_max_q* é chamado no método *step* correspondente ao agente a ser treinado. Tal como referido na sub-subsecção 4.3.1.4, os métodos usados nos cenários de testes são muito semelhantes aos utilizados durante o treino.

4.3.3 Algoritmo Heurístico *A**

O algoritmo heurístico *A** não necessita de treino, como tal só é necessário desenvolver um método que cumpra os requisitos presentes na subsecção 3.1.3.

4.3.3.1 Implementação do Algoritmo Heurístico *A**

O agente *A** necessita de ter conhecimento acerca do ambiente, de modo a calcular o seu trajeto, como tal a **simulação 1**, neste algoritmo, é representada através de uma matriz, em que cada posição da matriz necessita de corresponder a cada posição da simulação. Porém, as posições de uma matriz criada pela biblioteca *numpy* não correspondem às posições da simulação, ou seja, a origem, ponto **(0, 0)** da simulação encontra-se no canto inferior direito, enquanto a origem da matriz encontra-se no canto superior esquerdo. Neste sentido, foi necessário averiguar as posições da matriz que correspondiam às posições da simulação e dessa forma construí-la.

A matriz referente à **simulação 1** é a que se encontra na listagem 4.4. De notar que as posições com o valor **três** correspondem às posições onde o agente não se pode movimentar, enquanto as posições com o valor **zero** correspondem às posições onde o agente

se pode movimentar.

A abordagem do algoritmo *A**, utilizada nesta dissertação, não tem em conta as posições dos restantes agentes, como tal este vai sempre escolher o caminho mais curto, o que não significa que escolha o mais rápido.

Listagem 4.4: Matriz referente ao ambiente de simulação 1

```
sim_1 =  [[3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
          [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
          [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
          [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
          [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
          [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [3, 0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0],
          [3, 0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0],
          [3, 0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0],
          [3, 0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0],
          [3, 0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0],
          [3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]]
```

Em seguida, foi criada uma classe denominada de **nó**, que permite armazenar a posição, o tipo, neste caso *parent* em inglês, e os valores de **G**, **H** e **F**, de cada nó.

Após a criação da classe **nó**, foi desenvolvido um método, denominado de *a_star*, que permite aplicar o algoritmo *A** e calcular o caminho mais curto, desde o ponto (6, 12), primeiro ponto de decisão correspondente ao **Agente Produto Tipo 3**, ao ponto (6, 0), correspondente à posição do **Agente Estação Final**.

Neste método foram criadas duas listas, a **lista aberta**, correspondente aos nós a testar, e a **lista fechada**, correspondente aos nós referentes ao caminho mais curto. Primeiramente, foi necessário averiguar qual o nó presente na lista aberta, que possui um menor valor de **F** e adicioná-lo à lista fechada. Em seguida, averiguaram-se os nós adjacentes e confirmaram-se as suas posições, ou seja, se estão numa posição interdita ou não. O nó que não estiver numa posição interdita, é adicionado a uma lista de nós secundários, onde por sua vez são calculados os valores **G**, **H** e **F** destes. Posteriormente, os nós secundários são adicionados à lista aberta, onde é feita, novamente, a comparação para analisar qual o nó com o menor valor de **F**. Por fim, quando a posição do nó atual é igual à posição do nó final, é construída uma lista que possui as posições que levam ao caminho mais curto, através da posição dos nós presentes na lista fechada.

Posteriormente, o **Agente Produto Tipo 3** utiliza as coordenadas presentes na lista correspondente ao caminho, para se movimentar no ambiente de simulação.

Foi escolhido o algoritmo heurístico *A**, pois após uma pesquisa na literatura foi comprovado que este algoritmo é utilizado para problemas de encaminhamento, [40], [16]. Foi também concluído que existem inúmeros algoritmos heurísticos para este propósito,

no entanto como o foco desta dissertação é o algoritmo *Double-DQN* e o objetivo do algoritmo heurístico é meramente comparativo, foi escolhido o algoritmo heurístico *A**.

Capítulo 5

Resultados

Neste capítulo é apresentado, e comparado, o desempenho dos algoritmos propostos nesta dissertação. Enquanto os agentes *Double-DQN* e *Q-learning* foram avaliados tanto na fase de treino, como na fase de teste, o algoritmo heurístico *A** só foi avaliado durante a fase de teste, visto que, não necessita de treino.

O treino dos agentes *Double-DQN* e *Q-learning*, bem como todos os cenários de teste, foram realizados num computador com as seguintes caraterísticas:

- Processador Intel Core I7-4790 3.6 GHz,
- Placa gráfica Nvidia Geforce GTX 970,
- 16 GB de *RAM*,
- Arquitetura de 64 bits.

De notar que relativamente ao treino de uma rede neuronal, a utilização de uma placa gráfica dedicada é útil, visto que, assim é possível diminuir o tempo de treino.

Estes dados servem para salientar que os treinos e os cenários de teste, foram realizados num computador de gama média, com cerca de cinco anos, sendo que num computador atual, tanto os treinos como os cenários de teste, seriam realizados em menor tempo.

O tempo de cada treino não é necessariamente o mais otimizado, ou seja, como o objetivo desta dissertação é otimizar políticas de encaminhamento, não é tido em conta a otimização do treino dos respetivos algoritmos, sendo que podem ter ficado a treinar mais tempo que o necessário para garantir a otimização das respetivas políticas. Como tal, através da observação dos gráficos presentes neste capítulo, é possível averiguar o número aproximado de iterações em que os agentes aprenderam a política e, desta forma, otimizar o tempo de treino de cada algoritmo.

No treino dos agentes *Double-DQN*, é concluído que, no computador utilizado, o seu desempenho era inversamente proporcional ao tamanho da *replay memory*, ou seja, quanto maior era a *replay memory*, menor era o desempenho do computador, diminuindo a velocidade da simulação. Sendo assim, é necessário arranjar um tamanho equilibrado para a *replay memory*, de modo a que os agentes tenham acesso ao número de experiências necessárias, não afetando o desempenho do computador.

5.1 Métricas e Cenários de Teste

Na primeira simulação, são utilizados quatro cenários de teste distintos, onde cada algoritmo é testado durante **1000 ciclos de produção**[1]. Nos primeiros três cenários de teste, os algoritmos inteligentes controlam o **Agente Produto Tipo 4** presente em cada cenário. Nestes cenários os dois **Agentes Produto** que não são treinados, possuem as seguintes ações programadas:

- no primeiro cenário, ambos os **Agentes Produto** são encaminhados para a segunda transportadora vertical do seu tipo,
- no segundo cenário, ambos os **Agentes Produto** são encaminhados para a primeira transportadora vertical do seu tipo,
- no terceiro cenário, cada **Agente Produto** é encaminhado para uma transportadora vertical distinta.

É importante referir que o algoritmo heurístico A^* só é testado nos cenários de teste **um**, **dois** e **quatro**, visto que, como este não tem em conta os restantes agentes presentes na simulação, o resultado deste algoritmo no cenário de teste **três**, é igual ao resultado adquirido no cenário de teste **dois**, pois é sempre escolhido o caminho mais curto, ou seja, a primeira transportadora vertical do seu tipo.

O quarto cenário consiste em aplicar os três algoritmos nos **nove agentes** e avaliar a sua prestação através da média de pontos de cada tipo de agente, ou seja, avaliar o tempo médio que cada tipo de produto demorou a ser produzido. Neste cenário, a inteligência é distribuída pelos diferentes **Agentes Produto** de cada tipo, como por exemplo, no caso dos agentes que são testados com o algoritmo *Double-DQN*, é utilizado o conceito de transferência de conhecimento, onde cada um tem a sua rede neuronal, no entanto todos utilizam os pesos provenientes do treino do primeiro agente *Double-DQN*.

Por sua vez, na segunda simulação são criados dois cenários de teste, onde o algoritmo *Double-DQN* é testado durante **1000 ciclos de produção**.

As métricas utilizadas relativamente ao desempenho dos algoritmos *Double-DQN* e *Q-learning* durante o treino, na **simulação 1**, são as seguintes:

[1] É considerado um ciclo de produção, quando um agente chega à estação final, após passar por todos os recursos necessários pela ordem correta.

- o somatório de todas as recompensas ao longo do treino,
- o valor médio de pontos, ou seja, tempo médio de produção.

É importante referir que os pontos (tempo de produção) dos agentes só são armazenados quando estes alcançam o **Agente Estação Final**, perfazendo um ciclo de produção. De notar que as métricas utilizadas para avaliar o desempenho do algoritmo *Double-DQN* durante o seu treino na **simulação 2**, são iguais às métricas supracitadas.

Por outro lado, nos cenários de teste da primeira simulação é utilizada uma única métrica, sendo esta a média de pontos que cada agente fez durante o teste, ou seja, o tempo médio que cada produto demorou a ser produzido. Esta métrica é suficiente, porque através da sua análise é possível averiguar e comparar o tempo que cada agente permaneceu na linha de produção e assim concluir se cada algoritmo conseguiu encontrar a política de encaminhamento mais otimizada para cada situação.

No caso da segunda simulação, a métrica utilizada para averiguar o desempenho do agente *Double-DQN*, durante os cenários de teste, é a sua pontuação, ou seja, o tempo de produção.

5.2 Simulação 1

O objetivo da **simulação 1** é comparar o desempenho do algoritmo *Double-DQN* com os algoritmos *Q-learning* e A^*, no que toca à otimização de políticas de encaminhamento dos produtos presentes nesta simulação.

5.2.1 Treino

Os agentes *Double-DQN* e *Q-learning*, treinaram durante o mesmo número de iterações, presentes na tabela 4.10 e na tabela 4.15, sendo que cada treino demorou cerca de **14 horas**. É importante referir que esta simulação demorou mais tempo a treinar, visto que, nem todas as ações do agente contavam como iterações de treino, como já foi referido anteriormente.

A complexidade do treino do agente *Double-DQN* é superior à complexidade do treino do agente *Q-learning*, porque o número de ações que o agente *Double-DQN* pode realizar é superior ao número de ações que o agente *Q-learning* pode realizar.

5.2.1.1 Recompensas Acumuladas

Os algoritmos que utilizam a equação de Bellman são algoritmos gananciosos, ou seja, tentam encontrar a política que permite acumular a maior quantidade de recompensas positivas.

É possível observar este comportamento através dos gráficos, presentes na figura 5.1, visto que, as recompensas acumuladas pelos agentes tendem a aumentar à medida que o treino avança, apresentando um formato quase exponencial, sendo possível analisar o

instante onde o agente aprendeu a melhor política e começou a adquirir muito mais recompensas positivas do que negativas. Através deste comportamento também é possível concluir que ambos os agentes aprenderam a melhor política para acumular recompensas, ou seja, ambos os agentes encontraram a melhor política de otimização de encaminhamento do respetivo produto, consoante a função recompensa utilizada em cada treino.

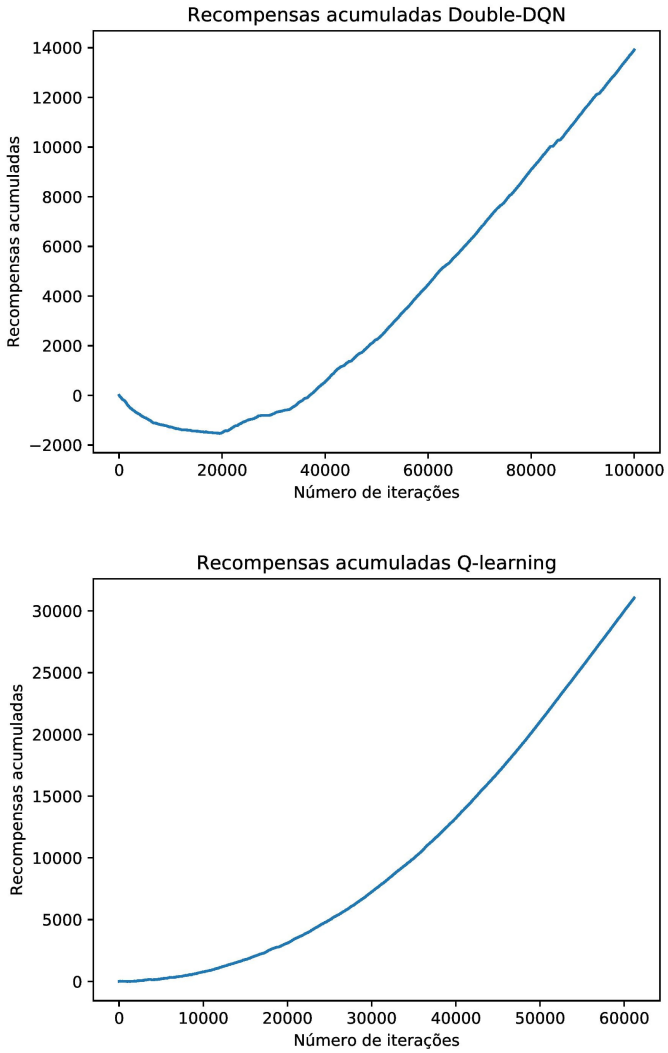


Figura 5.1: Recompensas acumuladas, pelos agentes *Q-learning* e *Double-DQN*, durante o treino na simulação 1.

