NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE ⟨ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES⟩

PEDRO MIGUEL SILVÉRIO DE OLIVEIRA

Licenciado em ⟨Ciências de Engenharia Eletrotécnica e de Computadores⟩

DYNAMIC NETWORK SLICING USING DEEP REINFORCEMENT LEARNING

MESTRADO EM ⟨ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES⟩

Universidade NOVA de Lisboa
⟨Novembro⟩, ⟨2021⟩

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTAMENTO DE
⟨ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES⟩

DYNAMIC NETWORK SLICING USING DEEP REINFORCEMENT LEARNING

PEDRO MIGUEL SILVÉRIO DE OLIVEIRA

Licenciado em ⟨Ciências de Engenharia Eletrotécnica e de Computadores⟩

Orientador: Pedro Miguel Figueiredo Amaral
Professor Auxiliar, Universidade NOVA de Lisboa

MESTRADO EM ⟨ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES⟩

Universidade NOVA de Lisboa
⟨Novembro⟩, ⟨2021⟩

Dynamic Network Slicing Using Deep Reinforcement Learning

Dedicado à minha família e à Margarida.

Agradecimentos

Começo por agradecer ao meu orientador, professor Pedro Amaral, por me ter apresentado este desafio e apoiado ao longo da sua resolução, desde os materiais de apoio às discussões de implementação, as críticas e sugestões, mantendo-se sempre compreensivo mesmo quando me tive de afastar parcialmente do projeto. Um grande obrigado.

De seguida, um agradecimento a toda a NOVA-FCT. A formação adquirida nesta instituição servirá para toda uma carreira. Um obrigado especial aos dois professores mais marcantes no meu percurso: o monitor de Programação de Microprocessadores Gonçalo Freitas e o monitor da área de Robótica Ricardo Peres. Obrigado Gonçalo, pela empatia pelas dificuldades dos mais novos e capacidade de motivar alguém tão inexperiente na área a querer aprender e fazer mais com programação. Este foi o ponto de viragem que me fez desenvolver uma paixão pela programação e me criou as oportunidades de carreira que agora aproveito. Obrigado Ricardo pela capacidade de manter aulas *online* tão interessantes e com tópicos tão atuais e com aplicações reais, tópicos estes que lido agora diariamente. Obrigado também ao professor João Lourenço por criar o *template* para este trabalho, tornando a experiência de escrever uma dissertação muito mais simples e agradável.

Um enorme obrigado àqueles que a faculdade me trouxe: Ana, Guilherme e Henrique, que estiveram comigo do primeiro ao último dia, celebrando juntos os sucessos e suportando-nos nos momentos em que a vontade de desistir superava todas as outras. Assim continuaremos.

Um gigantesco agradecimento à minha namorada Margarida: a pessoa que nunca duvidou das minhas capacidades por mais que eu duvidasse, a inspiração que tornou este enorme processo de Mestrado aceitável. Sem qualquer dúvida, a minha motivação para ter atingido este grau académico.

Finalmente, o maior obrigado possível àqueles sem os quais este percurso seria impossível: os meus pais. Obrigado pela educação que me deram, por me terem dado tudo o que sempre precisei, por todos os sacrifícios feitos para que pudesse atingir o que sempre desejaram para mim. Não há palavras para descrever a gratidão que sinto. Esta é para vocês.

"Sometimes a scream is better than a thesis." (Ralph Waldo Emerson)

Resumo

Atualmente o *network slicing* é um dos maiores potenciadores de novos elementos no negócio das redes 5G. Isto deve-se ao facto de este paradigma permitir a criação de *slices* independentes, com os seus recursos rádio, de rede e computacionais virtual e logicamente separados.

Utilizando *network slicing*, as operadoras poderão vender recursos de infraestrutura de qualquer tipo a *tenants*. Os *tenants* utilizam estes recursos para vender serviços aos seus clientes, os utilizadores finais. Neste contexto, um problema que é fundamental resolver é o de como melhorar o lucro da operadora, garantindo o cumprimento dos SLAs dos pedidos e distribuindo os recursos da rede de forma a aumentar a sua utilização.

Nesta dissertação propõe-se desenhar dois algoritmos baseados em DRL para a admissão de *slices* na rede de transporte, aprendendo que pedidos aceitar e rejeitar, procurando satisfazer sempre os requisitos dos pedidos dos *tenants*. Os contributos deste estudo passam pela formalização do problema da admissão de *slices* na rede, seguindo-se a sua simulação e implementação dos agentes utilizando conjuntamente o *Containernet*, o controlador *Ryu*, o *OpenAI Gym* e o *framework PyTorch*. O resultado são dois algoritmos baseados em DRL capazes de atingir boas *performances* neste cenário simulado.

Palavras-chave: Network Slicing, Deep Reinforcement Learning, Controlo de Admissão

Abstract

Nowadays network slicing is one of the biggest drivers of new elements in the 5G network business. This is because this paradigm allows the creation of independent slices, with their virtually and logically separated radio, network and computational resources.

Using network slicing, operators sell infrastructure resources of any kind to tenants, while tenants use these resources to sell services to their customers, the end users. In this context, a problem that is essential to solve is how to improve the operator's profit, ensuring compliance with the requests' SLAs and distributing network resources in order to increase its usage rate.

This dissertation proposes to design two algorithms based on DRL for slice admission in the transport network, learning which request to accept and reject while guaranteeing the requirements of the tenants requests. The contributions of this study start with the formalization of the problem of slice admission, followed by its simulation and implementation of DRL agents using Containernet, the Ryu controller, OpenAI Gym and the PyTorch framework. The result is two DRL-based algorithms capable of achieving good performances in this simulated scenario.

Keywords: Network slicing, Deep Reinforcement Learning, Admission Control

Índice

Índice de Figuras

Siglas

API	Application Programming Interface 1
BS	Base Station 16
CDN	Content Distribution Network 18
CN	Core Network 2
CNN	Convolutional Neural Network 9
CPU	Computation Processing Unit 16
DDQN	Deep Distributional Q-Network 14
DL	Deep Learning 2
DNAF	Discrete Normalized Advantage Function 14
DQL	Deep Q-Learning 11
DQN	Deep Q-Network 3
DRL	Deep Reinforcement Learning 2
E2E	End-to-End 1
eMBB	Enhanced Mobile Broadband 1
FIFO	First-In-First-Out 12
GAN	Generative Adversarial Network 10
GAN-DDQN	GAN-powered Deep Distributional Q Network 17
GPU	Graphics Processing Unit 9
IMS	IP Multimedia Subsystems 18
IoT	Internet of Things 1

k-NN k-Nearest Neighbour 13
KL Kullback-Leibler 8
KPI Key Performance Indicator 1

ML Machine Learning 2
mMTC Massive Machine-type Communications 1
MSE Mean-Squared Error 8

NAF Normalized Advantage Function 13
NFV Network Functions Virtualization 1

OVS Open vSwitch 30

QoE Quality of Experience 1
QoS Quality of Service 1
QR-DQN Quantile Regression DQN 14

RAN Radio Access Network 2
ReLU Rectified Linear Unit 35
RL Reinforcement Learning 2

SDN Software-Defined Networking 1
SE Spectral Efficiency 16
SGD Stochastic Gradient Descent 8
SLA Service Level Agreement 2
SMDP Semi-Markov Decision Process 21
SSR SLA Satisfaction Ratio 17

TD Temporal Difference 7

URLLC Ultra-Reliable and Low-Latency Communications 1

V2X Vehicle-to-Everything 18
VNF Virtual Network Function 1

1

Introdução

À medida que os dispositivos móveis se tornam cada vez mais essenciais no dia-a-dia, a infraestrutura e respetiva gestão que suportam a sua comunicação tornam-se críticas. O crescimento exponencial de serviços móveis e os avanços na *Internet of Things (IoT)* promoveram iniciativas globais para o desenvolvimento de sistemas de comunicação móvel e sem fios de (5G) [3].

As redes 5G de 2020 em diante terão como objetivo suportar alguns cenários genéricos, nomeadamente *Enhanced Mobile Broadband (eMBB)*, *Ultra-Reliable and Low-Latency Communications (URLLC)* e *Massive Machine-type Communications (mMTC)*, assim como outros cenários com os seus requisitos específicos. Para isto, espera-se que as suas métricas *Key Performance Indicator (KPI)* incluam: cobertura melhorada e aumentada ou quase 100% de cobertura para ligações a qualquer momento, em qualquer lugar, ritmos de dados 10 a 100 vezes mais elevados para clientes, baixa latência *End-to-End (E2E)* e uma poupança de energia superior a 90% [8]. Graças a tecnologias impulsionadoras como *Software-Defined Networking (SDN)* e *Network Functions Virtualization (NFV)*, uma visão do 5G que vai de encontro a estes requisitos é a de um sistema *service-oriented*, capaz de lidar com uma variedade de serviços com diferentes requisitos e dispositivos sobre uma infraestrutura comum. Uma possível arquitetura de rede deste tipo divide-se em três camadas: infraestrutura (computação, armazenamento e rede), função de rede (redes virtuais e *Virtual Network Function (VNF)*) e serviço (*Application Programming Interface (API)* para consumo) [14].