É importante referir que os gráficos, presentes na figura 5.1, apresentam um número de iterações diferentes, visto que, o agente *Double-DQN* começa a tomar decisões a partir do **Agente Estação Inicial**, enquanto o agente *Q-learning* só toma decisões no primeiro ponto de decisão do seu tipo, como tal para as mesmas iterações de treino, o agente *Double-DQN* toma mais decisões do que o agente *Q-learning*.

As recompensas acumuladas por ambos os agentes são diferentes, isto porque são utilizadas diferentes funções recompensa no treino de cada algoritmo. A utilização de

diferentes funções recompensa deve-se ao facto de que o agente *Double-DQN* tem em conta todas as possibilidades de encaminhamento, visto que, visualiza os diferentes estados do ambiente através de imagens. Porém, como referido anteriormente, a **simulação 1** apresenta inúmeras possibilidades de posições onde cada agente pode estar, como tal a construção de uma tabela Q com todas essas possibilidades demoraria muito tempo. Como tal, foi necessário desenvolver uma função recompensa diferente que só tem em conta os casos presentes na tabela Q, desenvolvida para o treino do agente *Q-learning*.

5.2.1.2 Tempo Médio de Produção

A média dos pontos acumulados pelos agentes, ou seja, a média do tempo de produção de cada produto, é a métrica mais importante nesta dissertação, visto que, através desta é possível averiguar se cada algoritmo conseguiu encontrar a melhor política de encaminhamento. É importante referir que, na **simulação 1**, o menor número de pontos que cada agente pode ter é **53 pontos**, ou seja, o tempo de produção mais rápido é de **53 segundos**, sendo que isto acontece quando o produto entra na primeira estação vertical do seu tipo, enquanto esta está vazia.

O número de iterações dos gráficos, presentes na figura 5.2, é diferente, visto que, como o agente *Double-DQN* começa o seu treino no **Agente Estação Inicial**, este pode cometer ações incorretas, como por exemplo sair da linha de produção, sendo que quando isto acontece, este é enviado novamente para o **Agente Estação Inicial** e, por esta forma, não chegou ao **Agente Estação Final** em todas as iterações, ao contrário do agente *Q-learning*.

Como é possível observar nos gráficos, presentes na figura 5.2, ambos os agentes conseguiram minimizar o tempo de produção, encontrando a política de encaminhamento mais otimizada para a **simulação 1**, consoante as suas caraterísticas. É também possível observar que o agente *Double-DQN* obteve uma melhor pontuação média, sendo que rapidamente entendeu qual o melhor caminho a escolher em cada situação, de modo a minimizar o tempo de produção.

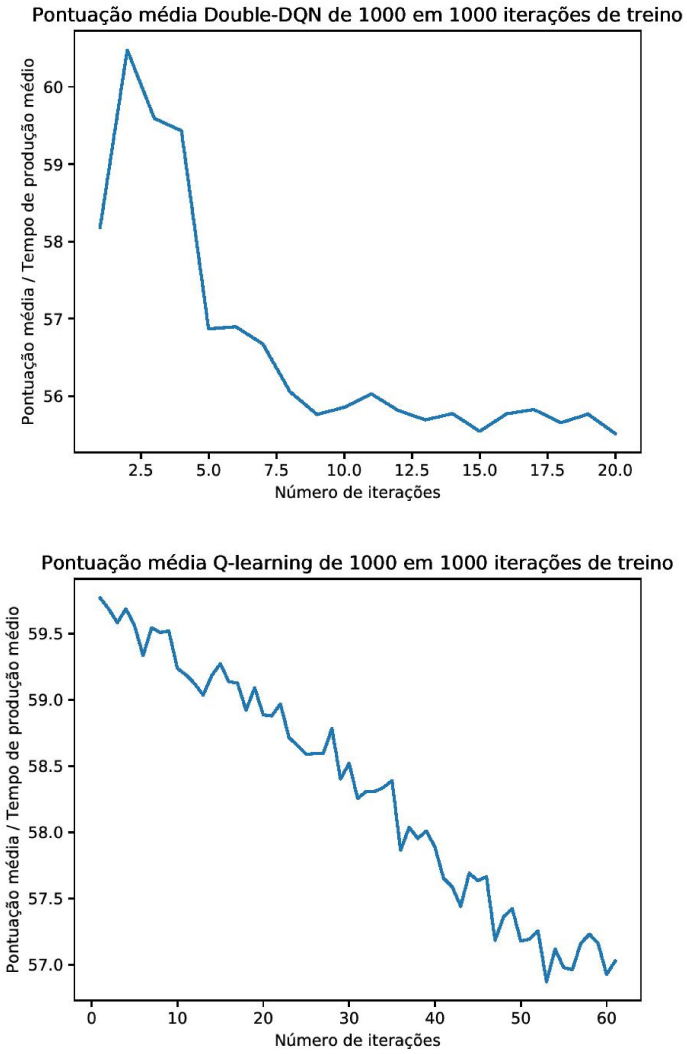


Figura 5.2: Média de pontos, de 1000 em 1000 iterações, realizados pelos agentes *Q-learning* e *Double-DQN*, durante o treino na simulação 1.

5.2.2 Cenários de Teste

Na **simulação 1** são realizados quatro cenários de teste, sendo que nos três primeiros os algoritmos são avaliados de forma individual, enquanto no quarto todos os agentes presentes na simulação são controlados pelos respetivos algoritmos.

5.2.2.1 Cenário de Teste 1

No cenário de teste 1, os dois **Agentes Produto** de cada tipo que não foram treinados encontram-se programados para escolher a segunda transportadora vertical do seu tipo, sendo que o comportamento ótimo para o agente a ser testado é escolher a primeira transportadora vertical do seu tipo.

Nos gráficos, presentes na figura 5.3, é possível observar a pontuação média, de 50 em

50 iterações, dos agentes correspondentes aos diferentes algoritmos. Através da análise destes gráficos, é possível concluir que tanto o algoritmo *Double-DQN*, como o algoritmo *Q-learning*, obtiveram uma média aproximada de **53 pontos**, ou seja, **53 segundos**, enquanto o agente *A** obteve uma média de **54 pontos**. Analisando estes valores, chega-se à conclusão de que os três agentes, correspondentes a cada algoritmo, decidiram escolher a primeira transportadora vertical do seu tipo.

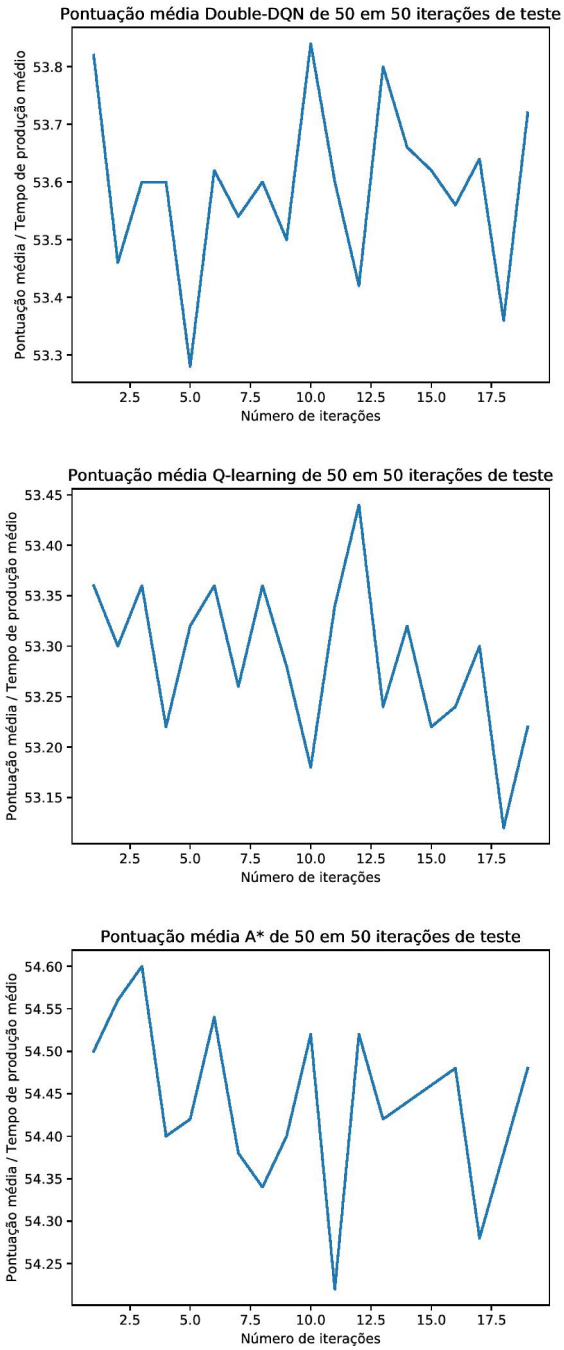


Figura 5.3: Média de pontos, de 50 em 50 iterações, realizados pelos agentes *Double-DQN*, *Q-learning* e *A**, no cenário de teste 1 na simulação 1.

5.2.2.2 Cenário de Teste 2

No cenário de teste 2, os dois **Agentes Produto** de cada tipo que não foram treinados, encontram-se programados para escolher a primeira transportadora vertical do seu tipo, sendo que, em maior parte dos casos, o comportamento ótimo para o agente a ser treinado é escolher a segunda transportadora vertical do seu tipo.

Tal como já foi explicado anteriormente, enquanto os algoritmos *Double-DQN* e *Q-learning* têm em conta a posição dos outros agentes, o algoritmo *A** não, pelo que vai sempre escolher o caminho mais curto, sendo este a primeira transportadora vertical, e não o mais rápido.

Neste cenário de teste aconteceu algo inesperado. O agente *Q-learning* escolheu sempre a segunda transportadora vertical do seu tipo, como sendo o caminho mais rápido, visto que, os outros dois **Agentes Produto** do seu tipo deslocaram-se sempre para a primeira transportadora vertical do seu tipo. O agente *Q-learning* teve este comportamento, pois como já referido anteriormente, os estados presentes na tabela *Q* são genéricos e foi deduzido que sempre que dois **Agentes Produto** do seu tipo encontravam-se na primeira transportadora vertical do seu tipo, o caminho mais rápido seria escolher a segunda transportadora vertical do seu tipo.

Porém, no caso específico onde ambos os **Agentes Produto** não inteligentes encontram-se na primeira transportadora vertical, mas um deles encontra-se no último recurso ou na posição seguinte a esse recurso, o caminho mais rápido é obtido escolhendo a primeira transportadora vertical e não a segunda. De notar que este caso específico foi observado após a análise do comportamento do agente *Double-DQN* no cenário de teste 2, bem como a observação do gráfico, presente na figura 5.4.

Como o agente *Double-DQN* aprendeu através de imagens, tendo acesso a inúmeros estados do ambiente, e a sua função recompensa é dependente do número de pontos que este obtém, no caso específico supracitado este optou por escolher a primeira transportadora vertical do seu tipo, pois esta escolha é a que permite obter um menor tempo de produção. Como tal, o agente *Double-DQN* apresenta uma média menor de pontos, comparativamente aos restantes agentes inteligentes, visto que, em certos casos específicos, conseguiu entender que apesar dos dois **Agentes Produto** do seu tipo encontrarem-se na primeira transportadora vertical, o caminho mais rápido seria escolher esta transportadora e não a segunda, enquanto nos outros casos este decidiu escolher a segunda transportadora vertical, visto que, era a que permitia uma produção mais rápida.