De modo a concretizar uma visão 5G, é necessário que a rede tome diferentes formas de acordo com o serviço em questão, levando naturalmente à noção de *network slicing*. Com *slicing*, uma rede física 5G é dividida em múltiplas redes lógicas isoladas (*slices*), constituídas por recursos das três camadas dedicados a diferentes tipos de serviços. Desta forma, as redes 5G suportam *multi-service*, *multi-tenancy* (em que *tenant* corresponde a um utilizador ou grupo de utilizadores com privilégios sobre um recurso partilhado) e elevada *Quality of Service (QoS)*/*Quality of Experience (QoE)*, oferecendo um aprovisionamento de serviços verdadeiramente diferenciado sobre uma estrutura partilhada subjacente [2]. A aplicação de *network slicing* envolve alguns desafios técnicos, tendo em

conta que o espetro é um recurso limitado na *Radio Access Network (RAN)* e a utilização de VNFs depende do poder computacional disponível na *Core Network (CN)* [27], sendo ainda necessário coordenar de forma eficiente o transporte de tráfego das várias *slices* entre RAN e CNs numa infraestrutura de transporte partilhada. Assim, é importante encontrar algoritmos inteligentes capazes de garantir QoS suficiente em cada *slice* e orquestrar a criação de *slices*, alocando os respetivos recursos espetrais, computacionais e de transporte de tráfego de cada *slice*.

Foram feitos múltiplos estudos de modo a resolver a alocação dinâmica de recursos com soluções matemáticas [23, 10, 39], mas estes problemas de otimização tradicionais tornam-se rapidamente intratáveis e não escalam para infraestruturas maiores do que alguns nós. Uma potencial solução alternativa é *Reinforcement Learning (RL)*. RL é um tipo de *Machine Learning (ML)* em que um agente aprende a executar ações ótimas a partir de observações de estado de um ambiente complexo e das recompensas (*rewards*) obtidas, indicativas da qualidade das ações tomadas. A motivação para utilizar este tipo de algoritmo provém do êxito da sua aplicação em problemas de controlo nas áreas de robótica, condução autónoma e vídeo-jogos [45]. No entanto, os algoritmos de RL não são eficientes em situações com um espaço de estados grande por ser necessário armazenar todos esses estados. O sucesso do *Deep Learning (DL)* em classificação de imagens [16], derivado do facto de conseguir aprender a concentrar-se nas caraterísticas importantes de um estado, atraiu alguma curiosidade sobre a sua aplicação em RL, dando origem ao *Deep Reinforcement Learning (DRL)*.

1.1 Objetivos

Com esta dissertação pretende-se fazer um estudo do estado-da-arte sobre a alocação dinâmica de recursos em *slices* baseada em agentes de DRL em vez de nas técnicas tradicionais de otimização. Este problema coloca-se a vários níveis da arquitetura, desde o acesso rádio na RAN à alocação de recursos computacionais nos pontos de computação, nas CNs, passando pela colocação do tráfego das *slices* de forma dinâmica e otimizada nas redes de transporte entre RANs e CNs. Envolve ainda dois momentos distintos que devem ser considerados: o primeiro é a admissão de novas *slices* na rede que deve ser feita de forma a maximizar o lucro e a utilização da rede sem comprometer os acordos de nível de serviço *Service Level Agreement (SLA)* de cada *slice*. O segundo é o redimensionamento de cada *slice* durante o seu período de operação de modo a permitir a utilização dinâmica dos recursos. Em todas estas dimensões se aplicam as dificuldades descritas, sendo o DRL uma abordagem promissora para este tipo de problema.

Este trabalho foca-se no problema da admissão de novas *slices* na rede de transporte entre a RAN e a CN, otimizando conjuntamente a utilização da rede e o lucro do operador. Propõe-se a utilização do simulador de redes com *hosts* em *containers*, Containernet, juntamente com o controlador SDN *Ryu* para emular uma rede com um comportamento

fiel à realidade. Os agentes DRL que admitirão pedidos na rede com base no seu estado implementam uma *Deep Q-Network (DQN)* e uma variação desta, *dueling* DQN.

1.2 Estrutura do Documento

O restante desta dissertação é organizado da seguinte forma:

- A Secção 2 começa com uma introdução dos conceitos básicos de RL sobre os quais DRL constrói, seguindo-se uma apresentação das bases do funcionamento de DL e alguns dos seus métodos mais relevantes. Termina com um estudo do estado-da-arte das arquiteturas de *network slicing* e métodos de DRL utilizados para a alocação dinâmica de recursos em *slices*, dividido em *slices* geridas por operadoras ou por *tenants*.

- A Secção 3 apresenta uma formalização do problema do controlo de admissão de *slices* na rede de transporte considerando o estado dos caminhos, começando por descrever matematicamente os pedidos. De seguida é explicado o funcionamento da seleção de caminhos entre pontos na rede, juntamente com uma descrição do modelo do sistema como um *Markov Decision Process*. Finalmente, é apresentado o algoritmo de treino do agente.

- A Secção 4 descreve a implementação do ambiente através do *OpenAI Gym*. É descrita a criação dos agentes DRL utilizando o *framework* de DL *PyTorch*, seguida da arquitetura utilizada para simulação da rede *Containernet* e o controlador SDN *Ryu*.

- A Secção 5 contém todos os parâmetros usados, desde a topologia da rede aos parâmetros das redes neuronais, seguidos dos resultados obtidos, quer nos treinos, quer nos testes, e discussão destes. Os dois agentes utilizados, com uma DQN e uma *dueling* DQN, são comparados com diferentes políticas de admissão de *slices* de modo a compreender se a sua utilização é adequada a este problema numa simulação de rede.

- Finalmente, a Secção 6 encerra a dissertação com um sumário das conclusões retiradas sobre a aplicação de DQNs e *dueling* DQNs para resolver o problema descrito, terminando com algumas sugestões de trabalhos futuros para estudar métodos que potencialmente levem a melhores resultados.

2

Estado de Arte

Neste capítulo é feita uma apresentação do DRL, começando com uma descrição de *Reinforcement Learning*, *Deep Learning* e a sua combinação. Finalmente, é feito um estudo do estado-da-arte das arquiteturas de *network slicing* e métodos de DRL usados para a alocação dinâmica de recursos em *slices*.

2.1 Reinforcement Learning

As tarefas em que ações têm de ser tomadas com base no estado do sistema são designadas problemas de controlo. RL é um formalismo genérico para representação e resolução deste tipo de problemas, encapsulando uma grande variedade de algoritmos aplicáveis, em que um agente interage com um ambiente e é incentivado no caso de levar o sistema a comportar-se como desejado, ou penalizado no caso contrário.

Em RL, um ambiente é qualquer processo dinâmico que produza dados relevantes para atingir um objetivo. Este é amostrado em intervalos de tempo discretos, criando um espaço de estados/observações S. Ao algoritmo que recebe estados dá-se o nome de agente e este interage com o ambiente através de ações pertencentes a um espaço de ações A, que pode ser discreto ou contínuo. Uma ação a_t tomada no estado s_t pode ser descrita pelo par estado-ação (s_t, a_t). Finalmente, a *reward*, r_t é uma representação numérica da qualidade da transição entre estados $s_t \rightarrow s_{t+1}$, podendo ser positiva ou negativa, o que permite ao agente aprender a agir da melhor forma. Estes conceitos são representados na Figura 2.1.

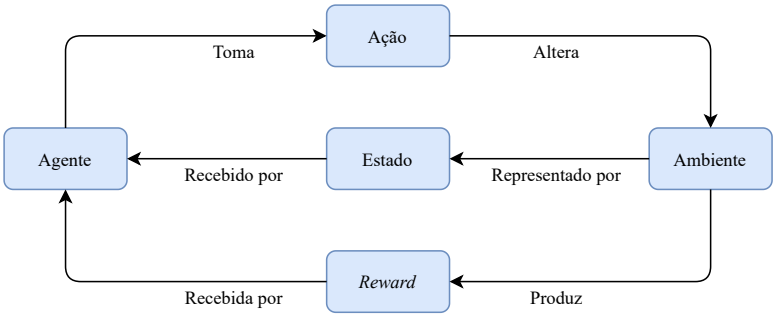

Figura 2.1: *Framework* RL.

Em RL, uma transição de estados é chamada de experiência e o conjunto destas do início ao fim de um ciclo de execução do ambiente designa-se época/episódio. Idealmente, o ciclo representado na Figura 2.1 é repetido durante várias épocas até que o agente compreenda o seu ambiente e seja capaz de tomar boas ações em qualquer estado.

2.1.1 Markov Decision Process

Modelar uma tarefa de controlo como um MDP é fundamental em RL. Num MDP, o estado atual do sistema é suficiente para escolher as ações ótimas que maximizarão as *rewards*, não sendo necessário guardar todos os pares (s_t, a_t) em memória. Este tipo de modelo é composto por um espaço de estados S, um espaço de ações A, as probabilidades de transição entre estados, dependentes da ação tomada e do estado atual, e as *rewards* por atingir cada estado. A Figura 2.2 representa um diagrama MDP simples de modo a ilustrar o conceito, com um espaço de estados $S = \{s_0, s_1, s_2\}$, um espaço de ações $A = \{a_0, a_1\}$ e as *rewards* {-1, +1, +5}.

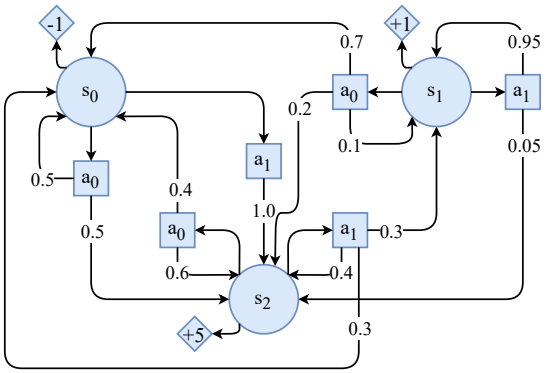

Figura 2.2: Diagrama MDP.

Num caso real, o agente pode ou não conhecer o modelo do sistema. No caso de conhecer ou procurar construir um modelo do ambiente, como no xadrez, é designado *model-based*, e no caso contrário, em que o agente apenas decide como agir em cada estado sem recorrer a um modelo do ambiente para fazer previsões, dá-se o nome *model-free*.