Por sua vez, o agente *A** é o que apresenta um maior tempo de produção, visto que, não tem em conta a posição dos restantes agentes, escolhendo sempre o caminho mais curto e não o mais rápido.

Através deste cenário de teste é possível demonstrar a versatilidade do agente *Double-DQN* na otimização do encaminhamento de produtos na **simulação 1**, visto que, aprendeu que apesar de se encontrarem dois produtos na primeira estação vertical, não significa que o caminho mais rápido seja a estação vertical vazia, ou seja, a segunda.

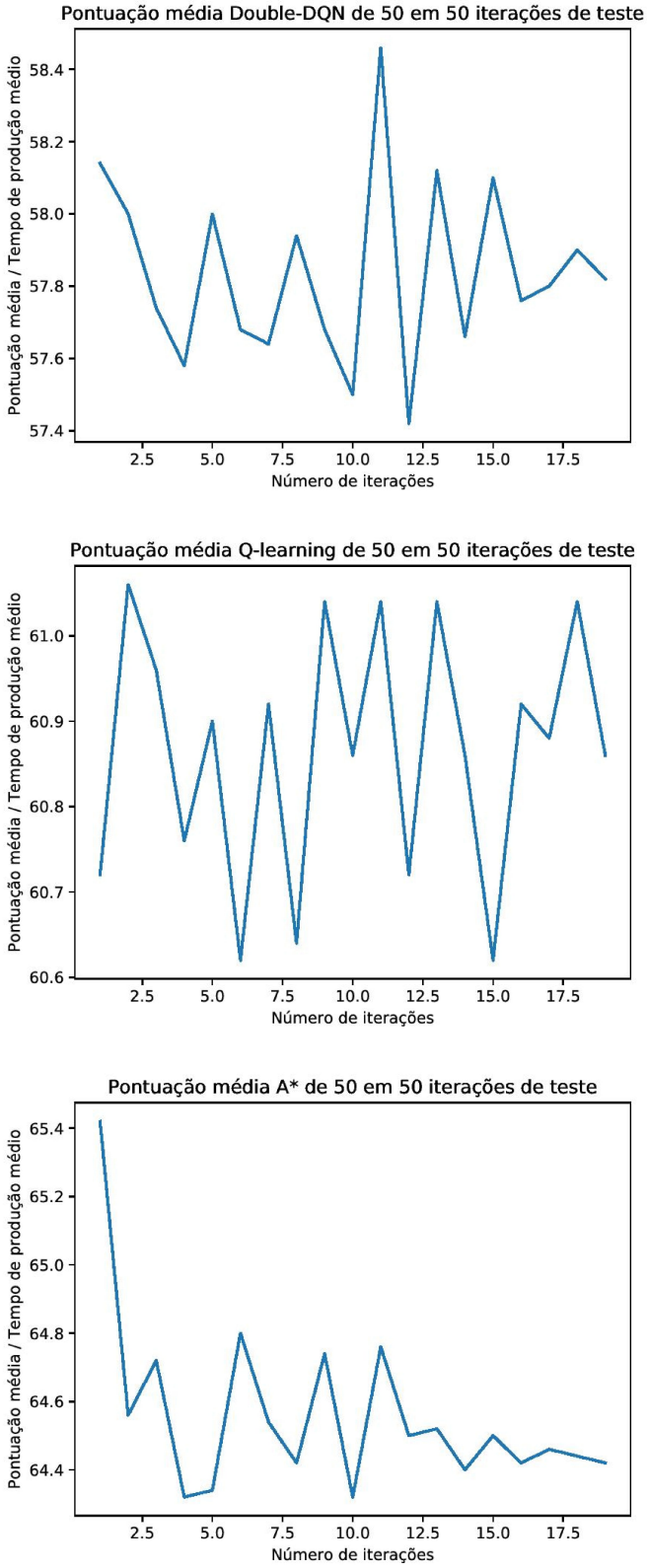


Figura 5.4: Média de pontos, de 50 em 50 iterações, realizados pelos agentes *Double-DQN*, *Q-learning* e *A**, no cenário de teste 2 na simulação 1.

5.2.2.3 Cenário de Teste 3

No cenário de teste 3, os dois **Agentes Produto** de cada tipo que não estavam a ser treinados, encontram-se programados para escolher uma das duas opções possíveis, ou seja, um dos agentes não inteligentes escolheu a primeira transportadora vertical do seu tipo, enquanto o outro escolheu a segunda. É importante relembrar que neste cenário de teste, o agente *A** não foi utilizado, visto que, como explicado anteriormente, o resultado seria igual ao resultado presente no cenário de teste anterior.

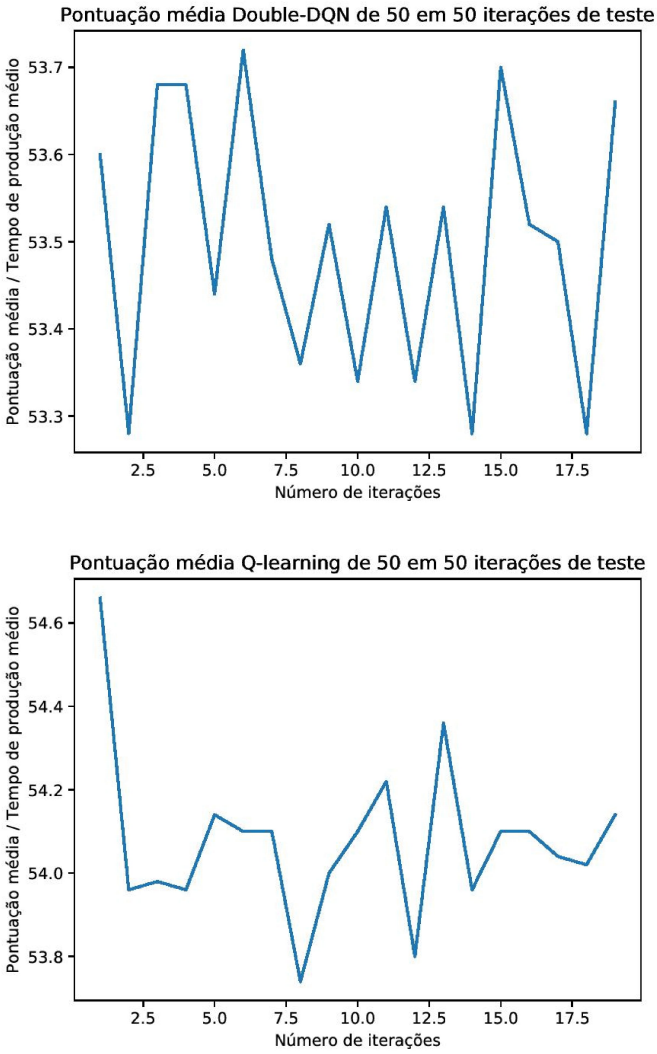


Figura 5.5: Média de pontos, de 50 em 50 iterações, realizados pelos agentes *Double-DQN* e *Q-learning*, no cenário de teste 3 na simulação 1.

Através dos gráficos, presentes na figura 5.5, é possível concluir que ambos os agentes conseguiram encontrar o encaminhamento mais otimizado, relativamente a este cenário de teste, sendo que a diferença entre a média dos dois é muito pequena.

5.2.2.4 Cenário de Teste 4

No cenário de teste 4, os três algoritmos propostos nesta dissertação são aplicados a todos os **Agentes Produto** do seu tipo, ou seja, os **nove agentes** produto presentes na **simulação 1** encontram-se controlados pelo respetivo algoritmo inteligente.

Durante o treino dos agentes *Double-DQN* e *Q-learning*, cada agente só controlou um único **Agente Produto**, porém neste cenário utilizou-se transferência de conhecimento pelos restantes **Agentes Produto** de cada tipo, ou seja, no caso dos agentes *Double-DQN* cada um possui uma CNN individual, no entanto todas utilizam os mesmos pesos provenientes do treino do primeiro agente. Assim é possível concluir que nas abordagens apresentadas nesta dissertação basta treinar um só agente *Double-DQN* ou *Q-learning*, num ambiente, em modo *offline*, e através da transferência de conhecimento, aplicar a política adquirida durante o treino para **N Agentes Produto**.

Nos gráficos, presentes na figura 5.6, é possível observar que os agentes A^* são os que apresentam a maior média de pontuação, ou seja, a pior política de encaminhamento, isto porque todos os agentes escolheram o caminho mais curto, ou seja, a primeira transportadora vertical.

De notar que, caso o **Agente Produto** seja o primeiro do seu tipo no primeiro ponto de decisão do seu tipo, e escolha a primeira transportadora vertical do seu tipo, este adquire **53 pontos**, enquanto se por alguma razão, este tiver de escolher a segunda transportadora vertical, e esta se encontrar vazia, o agente adquire **57 pontos**.

Os agentes *Double-DQN* e *Q-learning*, conseguiram aprender a melhor política de encaminhamento para a **simulação 1**, visto que, a média dos três agentes *Double-DQN* e dos três agentes *Q-learning*, não é superior a **57 pontos**, isto significa que dois **Agentes Produto** optaram escolher a primeira transportadora vertical do seu tipo, visto que, era o caminho mais rápido, enquanto o terceiro agente optou por escolher a segunda transportadora vertical do seu tipo, pois é produzido mais rapidamente indo para esta, do que esperando na primeira transportadora vertical. Porém, o agente *Double-DQN* possui a vantagem de ser mais versátil e não necessitar de intervenção humana, no que toca a identificar os diferentes estados do ambiente.

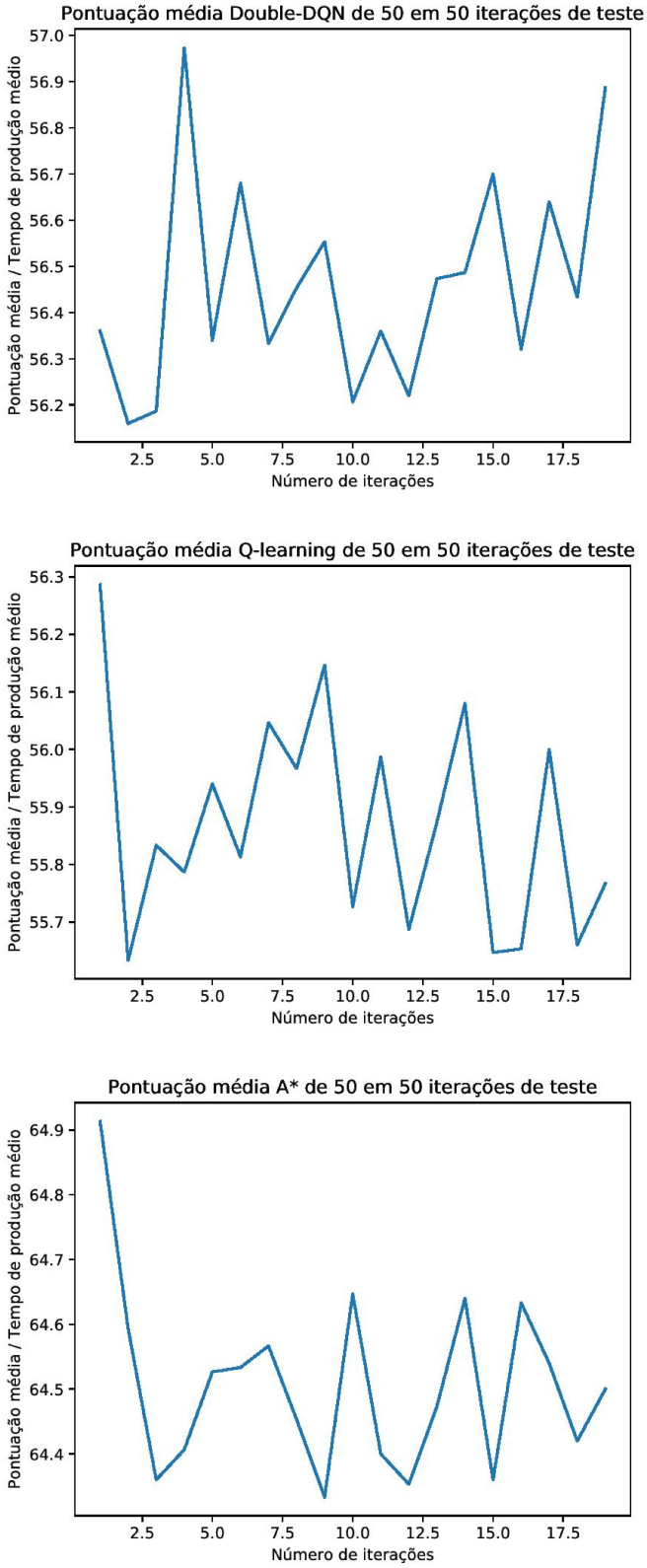


Figura 5.6: Média de pontos, de 50 em 50 iterações, realizados pelos agentes *Double-DQN*, *Q-learning* e *A**, no cenário de teste 4 na simulação 1.