O objetivo do agente é maximizar as *future rewards* (*return*, R_t) de cada episódio, definidas na Equação 2.1, em que T é o número de experiências num episódio. O *discount factor*, γ, pode ser qualquer valor entre 0 e 1 e dita o quanto o agente desconta *rewards* futuras quando toma uma decisão. Este desconto é exponencial (γ^n), ou seja, se γ for igual a 1 (1^n), as *rewards* futuras são consideradas infinitamente, o que só é possível em cenários com episódios relativamente curtos, mas qualquer valor inferior a 1 (0.999^n, por exemplo) implica que eventualmente estas passam a ser 0, e assim é possível limitar o quanto e até quando o agente considera *rewards* futuras.

$$R_t = \sum_{t=0}^{T} \gamma^t r_t \tag{2.1}$$

2.1.2 Policy vs. Value

O agente recebe uma *reward* r_t por ter tomado a ação a_t no estado s_t, levando ao novo estado s_{t+1}, e o seu objetivo é maximizar as *future rewards*. É importante reforçar que é a transição $s_t \rightarrow s_{t+1}$ que produz a *reward* e não a ação por si. Assim, o agente procura escolher as ações que produzam as transições que levem a maximizar as *future rewards*. Esse treino pode ser feito direta ou indiretamente. No primeiro caso, o agente aprende quais as melhores ações por estado, levando à noção de *policy functions*. No segundo caso, o agente aprende que estados são mais valiosos e deduz que ações tomar de modo a atingi-los, seguindo o conceito de *value functions*.

Uma *policy*, π, é a estratégia que o agente utiliza para tomar ações num ambiente. Matematicamente, é uma função que mapeia estados para uma distribuição de probabilidades das suas ações, e representa-se de acordo com a Equação 2.2, em que s simboliza um estado e $Pr(A|s)$ é a distribuição de probabilidades sobre um espaço de ações A, dado s.

$$\pi : s \rightarrow Pr(A|s) \tag{2.2}$$

Um desafio na escolha de uma *policy* é equilibrar *exploration/exploitation*. É importante explorar o ambiente, ou seja, tomar uma ação que não a ótima, de modo a aprender mais sobre este, mas também é necessário tirar partido das melhores ações conhecidas para maximizar as *rewards*. Se o *output* de uma *policy* consistir numa distribuição de probabilidades em vez de uma única ação, as melhores ações têm uma elevada probabilidade de serem escolhidas mas as restantes não são excluídas, permitindo ao agente explorar o ambiente.

Por outro lado, *value functions* calculam *state-values*, ou seja, mapeiam um estado para as *expected rewards* de começar neste e seguir uma determinada *policy* π, como representado na Equação 2.3.

$$V_\pi : s \rightarrow E[R|s, \pi] \tag{2.3}$$

Expected rewards são a média a longo prazo do *return*, e podem ser usadas para generalizar a definição de *state-value* na Equação 2.4.

$$V(s_t) = E[R_t|s_t] = E[\sum_{t=0}^{T} \gamma^t r_t] \tag{2.4}$$

A *value function* depende da *policy* porque esta dita que ações são tomadas. Por exemplo, se a *policy* consistir em selecionar ações aleatoriamente, o *state-value* será baixo. No entanto, mesmo com ações aleatórias, eventualmente existiriam experiências suficientes e o algoritmo conseguiria inferir os *state-values*. A grande utilidade da *policy* nestes métodos consiste em reduzir a quantidade de dados necessária para treino.

A partir das *policies* surgem os algoritmos *policy-based* que procuram aprender que ações o agente deve executar. Contrariamente, algoritmos *value-based*, que calculam o *value* de cada estado e agem a partir de uma *policy* derivam das *value functions*. Surgem ainda as definições de *on-policy*, em que o agente procura aprender uma *policy* enquanto

que a utiliza simultaneamente para adquirir dados, ou seja, requer dados recentes, e *off-policy*, em que a escolha de *policy* não impede que os *state-values* sejam aprendidos corretamente e conseguem aprender com dados antigos.

2.1.3 Q-learning

Um algoritmo *off-policy* particularmente eficaz baseado em *value functions* é o *Q-learning*. Depende de uma *Q-function* ou *action-value function* que calcula o *Q-value* (também referido como *action-value* ou *state-action-value*), ou seja, mapeia um par *(s, a)* para a *expected reward* de tomar a ação *a* no estado *s*, tendo em conta a *policy* π. Esta noção pode ser representada de acordo com a Equação 2.5.

$$Q_\pi : (s, a) \rightarrow E[R|a, s, \pi] \tag{2.5}$$

Um *optimal state-value* pode ser definido recursivamente (através de *bootstrapping*), a partir da equação de *Bellman*, como a ação que origina a maior *reward* imediata possível somada à *reward* descontada do próximo estado, como na Equação 2.6.

$$V(s_t) = \max_{a_t \in A}(E[r_t + \gamma V(s_{t+1})]) \tag{2.6}$$

Construindo sobre o conceito de *optimal state-value*, o *Q-value* é aproximado com *bootstrapping* segundo a Equação 2.7, em que α é o *learning rate* usado para determinar o tamanho do ajuste do *Q-value* e a expressão dentro dos parênteses é designada de Temporal Difference (TD) *error*. O objetivo em *Q-learning* é atualizar os *Q-values* para que o TD *error* se aproxime de 0.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a_{t+1} \in A}(Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t)) \tag{2.7}$$

2.1.4 Policy Function

Q-learning necessita que se escolha uma *policy* para tomar ações, mas esse passo pode ser evitado ao implementar uma *policy function* que retorna a probabilidade de escolher cada ação em vez de *state-values* ou *Q-values*. Este método também é mais eficaz em situações com espaços de ações grandes ou até contínuos [26].

No início do treino, a distribuição de probabilidades produzida é provavelmente uniforme, de modo a maximizar a exploração do ambiente. Eventualmente, essa distribuição evolui de modo a favorecer a escolha de determinadas ações em cada estado. Quando existem várias ações com probabilidades diferentes de 0, 1 e entre si, a *policy function* é estocástica. Se existir uma ação ótima para um dado estado, ou seja, a probabilidade de uma ação é 1 e todas as outras são 0, esta designa-se uma distribuição de probabilidades degenerada e a *policy function* é determinística.

Este tipo de algoritmos beneficia vastamente da associação com DL, dando origem aos métodos *policy gradient* descritos em maior pormenor na Secção 2.3.6.

2.2 Deep Learning

Redes neuronais são conjuntos de neurónios artificiais e as suas respetivas ligações. Cada neurónio, como representado na Figura 2.3, recebe um ou vários sinais de entrada x, sendo estes multiplicados pelos seus respetivos pesos w e, se necessário, posteriormente somados a um *bias* b. Finalmente, é feito o somatório dos produtos obtidos e é aplicada uma função de ativação não-linear, por exemplo a função sigmóide. DL [16] consiste na utilização de uma *deep neural network*, visível no lado direito da Figura 2.2, que é uma rede neuronal com vários agrupamentos de neurónios, chamados camadas, densamente ligadas entre si (*dense layers*). Matematicamente, cada camada consiste num produto interno entre matrizes (matrizes de dados com matrizes de pesos/parâmetros) seguida de uma função de ativação. Existe sempre uma camada de entrada (*input layer*), pelo menos uma camada escondida (*hidden layer*) e uma camada de saída (*output layer*).

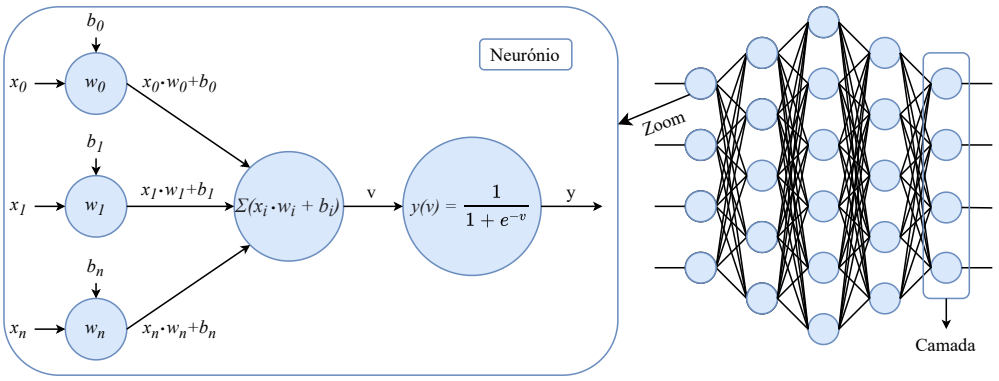

Figura 2.3: Neurónio numa *deep neural network*.

O treino de redes neuronais requer uma *loss function* que representa o erro entre um valor previsto e o real correspondente. As mais comuns são a *Mean-Squared Error (MSE) loss*, que calcula o erro médio quadrático, e a *Cross-Entropy loss*, baseada na divergência de *Kullback-Leibler (KL)* usada para comparar distribuições [26].

Uma vez aplicada a *loss function*, o valor resultante é utilizado para calcular o ajuste aos parâmetros da rede para aproximar o valor previsto do real. Este cálculo do ajuste a múltiplas variáveis é feito através do algoritmo de otimização *gradient descent*, que calcula a derivada da *loss function* em ordem aos parâmetros da rede de modo a compreender em que "direção"deslocar os parâmetros para se aproximar do mínimo da função. O *gradient descent* pode ser *Stochastic Gradient Descent (SGD)* se for calculado sobre cada experiência, *mini-batch* se sobre um conjunto ou *batch* se sobre todas as experiências de um episódio.