5.3 Simulação 2

A **simulação 2** é um *digital twin* de um caso real, cujo objetivo é demonstrar que o algoritmo *Double-DQN* é genérico e fácil de implementar num ambiente de produção diferente.

O algoritmo utilizado nesta simulação é semelhante, ao utilizado na simulação anterior, tirando o facto de que esta permite que o agente realize **quatro** ações diferentes, em vez de **três**.

O desenvolvimento do algoritmo *Double-DQN*, utilizado nesta simulação, foi rápido e fácil, visto que, o seu código é semelhante ao código do algoritmo utilizado na **simulação 1**, com a diferença de que a CNN utilizada nesta versão possui um *output* igual a **quatro**, em vez de **três**, porque a **simulação 2** permite que o agente *Double-DQN* realize **quatro** ações. De notar, que houve uma alteração no método responsável pela escolha das ações, porém esta não é obrigatória, pelo que foi realizada de modo a diminuir o tempo de treino do algoritmo. Nesta simulação, o agente *Double-DQN* treinou durante cerca de **8 horas**.

5.3.1 Treino

Nesta subsecção, encontram-se os gráficos construídos através dos valores adquiridos durante o treino do agente *Double-DQN* na **simulação 2**.

Uma das métricas mais importante na avaliação de algoritmos de RL e DRL, é a recompensa acumulada ao longo do treino, visto que, através da análise do gráfico da recompensa acumulada, é possível averiguar se o agente em questão, encontrou a política de recompensas mais otimizada. Através da análise do gráfico presente na figura 5.7, é possível concluir a afirmação supracitada.

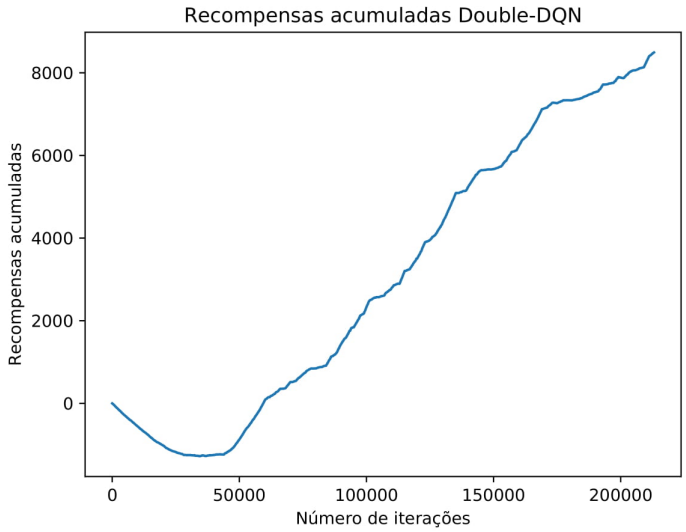


Figura 5.7: Recompensa acumulada, pelo agente *Double-DQN*, durante o treino na simulação 2.

Como referido na sub-subsecção 4.2.2.3, nesta simulação o agente *Double-DQN* pode obter duas pontuações, que indicam um encaminhamento eficiente do produto, sendo estas **28 pontos** ou **33 pontos**, consoante a posição do **Agente Produto Tipo 2**. Através da análise do gráfico presente na figura 5.8, é possível averiguar que a média dos pontos, adquiridos pelo agente, encontra-se entre os **30 pontos** e os **70 pontos**, dando a concluir que o agente encontrou os encaminhamentos ótimos, porém por vezes realizou mais pontos, devido à política de exploração pelo ambiente.

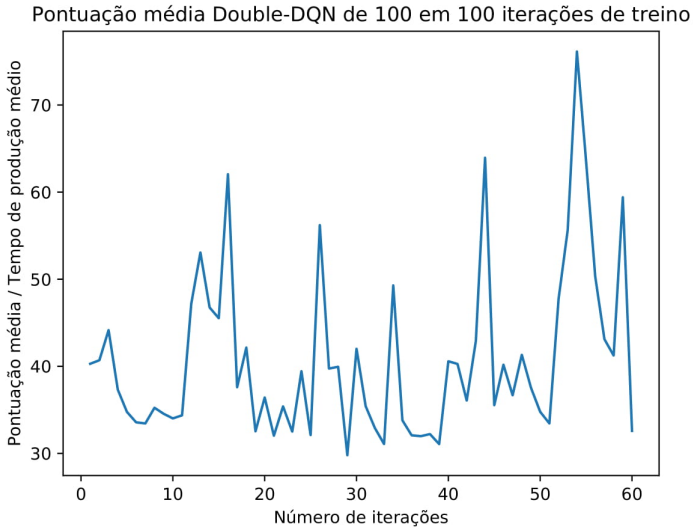


Figura 5.8: Média de pontos, de 100 em 100 iterações, realizados pelo agente *Double-DQN*, durante o treino na simulação 2.

5.3.2 Cenários de Teste

Como referido na sub-subsecção 4.2.2.3 na **Simulação 2**, o agente *Double-DQN* tem duas opções de encaminhamento, consoante a posição do **Agente Produto Tipo 2**, sendo estas:

- caso o agente *Double-DQN* seja o primeiro a sair da estação inicial, a opção de encaminhamento mais rápida é deslocar-se ao **Agente Recurso Tipo A** e, seguidamente, ao **Agente Recurso Tipo D**, perfazendo **28 pontos**,
- caso o agente *Double-DQN* seja o segundo a sair da estação inicial, e o **Agente Produto Tipo 2** se encontre no **Agente Recurso Tipo A**, a opção de encaminhamento mais rápida é deslocar-se ao **Agente Recurso Tipo D** e, seguidamente, ao **Agente Recurso Tipo C**, perfazendo **33 pontos**.

Sendo assim, nesta simulação são realizados dois cenários de teste, de modo a testar os casos acima referidos, e averiguar se o agente *Double-DQN* aprendeu a distinguir cada um dos casos e a escolher o melhor encaminhamento em cada um deles.

O primeiro cenário de teste representa o primeiro caso mencionado, como tal o agente *Double-DQN* é o primeiro a entrar na linha de produção. Por sua vez, o segundo cenário de teste representa o segundo caso supracitado, como tal o agente *Double-DQN* é o segundo a entrar na linha de produção.

5.3.2.1 Cenário de Teste 1

No gráfico presente na figura 5.9, é possível verificar que durante as **1000 iterações** deste cenário de teste, o agente *Double-DQN* realizou sempre o encaminhamento ótimo de produção, ou seja, tirando o pico inicial, este deslocou-se, primeiramente, para o **Agente Recurso Tipo A**, seguindo para o **Agente Recurso Tipo D**, utilizando sempre o caminho mais rápido possível. Desta forma, conclui-se que, neste cenário de teste, o agente *Double-DQN* escolheu a política ótima de encaminhamento, permitindo com que o produto fosse produzido no menor tempo possível, deste cenário, ou seja, **28 segundos**.

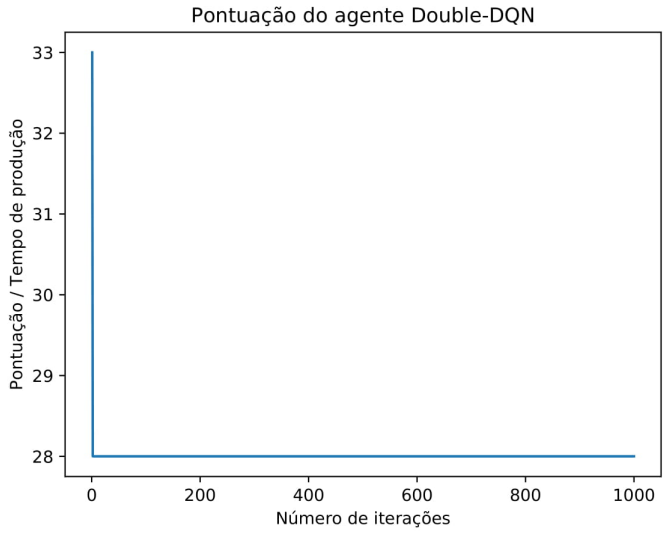


Figura 5.9: Pontuação realizada pelo agente *Double-DQN*, durante o cenário de teste 1 na simulação 2.

5.3.2.2 Cenário de Teste 2

No gráfico presente na figura 5.10, é possível verificar que durante as **1000 iterações** deste cenário de teste, o agente *Double-DQN* realizou sempre o encaminhamento ótimo de produção, ou seja, este deslocou-se, primeiramente, para o **Agente Recurso Tipo D**, seguindo para o **Agente Recurso Tipo C**, utilizando sempre o caminho mais rápido possível. Desta forma, conclui-se que, à semelhança do cenário de teste anterior, o agente *Double-DQN* escolheu a política ótima de encaminhamento, permitindo com que o produto fosse produzido no menor tempo possível, deste cenário, ou seja, **33 segundos**.

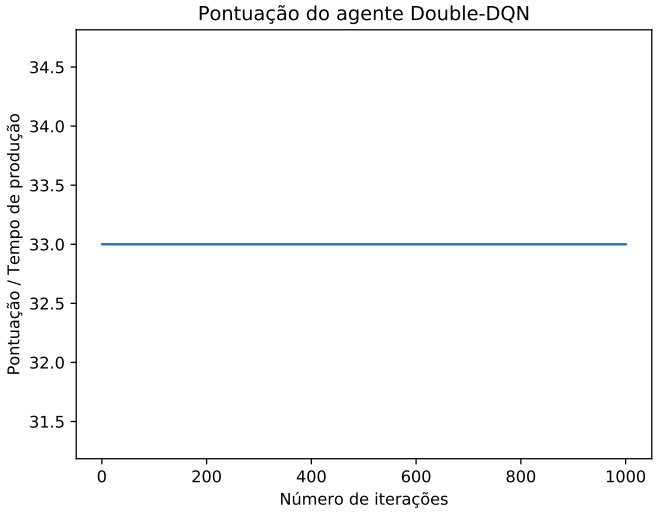


Figura 5.10: Pontuação realizada pelo agente *Double-DQN*, durante o cenário de teste 2 na simulação 2.

Capítulo 6

Conclusões e Trabalho Futuro

6.1 Conclusões

No início deste documento são propostas duas perguntas de investigação, bem como duas hipóteses que visam responder a estas perguntas. As perguntas propostas são as seguintes:

- Como conseguirá o produto escolher o caminho mais rápido, de modo a que seja feita uma produção eficiente?
- Que métodos podem ser utilizados para garantir inteligência individual a elementos de uma linha de produção?
- É possível garantir inteligência global através de inteligência individual?

Através dos resultados obtidos nos cenários de teste da **simulação 1**, é possível concluir que o algoritmo *Double-DQN*, juntamente com um simples SMA, é capaz de escolher o caminho de produção mais rápido, consoante a disposição dos restantes produtos na linha de produção simulada, permitindo uma produção no menor tempo possível, relativamente a esta simulação. Também é possível concluir que este algoritmo obteve um desempenho superior, comparativamente ao algoritmo heurístico *A**, e semelhante comparativamente ao *Q-learning*, no que toca ao encaminhamento de produtos. Nesta simulação também é possível comprovar que, através de transferência de conhecimento, é possível atribuir inteligência coletiva aos agentes, pela junção da inteligência individual.