Utiliza-se um *learning rate* para determinar a escala dos ajustes na direção do mínimo. Um *learning rate* muito pequeno aumenta o tempo de convergência mas um *learning rate* demasiado alto pode resultar em flutuações em torno do mínimo ou até divergência.

Finalmente, após calculado o gradiente, é utilizado o método de *backpropagation* [36] que percorre a rede de trás para a frente ajustando cada parâmetro de acordo com a

Equação 2.8, em que L representa a *loss function*.

$$w_n = w_n - \alpha \cdot \frac{\partial L}{\partial w_n} \tag{2.8}$$

Atualmente recomenda-se a utilização de um *optimizer* disponível nos *frameworks* de DL para fazer ajustes como aplicar textitlearning rates maiores a parâmetros atualizados com menor frequência e evitar que a rede fique presa num mínimo local [12].

O processo de treino deste tipo de redes é feito durante n épocas passando os dados pela rede, calculando o *gradient descent* da *loss function* sobre as matrizes de pesos, e ajustando-as através de *backpropagation* com um *optimizer*.

Um dos principais impulsionadores de DL é a abundância de recursos de computação. Com *datasets* cada vez maiores e modelos cada vez mais complexos, os recursos necessários para treinar um algoritmo aumentam significativamente. Os *frameworks* de DL atuais suportam paralelismo o que permite a distribuição das computações por vários *cores* de uma *Graphics Processing Unit (GPU)*, possivelmente em múltiplos nós, que reduz o tempo de treino de redes muito mais complexas[28].

DL é tipicamente aplicado a tarefas de classificação, pertencentes à categoria de *offline supervised learning*, em que o agente aprende com base num conjunto de dados com a resposta correta de cada caso. Consiste na utilização de modelos paramétricos na forma de *deep neural networks* que permitem ao agente aprender representações em camadas dos dados e assim abstrair-se dos detalhes e concentrar-se nas caraterísticas importantes. Esta propriedade é designada composicionalidade, o que significa que dados complexos são tratados como a combinação de componentes mais simples.

2.2.1 Convolutional Neural Network

A *Convolutional Neural Network (CNN)* [4] faz parte dos algoritmos de DL e é tipicamente aplicadas em classificação de imagens, mas podem ser usadas para analisar qualquer tipo de dados multi-dimensionais. As CNNs aprendem relações complexas entre os índices dos dados utilizando uma operação linear chamada convolução.

A convolução baseia-se na identificação de caraterísticas (*features*) nos dados através da aplicação de um filtro multi-dimensional (*kernel*) ao longo destes, da esquerda para a direita até ao fim de uma linha, passando para a seguinte, também percorrida da esquerda para a direita. Os dados são convolvidos com o *kernel*, o que significa que este é deslizado ao longo destes, calculando o produto interno entre a área dos dados sobreposta pelo *kernel*, como representado na Figura 2.4.

Quando se aplica convolução é típico adicionar *padding* às margens dos dados tendo em conta as situações em que o *kernel* está posicionado parcialmente fora dos dados originais. O *padding* é normalmente feito acrescentando zeros às margens, duplicando os dados interiores para o exterior da margem, espelhando o interior, entre outros métodos. Para além disso é comum aumentar o *stride* (número de índices pelo qual o *kernel* se desloca entre passos da convolução) de modo a diminuir o tamanho do *output* produzido.

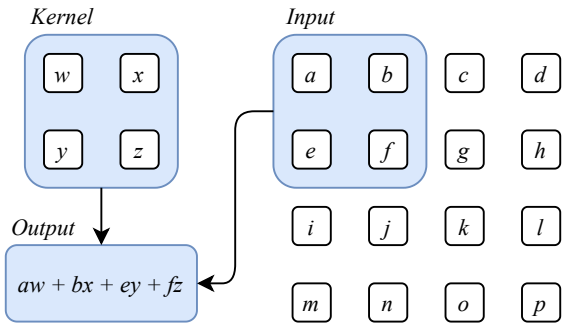

Figura 2.4: Exemplo de um passo de convolução sobre dados bidimensionais.

A operação de convolução é normalmente seguida por uma função de ativação não-linear e uma etapa de *pooling* que asseguram as propriedades não-lineares da rede e sumarizam o *output* enquanto realçam as *features* importantes [16], por exemplo, calculando o máximo de um filtro 2x2 deslizado pelos dados.

2.2.2 Generative Adversarial Networks

Uma *Generative Adversarial Network (GAN)* [17] baseia-se na existência de duas *deep neural networks*, em que a primeira funciona como *generator* e a segunda como *discriminator*. As duas redes competem uma com a outra: a partir de dados aleatórios o *generator* tenta gerar dados falsos que sejam difíceis de distinguir dos reais para o *discriminator*, e este tenta detetar os falsos. Este conceito é representado na Figura 2.5.

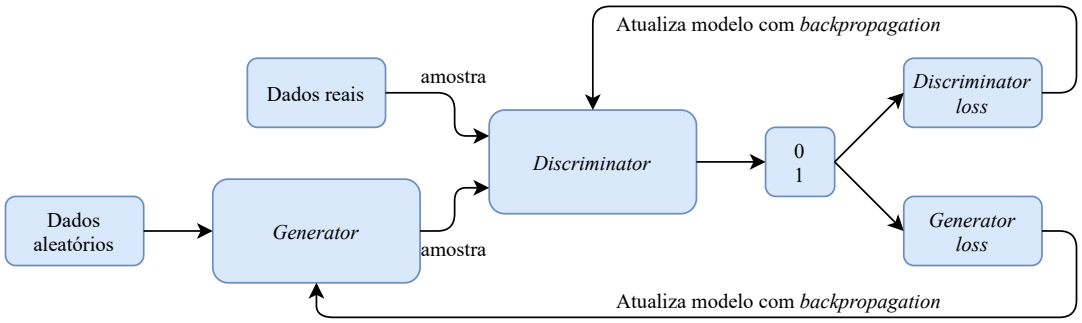

Figura 2.5: Funcionamento básico de *generative adversarial networks*.

As redes são treinadas individualmente utilizando *optimizers* separados. Primeiro, é criado um grupo de dados aleatórios e passado pelo *generator*. O *discriminator* é treinado uma primeira vez sobre os dados reais e uma outra sobre os dados gerados pelo *generator*, e apenas os seus parâmetros são atualizados. De seguida, para treinar o *generator*, o seu *output* passado pelo *discriminator* é usado para calcular o erro entre os valores obtidos e um conjunto de *labels* marcadas a 1 (reais). Desta forma, o *generator* aprende a confundir o *discriminator* [26].

2.3 Deep Reinforcement Learning

Muitos algoritmos de RL dependem do armazenamento e processamento das suas experiências em tabelas, o que significa que em ambientes complexos estas podem tornar-se computacionalmente intratáveis. No entanto, utilizando algoritmos de DL, que possuem um número finito de parâmetros, consegue-se comprimir qualquer estado possível de uma forma eficientemente processável e usar essa representação para tomar decisões [45].

Com a aplicação de DL em RL, DRL, o agente composto por uma *deep neural network* é capaz de aprender a compreender relações complexas entre pares *(s, a)* e *rewards* sem ter de guardar todas as experiências numa tabela, sendo o conhecimento guardado sobre a forma de parâmetros/pesos da rede.

2.3.1 Deep Q Network

O conceito de *Deep Q-Learning (DQL)* apresentado em [30] com uma DQN constrói sobre o *Q-learning* apresentado na Secção 2.1.3 e atinge-se substituindo o agente por uma *deep neural network* que funciona como *Q-function* e devolve a *reward* prevista para uma ação num estado. Inicialmente as previsões são aleatórias, mas à medida que são tomada ações em vários estados, o algoritmo aprende a prever *rewards* com maior precisão para cada caso, maximizando-as. O algoritmo base consiste em:

1. Obter um estado s_t e passá-lo à DQN para calcular o *Q-value* para cada par (s_t, a_t).
2. Tendo em conta os valores obtidos e a *policy* a usar, tomar uma ação no ambiente, que produzirá uma *reward* r_t e um novo estado s_{t+1}.
3. Calcular o *Q-value* para cada par no novo estado e guardar $\underset{a_{t+1}}{max}(Q(s_{t+1}, a_{t+1}))$.
4. Calcular o *MSE-error*. Se o estado s_{t+1} for terminal, a *loss function* é $\zeta = (Q(s_t, a_t) - r_t)^2$. Caso contrário torna-se $\zeta = (Q(s_t, a_t) - (r_t + \gamma \underset{a_{t+1} \in A}{max}(Q(s_{t+1}, a_{t+1}))^2$.
5. Atualizar $Q(s_t, a_t)$ utilizando SGD sobre os pesos da rede.
6. Repetir os passos 1-6 enquanto o ambiente estiver ativo. Quando este terminar, reiniciá-lo e repetir durante *n* épocas ou até convergir.

Uma *policy* tipicamente utilizada em *Deep Q-learning* é a ϵ-*greedy* (*epsilon-greedy*), em que com probabilidade ϵ é escolhida uma ação aleatoriamente e com probabilidade $1-\epsilon$ é escolhida a ação com o maior *Q-value*. ϵ é normalmente iniciado a 1 para permitir grande exploração durante a fase inicial do treino, e é gradualmente decrementado até um valor muito baixo, forçando o agente a tomar as ações com o melhor *Q-value* nas etapas mais avançadas.

Por ser um método *online*, DQL sofre de *catastrophic forgetting* [45], ou seja, pares *(s, a)* muito semelhantes com resultados muito diferentes sobrepõem-se durante a aplicação de SGD por não serem independentes e identicamente distribuídos (i.i.d.) [26], o que

confunde o agente e incapacita-o de aprender corretamente. Este problema pode ser resolvido recorrendo a um *experience replay buffer* que guarda tuplos (s_t, a_t, r_t, s_{t+1}). O *buffer* é populado com experiências até um ponto definido, a partir do qual um *batch* de tamanho específico destes tuplos é aleatoriamente selecionado para treinar a DQN, em vez de usar apenas a experiência mais recente. Se o *buffer* ficar cheio, as experiências são substituídas segundo a ordem *First-In-First-Out (FIFO)*.

A partir do momento em que há experiências suficientes no *experience replay buffer*, os pesos da DQN são atualizados após todas as ações. Assim, ao aplicar *bootstrapping* para aproximar $Q(s_t, a_t)$, podem-se alterar indiretamente os valores de $Q(s_{t+1}, a_{t+1})$ e de outros estados próximos, o que provoca alguma instabilidade na aprendizagem [26]. A solução proposta em [29] é duplicar a DQN, existindo uma *Q-network* normal e uma cópia chamada *target network* ($\hat{Q}$-*network*), e cada instância tem os seus próprios parâmetros, θ_Q e θ_T, respetivamente. A *target network* apenas é usada para calcular o valor $\max_{a_{t+1}}(\hat{Q}(s_{t+1}, a_{t+1}))$ durante o treino, e o $Q(s_t, a_t)$ resultante é *backpropagated* pela *Q-network* principal. Desta forma reduziu-se o efeito que as experiências recentes tinham sobre a seleção de ações porque a *target network* não é atualizada após todas as ações, aumentando a estabilidade. Com uma frequência definida, a *target network* é sincronizada com a *Q-network*, ou seja, $\theta_T = \theta_Q$. O ciclo completo é apresentado na Figura 2.6. Este algoritmo é a base de grande parte dos algoritmos de DQL, podendo ter variações como as apresentadas nas próximas subsecções.

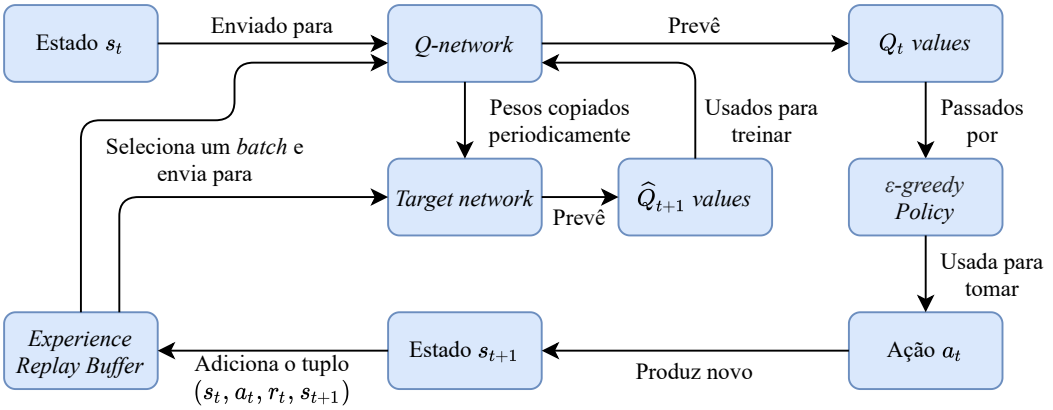

Figura 2.6: *Deep Q-learning* com ϵ-*greedy policy*, *experience replay* e *target network*.

2.3.2 Double Deep Q Network

Em [40] é mostrado que o algoritmo básico de DQN tende a sobrestimar os *Q-values*, o que pode prejudicar a sua *performance*. Este problema tem origem na operação *max* da equação de *Bellman*, em que a *target network* é usada para calcular os *Q-values* e a ação do próximo estado para treino da *Q-network*. A solução proposta é usar a *Q-network* principal para selecionar ações para o próximo estado e a *target network* apenas para calcular os

Q-values, de acordo com a Equação 2.9.

$$Q(s_t, a_t) = r_t + \gamma \max_{a_{t+1} \in A} (\hat{Q}(s_{t+1}, \operatorname*{argmax}_{a_{t+1}} (Q(s_{t+1}, a_{t+1})))) \tag{2.9}$$

2.3.3 Dueling Deep Q Network

Dueling Deep Q-Networks seguem o conceito apresentado em [43], em que os *Q-values* que a rede procura aproximar podem ser divididos em *state-value*, $V(s_t)$, e *advantage*, $A(s_t, a_t)$, das ações num estado, ou seja, $Q(s_t, a_t) = V(s_t) + A(s_t, a_t)$. A *advantage* indica quanta *reward* extra pode originar cada ação num dado estado.

O método propõe também uma mudança de arquitetura, em que a rede deve produzir dois *outputs* diferentes: $V(s_t)$ e $A(s_t, a_t)$. De seguida os valores são somados, obtendo-se $Q(s_t, a_t)$, usado para o treino.

Para que a aprendizagem funcione corretamente, a média das *advantages* para cada estado tem de ser 0. Isso pode ser feito de acordo com a Equação 2.10. Na Figura 2.7 é representado um exemplo de arquitetura de uma *dueling* DQN.

$$Q(s_t, a_t) = V(s_t) + A(s_t, a_t) - \frac{1}{N} \sum_k A(s_t, k) \tag{2.10}$$

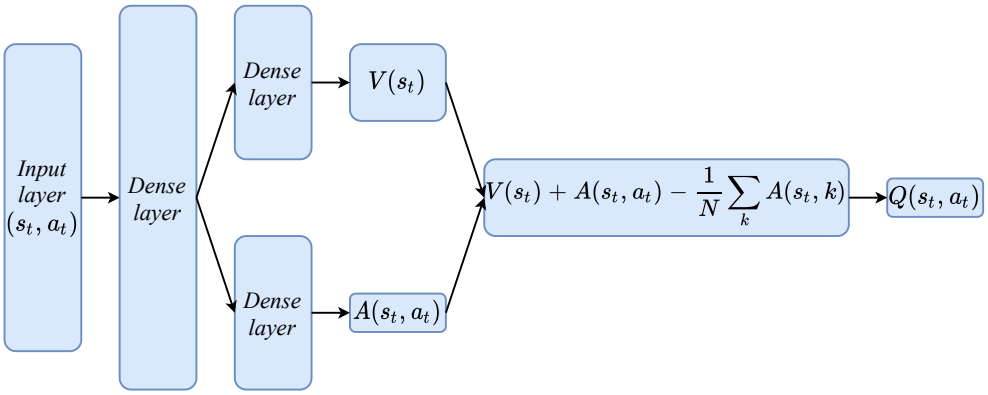

Figura 2.7: Arquitetura *dueling DQN*.

2.3.4 Discrete Normalized Advantage Function

Em [19] é introduzido o conceito de *Normalized Advantage Function (NAF)* como potencial solução para o problema de DQL com espaços de ações contínuos, que, tal como a arquitetura *dueling* apresentada na Secção 2.3.3, consiste em calcular $Q(s_t, a_t) = V(s_t) + A(s_t, a_t)$, mas para um espaço contínuo.

Para aproveitar a capacidade da NAFs de trabalhar com espaços de ações muito grandes mas generalizada para problemas discretos, [34] propõe combiná-las com um algoritmo *k-Nearest Neighbour (k-NN)*, neste caso 1-NN. Este determina uma ação discreta

como a ação contínua mais próxima, permitindo selecionar ações discretas a partir de valores contínuos e dando origem à *Discrete Normalized Advantage Function (DNAF)*.

2.3.5 Deep Distributional Q Network

Uma *Deep Distributional Q-Network (DDQN)*, como proposta em [7], é uma DQN que produz uma distribuição de probabilidades de *Q-values* para cada par (s_t, a_t) em vez de um único *Q-value*. Esta alteração é uma melhoria para ambientes em que as *rewards* têm alguma aleatoriedade porque em DQL os *Q-values* aprendidos são apenas *expected rewards*, perde-se alguma informação sobre as dinâmicas do ambiente, como a variância dos valores.

Neste tipo de DQN, a equação de *Bellman* é generalizada na sua forma distribucional, e usada como alvo no ajuste das distribuições de acordo com a Equação 2.11, em que Z e R representam as distribuições de probabilidades dos *Q-values* e *rewards*, respetivamente. No caso de um estado terminal, o objetivo é apenas $R(s_t, a_t)$.

$$Z(s_t, a_t) = R(s_t, a_t) + \gamma Z(S_{t+1}, A_{t+1}) \tag{2.11}$$

Para atualizar os parâmetros da DDQN através de *gradient descent* é necessária uma *loss function* que calcule a diferença entre a distribuição de *Q-values* prevista e a alvo. Para isso é usada *Cross-Entropy loss*, representada na Equação 2.12 com base na divergência de KL [45].

$$H(A, B) = -\sum_i A(x) \cdot log(B(x_i)) \tag{2.12}$$

Uma desvantagem deste método é não ser possível representar qualquer *Q-value*, arbitrariamente pequeno ou grande, porque o suporte (valores associados a cada probabilidade) é um espaço estático finito. Para resolver este problema, podem-se fixar os valores das probabilidades possíveis e aprender que suportes lhes correspondem. Este método é designado *Quantile Regression DQN (QR-DQN)* [11].