Por sua vez, através da **simulação 2**, *digital twin* de um cenário real, é possível demonstrar a genericidade do algoritmo proposto e a capacidade que este tem em aprender um novo ambiente só através de imagens, fazendo com que seja rápido de implementar e

capaz de aprender novos ambientes de produção, sem necessitar de ser novamente desenvolvido, ou de grandes alterações a nível de código. Estas caraterísticas são benéficas, visto que, é possível implementar o algoritmo *Double-DQN*, em novos ambientes de produção, num espaço de tempo relativamente curto, diminuindo o seu tempo de desenvolvimento.

Outra vantagem deste algoritmo, é que não necessita de informação prévia do sistema para a sua aprendizagem, sendo assim capaz, através de tentativa e erro, de aprender uma política ótima para o objetivo proposto.

Como foi demonstrado nesta dissertação, o algoritmo *Double-DQN* apresenta um custo computacional elevado, visto que, utiliza uma rede neuronal. Este algoritmo necessita de uma quantidade de dados muito grande, ou seja, num ambiente mais complexo, do que os apresentados nesta dissertação, este pode treinar durante mais de um milhão de iterações, como no caso de [22]. Porém, com a evolução tecnológica, especificamente na área de placas gráficas, e a redução do seu custo monetário, torna-se cada vez mais fácil construir um computador de gama alta que seja capaz de executar este algoritmo num tempo mais aceitável. Outra vantagem, é a utilização deste algoritmo juntamente com um *digital twin*, de modo a permitir que este seja treinado num modo *offline*, não sendo necessário parar o sistema de produção já implementado. Após o seu treino e respetivos testes, é possível implementá-lo na linha de produção em tempo real.

Os resultados demonstrados nesta dissertação, através do treino dos agentes e dos cenários de teste da **simulação 1**, demonstram que, no caso das simulações utilizadas nesta dissertação, o algoritmo *Double-DQN* e o algoritmo *Q-learning*, apresentam desempenhos similares, enquanto o algoritmo *A** apresenta um desempenho inferior, devido ao facto de não ter em conta a posição dos outros agentes, escolhendo assim o caminho mais curto e não o mais rápido. No entanto, o algoritmo *Double-DQN* apresenta a vantagem de conseguir aprender um ambiente só pela visualização de imagens, pelo que não é necessário qualquer conhecimento extra sobre este, sendo mais versátil nas suas escolhas, tal como demonstrado no **cenário de teste 2**, realizado na **simulação 1**.

Na realização desta dissertação, existiram dois pontos fulcrais que levaram a uma implementação mais difícil e complexa, sendo estes o desenvolvimento das simulações e a criação das funções recompensa, aliando-se ao facto de que estas duas componentes influenciam o treino do agente.

Relativamente ao desenvolvimento das simulações, estas tiveram de ser desenvolvidas de raiz, ou seja, foi necessário programar toda a lógica dos movimentos e interações dos agentes, pelo que estas são características fulcrais para a aprendizagem do respetivo agente. Para além disto, foi necessário perceber o funcionamento da *framework*, de modo a implementar os respetivos algoritmos inteligentes nas simulações.

Por sua vez, foi imprescindível desenvolver funções recompensas adequadas para ambas as simulações, porém só após algum tempo de treino é que foi possível observar se as respetivas funções eram as mais adequadas para cada simulação. A grande dificuldade foi o facto de certos problemas na simulação, ou nas funções recompensa, só serem detetados após algum tempo de treino, sendo assim necessário parar a execução do treino, aplicar

as necessárias alterações e executar um novo treino.

Apesar dos algoritmos de inteligência artificial, nomeadamente algoritmos de DRL, apresentarem resultados interessantes e promissores, estes ainda apresentam inúmeros problemas, como por exemplo, o facto de serem demasiado sensíveis aos parâmetros utilizados, nomeadamente o rácio entre exploração aleatória e exploração pelo agente, o facto de ser extremamente difícil desenvolver funções recompensa adequadas para cenários muito complexos, onde o agente necessita de realizar diferentes tarefas.

Porém, a área de IA é uma área promissora, sendo que este tipo de algoritmos são os que demonstram uma maior capacidade de desenvolver uma IA capaz de aprender a resolver diversos problemas distintos. Atualmente são algoritmos muito investigados, onde em curtos espaços de tempo se fazem novas descobertas que permitem resolver algumas das suas desvantagens.

Como conclusão final, o algoritmo *Double-DQN* utilizado nesta dissertação, apresenta resultados promissores, comparativamente aos dois outros algoritmos testados, bem como caraterísticas interessantes, tais como versatilidade, dinamismo e a capacidade de aprender novos ambientes com alterações mínimas ao seu algoritmo. Porém, como supracitado, os algoritmos de DRL, nomeadamente o algoritmo *Double-DQN*, ainda apresentam algumas desvantagens.

6.2 Trabalho Futuro

O tema desta dissertação recai sobre a utilização de um algoritmo de DRL, nomeadamente o algoritmo *Double-DQN*, como tal nesta secção são mencionadas otimizações futuras relativamente a este algoritmo, bem como alternativas para a otimização de encaminhamento da produção utilizando este método.

Relativamente ao trabalho desenvolvido, existem inúmeras otimizações, visto que, trata-se de um algoritmo muito investigado, onde estão constantemente a ser descobertas novas otimizações.

Primeiramente, existe a possibilidade de utilizar a **simulação 2** como *digital twin* de um cenário real e através da utilização do agente *Double-DQN*, treinado e testado nesta dissertação, e de um SMA mais complexo, aplicar esta tecnologia numa linha de produção física.

Relativamente ao algoritmo existem inúmeras otimizações, sendo que só são referidas as que apresentam mais relevância. Anteriormente, foi referido que este algoritmo é muito sensível a parâmetros, nomeadamente ao rácio entre a exploração aleatória e a exploração pelo agente. Como tal, [11] utiliza uma nova abordagem, que consiste em adicionar *barulho* nos pesos da rede neuronal, fazendo com que o agente tome decisões não decididas por ele, ou seja, o agente adquire a capacidade de gerir um rácio denominado de σ, cuja função é a mesma do rácio utilizado na política $\epsilon - greedy$, não sendo necessário a definição prévia dos valores de exploração, visto que é o próprio agente que gere o rácio de exploração aleatória.

O desenvolvimento de uma função recompensa pode tornar-se uma tarefa extremamente complexa, pelo que se não for adequada pode levar o agente a ter comportamentos indesejados. Porém, já existem alternativas onde o próprio agente é que decide a melhor política de recompensas, através de recompensas distribuídas, em vez da utilização de um único valor. Esta otimização é denominada de *distributional reinforcement learning* utilizada em [20].

Existe também a possibilidade da criação de uma *replay memory* mais otimizada, que permite, de tempo em tempo, escolher os estados que ocorrem menos frequentemente, melhorando o treino da rede.

Como referido anteriormente o valor Q, consiste num valor que representa a vantagem do agente estar num estado específico, bem como a vantagem de escolher uma certa ação nesse estado. Como tal, é possível decompor este valor numa soma de dois valores, sendo estes o valor de estar num estado específico, $\mathbf{V(s)}$, e o valor que representa a vantagem de escolher uma certa ação nesse estado, $\mathbf{A(s,a)}$. Sendo assim, é possível criar duas camadas distintas na CNN utilizada, onde uma tem o objetivo de estimar o valor de $\mathbf{V(s)}$ e a outra de estimar o valor de $\mathbf{A(s,a)}$. Posteriormente, os dois valores provenientes das duas camadas são aglomerados num só.

A vantagem de utilizar esta otimização, é que deste modo não é necessário calcular todas as ações possíveis num dado estado porque a CNN, de uma forma intuitiva, consegue perceber se é vantajoso encontrar-se num certo estado, sem ter de saber o efeito de todas as ações possíveis nesse estado. Esta otimização permite diminuir o tempo de treino, porque num estado onde todas as ações são más, é desnecessário estar a calculá-las, como tal, de uma forma intuitiva, o agente sabe que não é vantajoso encontrar-se nesse estado. O algoritmo que utiliza esta otimização, juntamente com a rede alvo utilizada nesta dissertação, é denominado de *Dueling Double Deep Q-Network*, [49].

O algoritmo *Double-DQN* utilizado nesta dissertação, apresenta resultados satisfatórios, porém ao aplicar as otimizações supracitadas é possível aumentar o seu desempenho, diminuir o tempo de treino e aplicá-lo em ambientes com maior complexidade. Como já referido anteriormente, os algoritmos de DRL estão a ser muito estudados atualmente, existindo inúmeras otimizações, para além das mencionadas, sendo que muitas delas são extremamente recentes.

Para além das otimizações supracitadas, existem outros algoritmos de DRL que apresentam resultados promissores na resolução de ambientes extremamente difíceis para algoritmos de *Machine Learning* (ML), como por exemplo o jogo chinês *Go*, ou o cubo de *Rubik*, tais como:

- o algoritmo *Approximate Policy Iteration* (API),
- o algoritmo *Advantage Actor Critic* (A3C),

- o algoritmo *Dual Policy Iteration*, implementado no treino da famosa IA *AlphaGo*[1], [41],
- o algoritmo *Autodidactic Iteration* (ADI)[2], [1].

Proponho uma abordagem diferente da apresentada nesta dissertação, para aplicar algoritmos de DRL na distribuição e gestão de produtos numa linha de produção. Esta abordagem utilizaria um algoritmo, baseado no algoritmo presente em [1], juntamente com a utilização de um sistema SMA mais complexo, como o apresentado em [38], por exemplo.

Nesta abordagem o sistema SMA ficaria encarregue do controlo de todos os agentes presentes no *digital twin* da linha de produção, enquanto o algoritmo de DRL teria como função otimizar uma função heurística, que por sua vez seria utilizada num algoritmo heurístico, como por exemplo o algoritmo A^*. Deste modo, seria possível utilizar um sistema SMA complexo para atribuir inteligência coletiva e algoritmo heurístico como utilizado em [38], juntamente com uma função heurística otimizada por um algoritmo de DRL, cujo objetivo seria tornar o sistema mais versátil a condições adversas, dinâmico e proativo.

No âmbito desta dissertação, está a ser preparado um artigo científico, cujo objetivo é a análise dos resultados obtidos durante o presente trabalho.

[1] *AlphaGo* é uma IA criada pela empresa *Google DeepMind*, cujo objetivo é jogar o famoso jogo de tabuleiro chinês *Go*, um jogo extremamente difícil para algoritmos de ML. Esta IA apresentou resultados promissores, quando venceu um dos melhores jogadores profissionais de *Go*, *Lee Sedol*, numa competição à melhor de cinco jogos.

[2] O algoritmo ADI consiste numa alternativa do algoritmo API. [1] é o primeiro trabalho a utilizar DRL, nomeadamente o algoritmo ADI, para resolver um cubo de *Rubik*, sem qualquer intervenção humana, sendo capaz de resolver cubos de *Rubik* totalmente alterados de forma aleatória, numa média de 30 movimentos.

Bibliografia

[1] F. Agostinelli, S. McAleer, A. Shmakov e P. Baldi, "Solving the rubik's cube with deep reinforcement learning and search", *Nature Machine Intelligence*, p. 1, 2019.

[2] A. Ahmadi, A. H. Sodhro, C. Cherifi, V. Cheutet e Y. Ouzrout, "Evolution of 3c cyber-physical systems architecture for industry 4.0", em *International Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing*, Springer, 2018, pp. 448–459.

[3] F. Ansaria, S. Erola e W. Sihna, "Rethinking human-machine learning in industry 4.0: How does the paradigm shift treat the role of human learning?", *Education & Training*, vol. 2351, p. 9789, 2018.

[4] M. E. Aydin e E. Öztemel, "Dynamic job-shop scheduling using reinforcement learning agents", *Robotics and Autonomous Systems*, vol. 33, nº 2-3, pp. 169–178, 2000.

[5] B. Bagheri, S. Yang, H.-A. Kao e J. Lee, "Cyber-physical systems architecture for self-aware machines in industry 4.0 environment", *IFAC-PapersOnLine*, vol. 48, nº 3, pp. 1622–1627, 2015.

[6] W Bouazza, Y. Sallez e B Beldjilali, "A distributed approach solving partially flexible job-shop scheduling problem with a q-learning effect", *IFAC-PapersOnLine*, vol. 50, nº 1, pp. 15 890–15 895, 2017.

[7] M. Brettel, N. Friederichsen, M. Keller e M. Rosenberg, "How virtualization, decentralization and network building change the manufacturing landscape: An industry 4.0 perspective", *International Journal of Mechanical, Industrial Science and Engineering*, vol. 8, nº 1, pp. 37–44, 2014.

[8] J. Carlier e É. Pinson, "An algorithm for solving the job-shop problem", *Management science*, vol. 35, nº 2, pp. 164–176, 1989.

[9] G. Ehn e H. Werner, "Reinforcement learning for planning of a simulated production line", tese de mestrado, 2018.

[10] P. Fattahi, M. S. Mehrabad e F. Jolai, "Mathematical modeling and heuristic approaches to flexible job shop scheduling problems", *Journal of intelligent manufacturing*, vol. 18, nº 3, pp. 331–342, 2007.

[11] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin et al., "Noisy networks for exploration", *ArXiv preprint arXiv:1706.10295*, 2017.

[12] A. Garro, M. Mühlhäuser, A. Tundis, S. Mariani, A. Omicini e G. Vizzari, "Intelligent Agents and Environment", en, em *Encyclopedia of Bioinformatics and Computational Biology*, Elsevier, 2019, pp. 309–314, ISBN: 978-0-12-811432-2. DOI: 10.1016/B978-0-12-809633-8.20327-0. endereço: https://linkinghub.elsevier.com/retrieve/pii/B9780128096338203270 (acedido em 06/01/2019).

[13] H. V. Hasselt, "Double q-learning", em *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel e A. Culotta, eds., Curran Associates, Inc., 2010, pp. 2613–2621. endereço: http://papers.nips.cc/paper/3964-double-q-learning.pdf.

[14] J.-R. Jiang, "An improved cyber-physical systems architecture for industry 4.0 smart factories", *Advances in Mechanical Engineering*, vol. 10, nº 6, p. 1 687 814 018 784 192, 2018.

[15] Y. M. Jiménez, "A generic multi-agent reinforcement learning approach for scheduling problems", tese de doutoramento, Bruxels University, Faculty of Science e Bio-Engineering Sciences, 2012.

[16] K. Khantanapoka e K. Chinnasarn, "Pathfinding of 2d & 3d game real-time strategy with depth direction a* algorithm for multi-layer", em *2009 Eighth international symposium on natural language processing*, IEEE, 2009, pp. 184–188.

[17] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld e M. Hoffmann, "Industry 4.0", *Business & Information Systems Engineering*, vol. 6, nº 4, pp. 239–242, 2014.

[18] J. Lee, B. Bagheri e H.-A. Kao, "A cyber-physical systems architecture for industry 4.0-based manufacturing systems", *Manufacturing Letters*, vol. 3, pp. 18–23, 2015.

[19] Y. Lu, "Industry 4.0: A survey on technologies, applications and open research issues", *Journal of Industrial Information Integration*, vol. 6, pp. 1–10, 2017.

[20] K. Min, H. Kim e K. Huh, "Deep distributional reinforcement learning based high level driving policy determination", *IEEE Transactions on Intelligent Vehicles*, 2019.

[21] K. Miyazaki, "Proposal of a deep q-network with profit sharing", *Procedia Computer Science*, vol. 123, pp. 302–307, 2018.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., "Human-level control through deep reinforcement learning", *Nature*, vol. 518, nº 7540, p. 529, 2015.

[23] P. K. Muhuri, A. K. Shukla e A. Abraham, "Industry 4.0: A bibliometric analysis and detailed overview", *Engineering Applications of Artificial Intelligence*, vol. 78, pp. 218–235, 2019.

[24] B Naderi e A Azab, "Modeling and heuristics for scheduling of distributed job shops", *Expert Systems with Applications*, vol. 41, nº 17, pp. 7754–7763, 2014.

[25] G. Neagu, "A multi-agent model for job-shop scheduling", *IFAC Proceedings Volumes*, vol. 28, nº 24, pp. 221–226, 1995.

[26] M. A. Nielsen, *Neural networks and deep learning*. Determination press USA, 2015, vol. 25.

[27] J. A.B. d. Oliveira, "Coalition based approach for shop floor agility–a multiagent approach", 2003.

[28] E. Öztemel e M. Aydin, "Q-ii: An improved learning algorithm for intelligent agents", em *Proceedings of the 11th European Simulation Multi-Conference, Istanbul*, 1997, pp. 275–279.

[29] H. V. D. Parunak, "Characterizing the manufacturing scheduling problem", *Journal of manufacturing systems*, vol. 10, nº 3, pp. 241–259, 1991.

[30] C. D. Paternina-Arboleda e T. K. Das, "A multi-agent reinforcement learning approach to obtaining dynamic control policies for stochastic lot scheduling problem", *Simulation Modelling Practice and Theory*, vol. 13, nº 5, pp. 389–406, 2005.

[31] R. S. Peres, M. Parreira-Rocha, A. D. Rocha, J. Barbosa, P. Leitao e J. Barata, "Selection of a data exchange format for industry 4.0 manufacturing systems", 2015.

[32] R. S. Peres, A. D. Rocha, J. Barata, J. Barbosa e P. Leitão, "Improvement of multistage quality control through the integration of decision modeling and cyber-physical production systems", em *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE*, IEEE, 2016, pp. 5723–5728.

[33] B.-C. Pirvu, C.-B. Zamfirescu e D. Gorecky, "Engineering insights from an anthropocentric cyber-physical system: A case study for an assembly station", *Mechatronics*, vol. 34, pp. 147–159, 2016.

[34] F. G. Quintanilla, O. Cardin, A. L'Anton e P. Castagna, "Implementation framework for cloud-based holonic control of cyber-physical production systems", em *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*, IEEE, 2016, pp. 316–321.

[35] S. Riedmiller e M. Riedmiller, "A neural reinforcement learning approach to learn local dispatching policies in production scheduling", em *IJCAI*, vol. 2, 1999, pp. 764–771.

[36] T. Ross, *Fuzzy logic with engineering application. new york: Mcgraw hills*, 1995.

[37] V. Rudtsch, J. Gausemeier, J. Gesing, T. Mittag e S. Peter, "Pattern-based business model development for cyber-physical production systems", *Procedia CIRP*, vol. 25, pp. 313–319, 2014.

[38] A. D. B. da Silva Rocha, *An agent based architecture for material handling systems*, 2013.

[39] D. C. Smith, A. Cypher e J. Spohrer, "Kidsim: Programming agents without a programming language", *Communications of the ACM*, vol. 37, nº 7, pp. 54–67, 1994.

[40] B. Stout, "Smart moves: Intelligent pathfinding", *Game developer magazine*, vol. 10, pp. 28–35, 1996.

[41] W. Sun, G. J. Gordon, B. Boots e J Bagnell, "Dual policy iteration", em *Advances in Neural Information Processing Systems*, 2018, pp. 7059–7069.

[42] R. S. Sutton, "Introduction: The challenge of reinforcement learning", em *Reinforcement Learning*, Springer, 1992, pp. 1–3.

[43] G. Tashakor e R. Suppi, "Agent-based model for tumor-analysis using python+mesa", out. de 2018, pp. 248–253.

[44] P. M. Team. (2016). Mesa: Agent-based modeling in python 3+, endereço: `https://mesa.readthedocs.io/en/master/`.

[45] M. Uslar e D. Engel, "Towards generic domain reference designation: How to learn from smart grid interoperability", *DA-Ch Energieinformatik*, vol. 1, pp. 1–6, 2015.

[46] S. Vaidyaa, P. Ambadb e S. Bhoslec, "Industry 4.0–a glimpse", *Design Engineering*, vol. 2351, p. 9789, 2018.

[47] H. Van Hasselt, A. Guez e D. Silver, "Deep reinforcement learning with double q-learning.", em *AAAI*, Phoenix, AZ, vol. 2, 2016, p. 5.

[48] Y.-C. Wang e J. M. Usher, "A reinforcement learning approach for developing routing policies in multi-agent production scheduling", *The International Journal of Advanced Manufacturing Technology*, vol. 33, nº 3-4, pp. 323–333, 2007.

[49] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot e N. De Freitas, "Dueling network architectures for deep reinforcement learning", *ArXiv preprint arXiv:1511.06581*, 2015.

[50] B. Waschneck, A. Reichstaller, L. Belzner, T. Altenmüller, T. Bauernhansl, A. Knapp e A. Kyek, "Deep reinforcement learning for semiconductor production scheduling", em *SEMI Advanced Semiconductor Manufacturing Conference (ASMC), 2018 29th Annual*, IEEE, 2018, pp. 301–306.

[51] C. J. Watkins e P. Dayan, "Q-learning", *Machine learning*, vol. 8, nº 3-4, pp. 279–292, 1992.

[52] M. Wooldridge e N. R. Jennings, "Intelligent agents: Theory and practice", *The knowledge engineering review*, vol. 10, nº 2, pp. 115–152, 1995.

[53] X. Yu e B. Ram, "Bio-inspired scheduling for dynamic job shops with flexible routing and sequence-dependent setups", *International Journal of Production Research*, vol. 44, nº 22, pp. 4793–4813, 2006.

[54] F. Zezulka, P. Marcon, I Vesely e O. Sajdl, "Industry 4.0–an introduction in the phenomenon", *IFAC-PapersOnLine*, vol. 49, nº 25, pp. 8–12, 2016.

[55] H. Zhou, Y. Feng e L. Han, "The hybrid heuristic genetic algorithm for job shop scheduling", *Computers & Industrial Engineering*, vol. 40, nº 3, pp. 191–200, 2001.

[56] K. Zhou, T. Liu e L. Zhou, "Industry 4.0: Towards future industrial opportunities and challenges", em *Fuzzy Systems and Knowledge Discovery (FSKD), 2015 12th International Conference on*, IEEE, 2015, pp. 2147–2152.

ANEXO I

ANEXOS

I.1 Listagens *Mesa*

Listagem I.1: Exemplo da classe *Model* e classe *Agent*

```
from mesa import Agent, Model
from mesa.time import BaseScheduler
from mesa.space import MultiGrid

class Exemplo_Agente(Agent):
    def __init__(self, id, model):
        super().__init__(id, model)
        self.id = id

    def step(self):
        print("Agent Step.")

class Exemplo_Modelo(Model):
    def __init__(self, n_agents):
        super().__init__()
        # inicializacao da classe Time
        self.schedule = BaseScheduler(self)
        # inicializacao da classe Space
        self.grelha = MultiGrid(20, 20, torus=False)
        for i in range(n_agents):
            # inicializacao da classe Agent
            agente = Exempo_Agente(i, self)
            # agendamento do agente na classe Time
            self.schedule.add(agente)
            # Posicionar o agente no ambiente
            self.grelha.place_agent(agente, (i, i))

```

```
28      def step(self):
29          self.schedule.step()
```

Listagem I.2: Exemplo da classe *ModularVisualization* e classe *Modules*

```
1  from mesa.visualization.modules import CanvasGrid
2  from mesa.visualization.ModularVisualization import ModularServer
3
4  def carateristicas_agentes(agent):
5      # definicao das componentes visuais do agente Exemplo_Agente
6      carateristicas = {"Filled": "true"}
7
8      if type(agent) is Exemplo_Agente:
9          carateristicas["Shape"] = "circle"
10         carateristicas["r"] = 0.5
11         carateristicas["Color"] = "green"
12         carateristicas["Layer"] = 1
13
14     return carateristicas
15
16 # criacao da grelha do ambiente
17 grelha = CanvasGrid(carateristicas_agentes, 20, 20, 200, 200)
18 # Criacao do servidor
19 servidor = ModularServer(Exemplo_Modelo,
20     [grelha], "Exemplo_Modelo", {'n_agentes': 10})
21 # execucao do servidor
22 servidor.launch()
```