2.3.6 Policy Gradient

Uma maneira de melhorar as *policy functions* descritas na Secção 2.1.4 é substituindo o agente por uma *deep neural network*. Desta forma, a rede aceitará um estado e produzirá uma distribuição de probabilidades sobre as ações. De modo a fazer a evoluir essa distribuição, os parâmetros da rede são ajustados utilizando SGD para maximizar um *policy gradient* definido na Equação 2.13. Assim, o objetivo é minimizar a *loss function* descrita na Equação 2.14 [26].

$$\nabla J \approx E[\, Q(s, a) \nabla log \pi(a|s)] \tag{2.13}$$

$$\zeta = -Q(s_t, a_t) log \pi(a_t|s_t) \tag{2.14}$$

Nas equações anteriores, $Q(s_t, a_t)$ funciona como um escalar para aumentar as probabilidades de boas ações no início do episódio e diminuir as de ações perto do final deste

(devido ao *discount factor* usado no *bootstrapping* para o cálculo de $Q(s_t, a_t)$), e é aplicado um *log* sobre a *policy* para aumentar o alcance dos valores de $[0, 1]$ para $[-\infty, 0]$, o que permite computações mais precisas nas probabilidades [45].

O treino destas redes é substancialmente diferente do das DQNs, tendo em conta que é naturalmente um método de *offline learning*, porque o treino apenas é feito ao fim de n episódios. O treino deste método *policy gradient* chamado *REINFORCE* [26] segue os passos:

1. Executar n episódios e guardar os seus tuplos (s_t, a_t, r_t, s_{t+1}).
2. Para cada experiência t de cada episódio n calcular o *return* dos passos seguintes $Q_{n,t}$ ou $R_{n,t}$: $Q_{n,t} = \sum_k \gamma^k r_k$.
3. Calcular a *loss function* para todas as transições: $\zeta = -\sum_{k,t} Q_{k,t} log(\pi(s_{k,t}, a_{k,t}))$.
4. Executar ajustes sobre os pesos com SGD.
5. Repetir os passos 1-5 durante n épocas ou até convergir.

É importante referir que existe uma classe de algoritmos chamada *Actor-Critic* [18], que aproveita a eficácia das DQNs em ambientes com um espaço de estados discreto mas requerem a escolha de uma *policy-function* separada, com algoritmos *policy gradient*. Consiste num *actor* (*policy network*) que toma uma ação no ambiente, produzindo o próximo estado e a *reward* resultante. Ao receber o novo estado, o *critic* avalia a *advantage* da transição de estado, sendo esse sinal utilizado na *loss function* da *policy network*. Por outro lado, tal como em DQL, a *loss function* do *critic* baseia-se apenas das *rewards* do ambiente, sendo que nesta situação as *rewards* dependem das ações tomadas pelo *actor*.

2.4 Trabalhos Relacionados

A alocação dinâmica de recursos em *slices* já foi profundamente estudada enquanto problema de otimização tradicional [23, 10, 39] mas sem sucessos extraordinários devido à sua elevada complexidade e dimensão do espaço de estados. Recentemente foram feitos vários trabalhos que propõem o uso de DRL para resolver o problema. A sua definição depende do modelo de *network slicing*, da parte da rede em questão e das técnicas de DRL aplicadas. Ao longo desta secção serão apresentados os modelos mais comuns e os trabalhos que estabelecem o estado-da-arte sobre a utilização de DRL para resolver estes problemas, divididos em *slices* geridas por operadoras ou por *tenants*. Destaca-se o facto de nos artigos discutidos ser rara a utilização de algoritmos de *policy gradient* e *actor-critic*, dando-se um maior foco às DQNs, as suas variações e melhorias.

2.4.1 Slices geridas por operadoras

Segundo este modelo de *slicing*, cabe às operadoras a criação e disponibilização de *slices*. A operadora tem conhecimento do tipo de pedidos mais comuns e modelos custo/lucro

para cada *slice* ou categoria. O desafio consiste na distribuição de recursos entre *slices* de acordo com a necessidade dos utilizadores de cada tipo de *slice*. Os requisitos dos *slices* são normalmente estáticos ao longo do tempo. Assim, a operadora disponibiliza categorias de *slices* para diferentes tipos de serviços comuns como eMBB, URLLC e mMTC, e distribui por estas os utilizadores.

Em [27] pretende-se compreender se a aplicação de DRL em *network slicing* produz um QoE satisfatório enquanto consome recursos aceitáveis. DQL mostra-se adequado para esta situação em que os pedidos por serviços são inconstantes e não se conhece o modelo de tráfego. O problema da alocação de recursos é abordado de duas perspetivas: *slicing* de recursos rádio e de VNFs, como distinguidos na Figura 2.8. Nas RANs o espetro é um

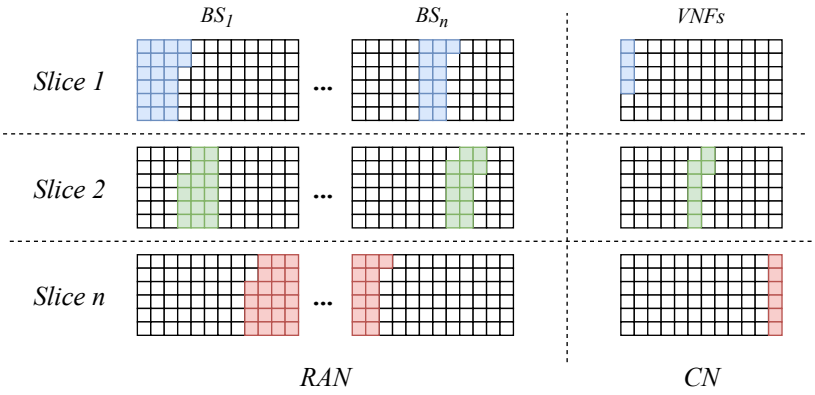

Figura 2.8: Arquitetura de *network slicing*.

recurso limitado, logo, o objetivo é maximizar a *Spectral Efficiency (SE)*, ou seja, a utilização dos recursos alocados, juntamente com a QoE oferecida aos utilizadores. A *performance* da DQN neste cenário foi testada simulando uma *Base Station (BS)* como agente, com três *slices* e 100 utilizadores que as partilham dinamicamente. O estado é o número de pacotes recebidos em cada *slice* durante um período de tempo. As ações correspondem à largura de banda alocada para cada *slice*, e a *reward* é a soma ponderada da SE e QoE nas três *slices*. O que se pretende é que o agente aprenda a alocar dinamicamente recursos para as 3 *slices* de acordo com o número de utilizadores a usar cada tipo de serviço. Os resultados obtidos validam a flexibilidade de DQL em cenários com recursos limitados, garantindo a QoE por utilizador necessária. Por outro lado, o *slicing* baseado em prioridades de recursos computacionais nas CNs para VNFs, de modo a encaminharem corretamente pacotes de uma *slice* com um tempo de espera mínimo, também constitui um problema. É simulado um cenário com 3 *service function chains* (SFCs) com as mesmas capacidades mas que recorrem a diferentes unidades *Computation Processing Unit (CPU)* e portanto resultam num sistema de agendamento baseado em prioridades com 3 categorias com tempos de espera diferentes. O estado é a categoria e o tempo de chegada dos últimos 5 *flows* em cada SFC. Uma ação corresponde à SFC alocada para um *flow* num determinado tempo e a *reward* é a soma ponderada, de acordo com a categoria, dos tempos médios nos três SFCs.

Deste cenário conclui-se que DQL consegue alocar recursos de computação e reduzir os tempos de espera ao servir primeiro utilizadores com maior prioridade. Este artigo mostra que, segundo os resultados das comparações entre alguns métodos tradicionais de *slicing* e DQL, este consegue aprender relações complexas entre procura e oferta de recursos, e aumentar a eficiência de *network slicing*. No entanto, deve-se ter em conta que num cenário real o número de utilizadores e requisitos dos *slices* são dinâmicos.

O método proposto em [27] não tem em consideração os efeitos do *random noise* no cálculo de SE e *SLA Satisfaction Ratio (SSR)*. Para atenuar o risco de estimar incorretamente os *Q-values* neste cenário, [22] introduz uma *Distributional* DQN como proposta em [7], que permite melhorar as estimações. Este método é combinado com GANs com o objetivo de aprender a gerar distribuições de *Q-values*, criando uma *GAN-powered Deep Distributional Q Network (GAN-DDQN)*. É ainda testado o efeito do uso de uma arquitetura *dueling* [43]. Os dois algoritmos, com e sem arquitetura *dueling*, são comparados no mesmo cenário com a DQN de [27] e mostram melhorias em termos de *performance* e estabilidade.

Em [34] é apontado que uma falha comum em trabalhos relacionados é que se considera um espaço de ações discreto muito limitado, ou seja, a alocação de recursos não é muito flexível. Para o resolver, é considerado um problema semelhante ao de [27, 22] para RAN *slicing*, neste caso resolvido utilizando uma DNAF. O contributo deste trabalho é o facto de o algoritmo se adaptar melhor a um grande espaço de ações com DNAFs, aumentando o ritmo de convergência das DQNs nestas situações e satisfazendo os requisitos de serviço através de uma alocação de recursos mais precisa.

O trabalho em [1] também considera RAN *slicing* da mesma forma que [27, 22, 34], mas desta vez para um número variável de *slices* por BS, que deve alocar recursos para cada *slice* dependendo do número de utilizadores. Neste cenário o estado é uma combinação de métricas que refletem a satisfação dos requisitos atuais, utilização de recursos e estado do *slice* em termos de tempo de chegada, transmissão e *buffering* de pacotes. O espaço de ações consiste em aumentar ou diminuir o número de recursos rádio dedicados a um *slice*, e a *reward* combina satisfação de requisitos com taxa de utilização de recursos. A novidade deste estudo é o uso de um método chamado *Ape-X* que utiliza múltiplos *actors* para obter experiências do ambiente concorrentemente e as guardam num *experience replay buffer* comum, de modo a aprender de forma distribuída e reduzir assim o tempo de aprendizagem. Este algoritmo combina os métodos *dueling network* e *double* DQN (apresentado em [40]) para diminuir o efeito da sobrestimação de *Q-values*. A principal vantagem relativamente aos trabalhos anteriores, é que o agente não precisa de ser treinado novamente no caso de o número de *slices* mudar. Os resultados mostram que para além de aumentar a flexibilidade de *slicing*, o método maximiza o número de *slices* em que os requisitos são satisfeitos enquanto minimiza os recursos alocados, independentemente do número de slices.