I.2 Listagens Simulação 1

Listagem I.3: Implementação das caraterísticas visuais de alguns agentes da simulação 1

```
1  from mesa_model.model import SimulationModel
2  from mesa.visualization.modules import CanvasGrid
3  from mesa.visualization.ModularVisualization import ModularServer
4  from mesa_model.agents import ProductAgent1, ConveyorAgent, ResourceAgentA, \
5  BeginAgent
6
7  def agent_portrayal(agent):
8      portrayal = {"Filled": "true"}
9
10     if type(agent) is ProductAgent1:
11         portrayal["Shape"] = "circle"
12         portrayal["r"] = 0.5
13         portrayal["Color"] = "green"
14         portrayal["Layer"] = 2
15     elif type(agent) is ConveyorAgent:
16         portrayal["Shape"] = "rect"
17         portrayal["w"] = 1
```

```
18          portrayal["h"] = 1
19          # #A9A9A9 equivale à cor cinzento escuro
20          portrayal["Color"] = "#A9A9A9"
21          portrayal["Layer"] = 0
22      elif type(agent) is ResourceAgentA:
23          portrayal["Shape"] = "rect"
24          portrayal["w"] = 1
25          portrayal["h"] = 1
26          # #5A4FCF equivale à cor Iris
27          portrayal["Color"] = "#5A4FCF"
28          portrayal["Layer"] = 1
29      (...)
30      elif type(agent) is BeginAgent:
31          portrayal["Shape"] = "rect"
32          portrayal["w"] = 1
33          portrayal["h"] = 1
34          # #008000 equivale à cor verde escuro
35          portrayal["Color"] = "#008000"
36          portrayal["Layer"] = 0
37
38      return portrayal
```

Listagem I.4: Instanciação de alguns agentes da simulação 1

```
1  from mesa_model.agents import ResourceAgentA, ProductAgent1, BeginAgent
2
3  def begin_agent(model):
4      begin_pos = (0, 12)
5      unique_id = "Begin_"+str(0)
6      begin = BeginAgent(unique_id, model)
7      model.grid.place_agent(begin, begin_pos)
8
9  # o metodo create_pa encontra-se incompleto neste excerto de codigo
10 def create_pa(model):
11     pos = (0, 12)
12
13     for i in range(3):
14         unique_id = "PA1_"+str(i)
15         pa = ProductAgent1(unique_id, model)
16         # metodo da classe model que permite adicionar
17         # os agentes ao agendamento
18         model.schedule.add(pa)
19         # metodo da classe model que permite adicionar
20         # os agentes no ambiente
21         model.grid.place_agent(pa, pos)
22
23 # adicionar o Agente Tipo 1 novamente na simulacao
24 def add_agent_1(pa, model):
25     pos = (0, 12)
26     unique_id = pa.unique_id
```

```
27      pa = ProductAgent1(unique_id, model)
28      model.schedule.add(pa)
29      model.grid.place_agent(pa, pos)
30
31  def ra_agent_a(model):
32      ra_pos_list_a = [(2, 8), (4, 6), (6, 8), (8, 8), (10, 6), (12, 8)]
33
34      for i in ra_pos_list_a:
35          index = ra_pos_list_a.index(i)
36          unique_id = "ResourceA_" + str(index)
37          ra = ResourceAgentA(unique_id, model)
38          model.grid.place_agent(ra, i)
39      return ra_pos_list_a
```

Listagem I.5: Excerto da classe *Model* da simulação 1

```
1   from mesa import Model
2   from mesa.space import MultiGrid
3   from mesa.time import RandomActivation
4
5   import sim_models.model_1.sim as sim
6
7   class SimulationModel(Model):
8       # relembrar que as variaveis cnn_model e model_target
9       # so sao utilizadas para a implementacao do algoritmo Double-DQN
10      def __init__(self, width, height, cnn_model, model_target):
11          super().__init__(width, height, cnn_model, model_target)
12          # Variável que define o início ou paragem da simulacao
13          self.running = True
14
15          self.ra_pos_list_A = None
16          self.ra_pos_list_B = None
17          self.ra_pos_list_C = None
18
19          self.cnn_model = cnn_model
20          self.target_model = model_target
21
22          self.schedule = RandomActivation(self)
23          self.grid = MultiGrid(width, height, True)
24
25          self.ra_pos_list_A, self.ra_pos_list_B,
26          self.ra_pos_list_C = sim.build_model(self)
```

Listagem I.6: Criação de alguns agentes presentes na simulação 1

```
1   class ConveyorAgent(Agent):
2       def __init__(self, unique_id, model):
3           super().__init__(unique_id, model)
4
5   class ResourceAgentA(Agent):
6       def __init__(self, unique_id, model):
```

```
7           super().__init__(unique_id, model)
8           self.ra_a_time = 10
9
10  class ProductAgent1(Agent):
11      def __init__(self, unique_id, model):
12          super().__init__(unique_id, model)
13          self.unique_id = unique_id
14          self.ResourcePoints = 0
15          self.MovePoints = 0
16          # parametro que permite definir o movimento do agente
17          self.move = 1
18
19      def step(self):
20          # quando o agente se encontra num recurso
21          # fica parado durante o tempo necessario
22          if not stop_move(self):
23              get_state(self, self.model)
```

Listagem I.7: Método *stop_move* da simulação 1

```
1
2   # permite parar os Agentes Produto nos Agentes Recurso
3
4   def stop_move(pa):
5       # caso o agente se encontre na posicao de um recurso
6       if pa.pos in pa.model.ra_pos_list_A and pa.ra_a_time > 0:
7           pa.ra_a_time -= 1
8           pa.ResourcePoints += 1
9           return True
10      elif pa.pos in pa.model.ra_pos_list_B and pa.ra_b_time > 0:
11          pa.ra_b_time -= 1
12          pa.ResourcePoints += 1
13          return True
14      elif pa.pos in pa.model.ra_pos_list_C and pa.ra_c_time > 0:
15          pa.ra_c_time -= 1
16          pa.ResourcePoints += 1
17          return True
18      return False
```

Listagem I.8: Métodos relacionados com a movimentação do agente produto na simulação 1

```
1   import random
2
3   # permite movimentar os Agentes Produto para cima
4
5   def move_up(pa, model):
6       x, y = pa.pos
7       y += 1
8       pa.MovePoints += 1
```

```
9       model.grid.move_agent(pa, (x, y))
10
11  # movimento aleatorio no ponto de decisao
12
13  def random_move(pa):
14      pos = []
15      x, y = pa.pos
16      x_aux = x + 1
17      right_pos = (x_aux, y)
18      y_aux = y - 1
19      down_pos = (x, y_aux)
20      pos.append(right_pos)
21      pos.append(down_pos)
22      choice = random.choice(pos)
23      pa.MovePoints += 1
24      if choice == right_pos:
25          pa.move = 1
26      elif choice == down_pos:
27          pa.move = 2
```

Listagem I.9: Método *check_right_avail* da simulação 1

```
1   import random
2
3   # verifica se existe algum Agente Produto
4   # na posicao a direita do Agente Produto em questao
5
6   def check_right_avail(pa):
7       if pa.move == 1:
8           x, y = pa.pos
9           x += 1
10          new_pos = (x, y)
11          next_cell = pa.model.grid.get_neighbors \
12          (new_pos, moore=False, include_center=True, radius=0)
13          if len(next_cell) != 1:
14              return True
15          else:
16              return False
```

Listagem I.10: Método *check_out* da simulação 1

```
1
2   # remove um Agente Produto da lista de agendamento
3   # e do ambiente de simulacao
4
5   def check_out(pa, model):
6       # (6, 0) -> posicao do Agente Estacao Final
7       if pa.pos == (6, 0):
8           model.schedule.remove(pa)
9           model.grid.remove_agent(pa)
10          return True
```

```
11      return False
```

Listagem I.11: Método *move_pa1* da simulação 1 correspondente ao movimento do Agente Produto Tipo 1

```
1  def move_pa1(pa, model, unique_id_list):
2      if pa.unique_id in unique_id_list:
3          choose_movement(pa)
4
5          if pa.pos == (2, 12):
6              random_move(pa)
7          elif pa.pos == (8, 12):
8              pa.move = 2
9
10         if check_right_avail(pa):
11             pa.move = 0
12         if check_down_avail(pa):
13             pa.move = 0
14         if check_left_avail(pa):
15             pa.move = 0
16
17         aux_pa = pa
18         if check_out(pa, model):
19             sim.add_agent_1(aux_pa, model)
20         else:
21             movement(pa, model)
```

Listagem I.12: Método *get_state* da simulação 1

```
1  def get_state(pa, model):
2      unique_id_1, unique_id_2, unique_id_3, unique_id_4 = \
3      get_unique_id_lists(model)
4
5      move_pa1(pa, model, unique_id_1)
6      move_pa2(pa, model, unique_id_2)
7      move_pa3(pa, model, unique_id_3)
8      move_pa4(pa, model, unique_id_4)
```

I.3 Listagens Simulação 2

Listagem I.13: Implementação visual do *EndAgent* na simulação 2

```
1  (...)
2  if type(agent) is EndAgent:
3          portrayal["Shape"] = "rect"
4          portrayal["w"] = 1
5          portrayal["h"] = 1
6          if CHANGE_COLOR_BEGIN:
7              portrayal["Color"] = "white"
```

```
        else:
            portrayal["Color"] = "black"
        portrayal["Layer"] = 0
(...)
```

Listagem I.14: Método *move_pa2* correspondente ao Agente Produto Tipo 2 presente na simulação 2

```
def move_pa2(pa, model):
    global grua_on_2, pa_move_2

    if not REMOVE_2:
        pa_pos = pa_move_2[0]

        # verificar vizinhanca do agente
        this_cell = pa.model.grid.get_neighbors(pa_pos,
        moore=False, include_center=True, radius=0)

        x, y = pa_pos

        # se o Agente Produto Tipo 2 encontrar-se num
        # Agente Grua
        if pa_pos in ca_pos:
            grua_on_2 = True
        else:
            grua_on_2 = False

        # caso um Agente Grua ou a posicao
        # seguinte estejam ocupados o agente
        # nao se move
        if pa_pos in ca_pos and grua_on_1:
            grua_on_2 = False
        elif len(this_cell) == 2:
            print("agent in the next cell.")
        else:
            # (6, 4) -> posicao do Recurso Tipo A
            if pa_pos == (6, 4):
                RESOURCE_OCCUPIED = True
                CHANGE_COLOR_A = False
            else:
                RESOURCE_OCCUPIED = False
                CHANGE_COLOR_A = True

            # o agente Double-DQN ja passou pelo Recurso Tipo A,
            # logo este fica a vermelho
            if RA_TIMER_CHECK:
                CHANGE_COLOR_A = False

            model.grid.move_agent(pa, (x, y))
            pa_move_2.pop(0)
```

```
43      else:
44          model.schedule.remove(pa)
45          sim.add_agent_2(pa, model)
46          # posicoes para o qual o Agente Tipo 2
47          # se irá deslocar
48          pa_move_2 = [(6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (5, 7), (5, 8)]
```

I.4 Listagens *Double-DQN*

Listagem I.15: Método *save_model*

```
1
2   # guarda os dados essenciais ao treino dos
3   # algoritmos
4
5   def save_model(cnn_model):
6       print("Save model.")
7       # colocar a simulacao em pausa
8       play_pause_model()
9
10      # armazenar os pesos da rede neuronal
11      cnn_model.save_weights(folder_var + ".h5", overwrite=True)
12      # armazenar a replay memory
13      save_obj(D, "D")
14      # armazenar o numero de iteracoes
15      save_obj(T, "time")
16      # armazenar o valor de atualizacao do target_model
17      save_obj(TAU, "tau")
18      # armazenar o valor de epsilon
19      save_obj(EPSILON, "epsilon")
20
21      # armazenar os pontos do Agente Produto a ser treinado
22      scores_victory_df.to_csv(scores_victory_file_path, index=False)
23      # armazenar os valores Q
24      q_values_df.to_csv(q_value_file_path, index=False)
25      # armazenar o valor acumulado de recompensas do agente
26      reward_df.to_csv(reward_file_path, index=False)
27
28      # retormar a simulacao
29      play_pause_model()
```