No trabalho [44] o *slicing* é feito em dois níveis, como apresentados na Figura 2.9, em que CUs são *cloud units*, AUs são *access units* e UEs são *user equipments*. Na *core cloud* é

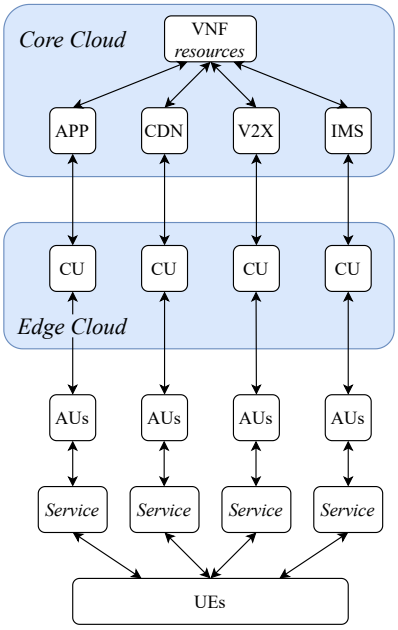

Figura 2.9: Arquitetura de *two-tier network slicing*.

usada uma DQN para alocar recursos rádio para as *edge networks* através da criação de *slices* para aplicações, *Content Distribution Network (CDN)*, *Vehicle-to-Everything (V2X)* e *IP Multimedia Subsystems (IMS)*, de modo a maximizar a QoE e a taxa de utilização de recursos alocados. O espaço de estados é uma combinação de satisfação de requisitos e taxa de utilização de recursos. O espaço de ações consiste em alocar *clusters* RAN, ou seja, grupos de células na mesma área geográfica, para *slices*. A *reward* é dependente do *state-value*, sendo positiva para elevados valores de *state-value*, 0 para valores médios ou negativa para valores baixos. Na *edge cloud* é usado um algoritmo heurístico para otimizar a alocação de recursos nas AUs tendo em conta os requisitos dos UEs. Este estudo conclui que o esquema proposto pode ser mais eficiente do que outros esquemas tradicionais.

O método proposto em [38] também se baseia em dividir a alocação de recursos em dois níveis. No primeiro nível, o *infrastructure provider* lida com a gestão de recursos *inter-slice*, reservando dinamicamente recursos para *slices* em BSs de acordo com os recursos não usados numa BS e a importância dos requisitos de uma *slice* relativamente às restantes. A soma destes recursos reservados para *slices* numa BS, designados e, determina os recursos não usados de uma BS, que podem ser usados para futuras reservas. Parte de e na BS é efetivamente alocada e usada por *slices*, sendo esses recursos alocados designados v. No segundo nível, os recursos alocados a utilizadores de cada *slice* são dinamicamente ajustados através de uma DQN para otimizar a QoE e a taxa de utilização de recursos. O espaço de estados é constituído por tuplos (v, U, R, e), em que U é a satisfação QoE e R é a taxa de utilização de recursos média. O espaço de ações consiste num conjunto de percentagens em que um valor positivo representa um aumento nos recursos da *slice*. A *reward* é definida como a soma ponderada de R e U. Os novos recursos determinados

para a *slice* após uma ação são traduzidos como reservas de recursos em cada BS e o *infrastructure provider* é responsável por atualizar e e v. Neste cenário, a operadora tem conhecimento dos requisitos das *slices* e dos requisitos de cada utilizador numa *slice*. Os resultados mostram que este método melhora a utilização de recursos e satisfação QoE das *slices*.

Em [42] é proposto um esquema de alocação dinâmica de recursos E2E, em que cada *slice* é modelada como múltiplas VNFs em diferentes equipamentos físicos, consideradas como um todo, como na Figura 2.10, e os seus requisitos são dinâmicos ao longo da sua existência. O agente junta uma CNN para melhor capturar as relações entre estados, ações

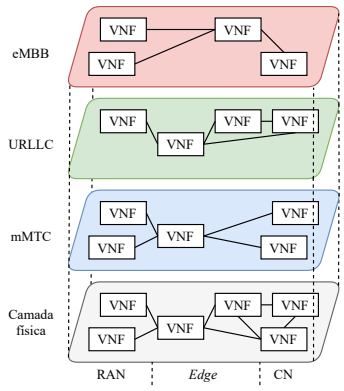

Figura 2.10: Arquitetura de *E2E network slicing*.

e *rewards* com um algoritmo *policy gradient* usado para ajustar os recursos entre *slices* e entre as VNFs de cada *slice* de acordo com os requisitos a cada *time step*. Cada *slice* s_m tem a observação $ob(s_m) = (C_m, U_m, I_m, P_m)$. Sendo C_m um vetor contendo a percentagem de recursos alocados para cada VNF de s_m, U_m um vetor de percentagens de uso dos recursos em cada VNF, I_m a soma ponderada dos recursos alocados por cada VNF nos nós físicos usados por estas, e P_m um vetor com as percentagens de utilização de recursos em cada nó físico sobre o qual o *slice* s_m é executado. Um estado é portanto um vetor, de tamanho fixo, com as obervações de cada *slice* $ob(s_m)$. Uma ação corresponde a alterar os recursos de uma VNF por uma dada percentagem entre -100% e 100%. O *time step* só avança quando os novos recursos de todas as VNFs tiverem sigo agendados pelo agente. A *reward* combina uma *penalty function* que penaliza a violação de SLAs e a utilização ineficiente de recursos na rede física. Os resultados mostram melhorias sobre as técnicas de *slicing* aleatórias, *best-effort* e heurísticas ao melhorar a utilização de recursos e garantir QoS, validando a ideia de *E2E slicing*.

2.4.2 Slices operadas por tenants

Com este modelo, as operadoras criam *slices* e passam o seu controlo aos *tenants*, podendo existir múltiplas *slices* do mesmo tipo de serviço para cada *tenant*. Desta forma, nenhuma das entidades consegue otimizar completamente todas os *slices* porque a operadora não

conhece os requisitos dos utilizadores das *slices* dos *tenants*, e estes não têm conhecimento das restantes *slices* na rede. Assim, o desafio para a operadora é realizar um controlo de admissão avançado para preservar a qualidade de serviço das *slices* existentes quando surgem pedidos por novas, sem violar os SLAs pedidos pelos *tenants*.

O estudo em [5], baseado no de [6], propõe a utilização de DRL para a otimização do lucro da operadora combinado com um modelo analítico para calcular a região de admissibilidade da rede que procura garantir que os SLAs requeridos pelos utilizadores sejam cumpridos. Esta combinação é designada N3AC. A região de admissibilidade pretende definir um máximo número de *slices* que podem ser aceites sem comprometer o cumprimento dos SLAs de qualquer uma. Este trabalho identifica os recursos espetrais como os mais importantes na decisão de aceitar ou rejeitar um pedido, focando-se sobre a RAN. Cada *step* corresponde a um evento: chegada de um pedido de uma *slice* elástica, inelástica, ou fim de uma *slice* em execução. O seu espaço de estados consiste no número de *slices* elásticas e inelásticas atualmente na infraestrutura, juntamente com os requisitos do novo pedido. Na situação de fim de uma *slice*, a ação tomada pelo agente é considerada irrelevante, mas quando em resposta a um pedido, esta pode ser aceitar ou rejeitá-lo. Se aceitar um pedido levar à saída da região de admissibilidade, ou seja, ao consequente incumprimento dos requisitos do pedido, este é automaticamente rejeitado. Quando um pedido é aceite, é recebida uma *reward* correspondente ao produto entre a duração do slice e o seu preço por unidade de tempo. Quando este termina, a *reward* é parcialmente retirada no caso de incumprimento dos SLAs do pedido, ou mantida se os requisitos forem satisfeitos. Este trabalho, em comparação com o de [6], mostra mais uma vez as melhorias obtidas pela utilização de DQN em vez do *Q-learning* básico, e a viabilidade da otimização do lucro de uma operadora para resolver o problema da alocação dinâmica de recursos de rede para *slices*.

Em [24] segue-se o cenário descrito anteriormente em [6], do ponto de vista de *tenant* em vez da operadora, segundo uma abordagem financeira. O objetivo é usar Q-learning para maximizar o lucro do *tenant* enquanto se cumprem os requisitos de QoE num ambiente de troca de recursos com a operadora e os utilizadores, em que cada *tenant* age individualmente. As entidades e relações deste modelo financeiro são mostradas na Figura 2.11, em que o *tenant* desempenha um papel principal nas negociações. Cada estado

Figura 2.11: Entidades e relações no modelo de negócio de *network slicing*.

é representado pelo tuplo $(\varphi, D_{n,t}, \rho, \hat{t})$. φ é o *flow ratio* do *slice*, ou seja, a taxa de *flows* que carregam tráfego inelástico (com requisitos de serviço muito estritos) na *slice*. $D_{n,t}$ é o total de recursos necessários para a *slice* n no *time slot* t, que consiste na soma dos

recursos mínimos de *flows* inelásticos adicionada à soma dos recursos médios necessários em *flows* elásticos. ρ é o rácio entre o preço pago à operadora por uma unidade de largura de banda e o valor taxado a clientes. Finalmente, para evitar que o número de estados cresça para infinito, é aplicado o conceito de *time zone*, $\hat{t}$, agrupando *time slots* próximos em *time zones*. O espaço de ações consiste em aumentar ou diminuir os recursos necessários vindos da operadora por uma quantidade definida em intervalos discretos. A *reward* considera o lucro do *tenant* e penaliza a violação de requisitos QoE. As simulações apresentadas consideram um cenário muito reduzido e não é discutida a escalabilidade do método, nomeadamente a complexidade espacial originada por tabelas de Q-learning.