Listagem I.16: Criação de *DataFrames* necessários para armazenar o primeiro grupo de dados

```
1   import os
2   import panda as pd
3
4   scores_victory_df = pd.read_csv(scores_victory_file_path) \
5   if os.path.isfile(scores_victory_file_path) else \
```

```
pd.DataFrame(columns=['scores_victory'])

q_values_df = pd.read_csv(q_value_file_path) if \
os.path.isfile(q_value_file_path) else pd.DataFrame(columns=['q_values'])

reward_df = pd.read_csv(reward_file_path) if \
os.path.isfile(reward_file_path) else pd.DataFrame(columns=['reward'])
```

Listagem I.17: Métodos utilizados para guardar objetos e carregar ficheiros

```
import pickle

# permite guardar o conteudo dos objetos num ficheiro .pkl
def save_obj(obj, name):
    with open(dir_folder + name + '.pkl', 'wb') as f:
    pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)

# permite carregar o conteudo de um ficheiro .pkl para os respetivos objetos
def load_obj(name):
    with open(dir_folder + name + '.pkl', 'rb') as f:
    return pickle.load(f)
```

Listagem I.18: Método *init_cache*

```
from collections import deque

# carrega os valores da simulacao
# para as respetivas variaveis

def init_cache():
    t_finish = 0
    save_obj(t_finish, "t_finish")
    tau = 0
    save_obj(tau, "tau")
    t = 0
    save_obj(INITIAL_EPSILON, "epsilon")
    save_obj(t, "time")
    D = deque()
    save_obj(D, "D")
```

Listagem I.19: Método *buildmodel*

```
from keras.optimizers import RMSprop
from keras.models import Sequential, model_from_json
from keras.layers.core import Dense, Activation, Flatten
from keras.layers.convolutional import Conv2D, MaxPooling2D

# numero de acoes do agente
ACTIONS = 4
# racio de aprendizagem
LEARNING_RATE = 0.00025
```

```
# tamanho do input da rede
img_cols, img_rows, img_channels = 80, 80, 4
# otimizador da rede
rms_prop = RMSprop(lr=LEARNING_RATE, rho=0.95, epsilon=0.01)

# cria a CNN utilizada nesta dissertacao

def buildmodel():
    cnn_model = Sequential()

    cnn_model.add(Conv2D(32, (8, 8), padding="same", strides=(4, 4),
                         input_shape=(img_cols, img_rows, img_channels)))
    cnn_model.add((MaxPooling2D(pool_size=(2, 2))))
    cnn_model.add(Activation('relu'))
    cnn_model.add(Conv2D(64, (4, 4), strides=(2, 2), padding='same'))
    cnn_model.add((MaxPooling2D(pool_size=(2, 2))))
    cnn_model.add(Activation('relu'))
    cnn_model.add(Conv2D(64, (4, 4), strides=(2, 2), padding='same'))
    cnn_model.add((MaxPooling2D(pool_size=(2, 2))))
    cnn_model.add(Activation('relu'))

    cnn_model.add(Flatten())
    cnn_model.add(Dense(512))
    cnn_model.add(Activation('relu'))
    cnn_model.add(Dense(ACTIONS))

    # compilar o modelo da rede neuronal
    cnn_model.compile(loss="logcosh", optimizer=rms_prop)

    # armazenar o modelo da rede neuronal
    model_json = cnn_model.to_json()
    with open(folder_var + ".json", "w") as json_file:
        json_file.write(model_json)

    # armazenar os pesos da rede neuronal
    cnn_model.save_weights(folder_var + '.h5')
    # imprimir o sumario da rede neuronal
    cnn_model.summary()
```

Listagem I.20: Método *open_model*

```
from keras.optimizers import RMSprop
from keras.models import model_from_json

# carrega o modelo da CNN para uma
# variavel

def open_model():
    # leitura do ficheiro .json
    json_file = open(folder_var + ".json", 'r')
```

```
    # criacao de um modelo json
    load_model_json = json_file.read()
    json_file.close()
    # criacao da rede neuronal a partir do modelo json
    cnn_model = model_from_json(load_model_json)
    # carregamento dos pesos da rede neuronal
    cnn_model.load_weights(folder_var + ".h5")
    # compilacao da rede neuronal
    cnn_model.compile(loss="logcosh", optimizer=rms_prop)
    # imprimir o sumario da rede neuronal
    cnn_model.summary()
    return cnn_model
```

Listagem I.21: Método *clone_cnn*

```
from keras.optimizers import RMSprop
from keras.models import clone_model

# clona a CNN principal

def clone_cnn(cnn_model):
    # clonagem da rede principal para a rede alvo
    model_target = clone_model(cnn_model)
    # atribuicao dos pesos da rede principal para a rede alvo
    model_target.set_weights(cnn.get_weights())
    # compilacao da rede alvo
    model_target.compile(loss="logcosh", optimizer=rms_prop)
    return model_target
```

Listagem I.22: Método *html_image*

```
from selenium import webdriver
from PIL import Image
from io import BytesIO

import base64

# adquire a imagem HTML da simulacao

def html_image():
    # armazenar a imagem base64 na variavel canvas
    canvas = driver.execute_script(getbase64Script)
    # converter para um numpy array
    screen = np.array(Image.open(BytesIO(base64.b64decode(canvas))))
    return screen
```

Listagem I.23: Método *process_image*

```
import cv2

# processa a imagem do ambiente
```

```
4
5  def process_image(image):
6      image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
7      image = cv2.resize(image, (80, 80))
8      return image
```

Listagem I.24: Métodos *image_treatment/grab_image/image2video*

```
1  import os
2  import cv2
3  import uuid
4  import glob
5
6  # permite guardar imagens do ambiente
7  # numa diretoria
8
9  def grab_image():
10     screen = html_image()
11     # biblioteca uuid para gerar nomes aleatórios para as imagens
12     cv2.imwrite(os.path.join(image_path, 'print_' + str(uuid.uuid4()) + \
13     '.jpg'), screen)
14
15 # permite criar um video atraves de imagens
16
17 def image2video():
18     # armazenar as imagens num objeto
19     files = glob.glob(image_path + '\\*.jpg')
20     # organizar as imagens por data
21     files.sort(key=os.path.getmtime)
22     # utilizar as imagens para criar um video
23     out = cv2.VideoWriter('cnn_train.avi', cv2.VideoWriter_fourcc(*'DIVX'), \
24     15, (500, 500))
25
26     for filename in tqdm(files):
27         img = cv2.imread(filename)
28         img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
29         out.write(img)
30     out.release()
31
32 def image_treatment():
33     screen = html_image()
34     image = process_image(screen)
35     return image
```

Listagem I.25: Método *init_train_network*

```
1  import numpy as np
2
3  # inicia o treino do algoritmo
4
5  def init_train_network():
```

```
    D = load_obj("D")

    # adquirir uma imagem do ambiente 80x80x1
    x_t = image_treatment()
    # empilhar a imagem numa pilha de quatro imagens
    s_t = np.stack((x_t, x_t, x_t, x_t), axis=2)
    # alterar a forma do vetor para 1x80x80x4
    s_t = s_t.reshape((1, *s_t.shape))
    # variavel referente à pilha de imagens iniciais
    initial_state = s_t
    # variavel referente ao numero de observacoes de treino
    OBSERVE = OBSERVATION
    # carregamento da variavel tau
    tau = load_obj("tau")
    # carregamento da variavel t_finish
    t_finish = load_obj("t_finish")
    # carregamento da variavel epsilon
    epsilon = load_obj("epsilon")
    # carregamento da variavel t
    t = load_obj("time")
    return last_time, D, s_t, initial_state, OBSERVE, \
        epsilon, tau, t, t_finish
```

Listagem I.26: Método *find_max_q* utilizado na simulação 1

```
import numpy as np
import random

# encontra a melhor acao para o
# estado atual

def find_max_q(cnn_model):
    # gerar um valor aleatorio entre zero e um
    random_exploration = np.random.rand()

    if EPSILON > random_exploration:
        ACTION_INDEX = random.choice(ACTIONS)
    else:
        q = cnn_model.predict(S_T)
        # averiguar o indice do maior valor de Q presente no vetor q
        choice = np.argmax(q)
        # escolher a melhor acao das acoes possiveis
        ACTION_INDEX = ACTIONS[int(choice)]
```

Listagem I.27: Método *train_memory_batch*

```
import numpy as np
import random

# treina a CNN utilizada

```

```
6  def train_memory_batch(memory, model_target, cnn_model):
7      # criacao da memoria auxiliar
8      minibatch = random.sample(memory, BATCH)
9      #criacao do numpy array correspondente aos estados
10     inputs = np.zeros((BATCH, S_T.shape[1], S_T.shape[2], S_T.shape[3]))
11     # criacao do numpy array correspondente aos valores Q
12     targets = np.zeros((inputs.shape[0], N_ACTIONS))
13
14     Q_sa = 0
15
16     for i in range(0, len(minibatch)):
17         # estado atual
18         state_t = minibatch[i][0]
19         # acao escolhida
20         action_t = minibatch[i][1]
21         # recompensa obtida
22         reward_t = minibatch[i][2]
23         # estado seguinte
24         state_t1 = minibatch[i][3]
25         # variavel terminal
26         terminal = minibatch[i][4]
27
28         # adicionar o estado atual ao numpy array de estados
29         inputs[i:i + 1] = state_t
30         # previsao dos valores Q para o estado atual
31         targets[i] = cnn_model.predict(state_t)
32         # previsao dos valores Q para o estado seguinte
33         Q_sa = model_target.predict(state_t1)
34
35         if terminal:
36             targets[i, action_t] = reward_t
37         else:
38             targets[i, action_t] = reward_t + GAMMA * np.max(Q_sa)
39     # treino da rede
40     loss = cnn_model.train_on_batch(inputs, targets)
41     return loss, Q_sa
```

Listagem I.28: Excerto do método *train_network*

```
1
2  # metodo principal do algoritmo Double-DQN
3
4  def train_network(cnn_model, model_target):
5      # decrementar o valor de epsilon
6      if EPSILON > FINAL_EPSILON and T > OBSERVE:
7          EPSILON -= (INITIAL_EPSILON - FINAL_EPSILON) / EXPLORE
8      # adquirir o novo estado
9      x_t1 = image_treatment()
10     # alterar o formato do novo estado
11     x_t1 = x_t1.reshape(1, x_t1.shape[0], x_t1.shape[1], 1)
```

```
12      # colocar o novo estado na pilha
13      s_t1 = np.append(x_t1, S_T[:, :, :, :3], axis=3)
14      # guardar as informacoes na replay memory
15      D.append((S_T, ACTION_INDEX, REWARD, s_t1, IS_OVER))
16      # se a replay memory esta cheia, retirar a entrada mais antiga
17      if len(D) > REPLAY_MEMORY:
18          D.popleft()
19      # se o agente saiu do modo de observacao
20      if T > OBSERVE:
21          # incrementar a variavel de atualizacao da rede alvo
22          TAU += 1
23          # treinar a rede
24          history, q_max = train_memory_batch(D, model_target, cnn_model)
25          # atualizar os pesos da rede alvo
26          if TAU > TARGET_MODEL_NUM:
27              model_target.set_weights(cnn_model.get_weights())
28              TAU = 0
29      # tornar a pilha atual na pilha com o estado seguinte
30      S_T = s_t1
31      # incrementar o numero de iteracoes
32      T = T + 1
33      (...)
```

I.5 Listagens *Q-learning*

Listagem I.29: Excerto do método *find_max_q* do algoritmo *Q-learning*

```
1  import numpy as np
2
3  # encontra a melhor acao
4  # para o estado atual
5
6  def find_max_q():
7      (...)
8
9      else:
10         ACTION_INDEX = np.argmax(q_table[STATE])
```

Listagem I.30: Excerto do método *train_network* do algoritmo *Q-learning*

```
1  import numpy as np
2
3  # funcao principal do algoritmo
4  # Q-learning
5
6  def train_network():
7      (...)
8
9      old_value = q_table[STATE, ACTION_INDEX]
```

```
10      next_max = np.max(q_table[next_state])
11      new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
12      q_table[STATE, ACTION_INDEX] = new_value
13
14      (...)
```