No trabalho [9] é proposto um sistema de *RAN slicing* em que os *tenants* competem por atribuições de canais rádio da operadora como num leilão. Neste cenário, cada *tenant* tem uma *slice* e não são consideradas *slices* com diferentes requisitos de QoE. O problema é formulado como um *non-cooperative abstract stochastic game*, em que os *tenants* competem em *time slots* por um número limitado de canais rádio de modo a dar aos *mobile users* (MUs) acesso aos seus serviços, agindo independentemente entre si. Esta solução implica a existência de um MDP por *tenant*, simplificando a tomada de decisões e permitindo calcular *action-values* nos MUs de cada *tenant*. Para resolver os problemas do grande espaço de estados e da sobrestimação de *Q-values* é empregue uma *double DQN*. Cada estado é uma combinação da localização/célula, número de pacotes recebidos e o tamanho do *buffer* para novos pacotes do MU. Uma ação corresponde a decisões de alocação de canais, *computational offloading* (transferência de processos computacionais para a *edge cloud*) e agendamento de pacotes. A *reward* corresponde a uma *utility function* que determina a qualidade da ação tomada. Apesar de os resultados serem promissores, não é claro até que ponto é viável calcular *Q-values* nos MUs, devido às suas baixas capacidades de processamento.

O estudo em [41] considera o agendamento de pedidos por *slices* de *tenants* na operadora, que executa controlo de admissão e agrupa vários tipos de recursos para diferentes classes de utilizadores separados em *slices*. São tidos em conta 3 *tenants* de três classes de serviços: automóvel, manufatura e utilidades, cada uma com os seus requisitos, levando a diferentes preços. O problema é visto do ponto de vista da operadora mas as *slices* continuam a ser geridas pelos respetivos *tenants*. Este é formulado como um *Semi-Markov Decision Process (SMDP)*, em que as ações só são tomadas no final de cada época, neste caso, no tempo entre dois pedidos sucessivos por *slices*, e são comparadas soluções utilizando DQN, *double DQN* e *dueling DQN*. Cada estado corresponde a um vetor com uma posição para cada classe de *slices*, marcada a 1 se existir um novo pedido por uma *slice* dessa classe, -1 se receber um indicador de término de uma *slice* ou 0 se não existir qualquer tipo de pedido. O espaço de ações resume-se a um binário, com o valor 1 no caso de aceitar um pedido de *slice* ou 0 se recusar. A *reward* representa a quantia recebida do *tenant* quando recebido e aceitado um pedido por um *slice*, ou é igual a 0 se um pedido for rejeitado ou não forem recebidos pedidos. Os resultados mostram a melhoria de *performance* introduzida pela *dueling DQN* mas são obtidos num cenário muito simplista

com um espaço de estados pequeno e onde os requisitos das *slices* são estáticos durante a sua existência, não sendo muito generalizável para casos reais. Uma melhoria apontada é a de futuramente considerar recursos de ligação e a existência de múltiplos *data centers* ao acomodar mais estados no sistema.

Em [25] é proposto um método de admissão de controlo de *slices* semelhante, mas neste caso considerando a alocação de recursos de largura de banda e *virtual machines* (VMs) por *slice* apenas na CN, com diferentes classes de *slices* existentes e requisitos estáticos durante a existência de cada um destes. São propostos dois modelos de alocação: *service upon arrival* e *batch service*. No primeiro, os pedidos são respondidos imediatamente, portanto o estado é constituído pelos pedidos por recursos recebidos e o espaço disponível nos *buffers* para mais pedidos desde o tempo de chegada do último. No segundo, os pedidos são mantidos em *buffers* e periodicamente processados pelo *slice orchestrator*, logo, o estado é semelhante mas considera-se apenas desde o tempo do último *service*. Para ambos os modelos as ações consistem na alocação de recursos por *slice* e a *reward* combina o *delay* de processamento do pedido e o custo do uso dos recursos. A maior novidade deste trabalho consiste no uso de um método *policy gradient* semelhante ao *REINFORCE* em vez de uma DQN. No entanto, a *reward* não penaliza violações de SLAs, o que impede a avaliação correta do desempenho do algoritmo neste cenário.

2.4.3 Discussão

A análise dos trabalhos relacionados permite concluir que de facto DRL é uma solução viável para o problema da alocação dinâmica de recursos em *slices*. O algoritmo DQN é o favorito na área, mas foram sublinhadas múltiplas melhorias, nomeadamente o uso de *double DQN* de modo a diminuir a sobrestimação de *Q-values* e a arquitetura *dueling* para aumento de *performance* através da divisão de $Q(s_t, a_t)$ em $V(s_t)$ e $A(s_t, a_t)$.

A utilização de DNAFs propõe uma divisão semelhante à *dueling DQN* mas possibilita o uso de um espaço de ações muito maior, o que permite alocações mais precisas e portanto uma maior velocidade de convergência. Uma *Distributional* DQN permite ao agente explorar o ambiente e reduzir o efeito das *noisy rewards* utilizando uma distribuição de *Q-values* em vez de um único *Q-value*, sendo, tal como com DNAFs, particularmente eficaz em situações com grandes espaços de ações. No entanto, este tipo de algoritmos não é útil numa situação em que a decisão é apenas entre aceitar ou rejeitar um pedido.

Outra estratégia promissora consiste na criação de múltiplos agentes que funcionem concorrentemente em vários ambientes enquanto guardam experiências num *experience replay buffer* comum, acelerando substancialmente o treino. Deste modo, a cada *slice* é associado um agente, não sendo necessário treinar novamente o algoritmo em cenários com diferentes números de *slices*. Apesar de DQN ser o algoritmo mais comum, alguns dos estudos apresentados provaram que os métodos *policy gradient* também podem apresentar bons resultados. Seria interessante avaliar o desempenho das combinações de algoritmos de DRL nesta aplicação mas deve-se ter em conta que isso pode levar à sobreposição de

técnicas que baixem o desempenho, e a agentes extremamente complexos com tempos de treino inaceitáveis em *hardware* comum, pelo que deve ser utilizado *hardware* dedicado para este tipo de testes.

Alguns dos estudos anteriores validam ainda a combinação de métodos como CNNs e GANs com DRL. As CNNs dão ao agente a capacidade de compreender relações complexas entre dados próximos no estado, enquanto que as GANs competem de modo a treinar um *generator* capaz de gerar dados relevantes baseados nos reais, como *Q-values* ou distribuições de valores. Estes métodos podem impulsionar substancialmente os resultados obtidos mas, tal como no ponto anterior, deve-se ter em conta a complexidade dos agentes.

Finalmente, quanto aos modelos de *slicing*, estes podem ser geridos por operadoras ou por *tenants*, e o *slicing* pode ser feito na RAN, na rede de transporte, *cloud network*, ou ao longo de todo o *slice* (*end-to-end*). *Slicing* na rede de transporte é ainda um modelo pouco explorado em trabalhos *state-of-the-art*. Esta parte da rede funciona como uma ligação entre a RAN e a *cloud network*, incluindo *Mobile Edge Computing* (MEC) *stations*.

Uma perspetiva provada eficaz é a da otimização do lucro obtido, quer pelos *tenants*, quer pelas operadoras. No entanto, otimização em *slices* geridas pela operadora permitem uma otimização conjunta com a utilização de recursos, aceitando ou rejeitando pedidos de acordo com o estado da rede. Nesta visão procura-se obter um maior lucro e simultaneamente fazer uma melhor distribuição dos recursos enquanto se cumprem os SLAs dos pedidos.

Implementando um modelo semelhante ao descrito anteriormente, combinado com a sua aplicação sobre a rede de transporte, seria possível utilizar um algoritmo que atribuísse caminhos entre pontos na rede de modo a melhorar a utilização da rede, em vez de calcular uma região de admissibilidade, enquanto que um agente DRL decidiria que pedidos aceitar ou rejeitar, tendo em conta os requisitos e o estado da rede.

3

Modelo do Sistema

Partindo da discussão do estado da arte, decidiu-se considerar neste trabalho o cenário da rede de transporte entre a RAN e a CN. O motivo para esta escolha é o facto de os trabalhos anteriormente abordados se focarem maioritariamente na RAN, na CN ou na sua combinação, mas nunca na rede de transporte entre ambas. Para resolver o problema da admissão de slices na rede de transporte, otimizando conjuntamente a utilização da rede e o lucro do operador, aplicou-se o algoritmo DQN e a sua melhoria de utilizar uma arquitetura *dueling*, tendo estes apresentado resultados promissores nalguns dos trabalhos discutidos anteriormente. Neste capítulo é feita uma descrição detalhada do modelo juntamente com a formalização do problema a resolver. Esta descrição inclui o algoritmo e o agente utilizados no trabalho.

O modelo em questão possui duas entidades: os *tenants* e a operadora. Os primeiros criam pedidos representados por um tipo e recursos durante um determinado período de tempo. Os recursos podem incluir determinados SLAs, neste caso um valor de largura de banda, e contém os nós da rede a interligar (*base stations*, MEC *stations* e *computing stations*).

Os pedidos podem ser por *slices* elásticas ou inelásticas. A diferença entre estas está na exigência do cumprimento dos requisitos do pedido: as *slices* elásticas devem manter uma média de largura de banda entre as suas *base stations* e *computing stations* igual ou superior à estabelecida no pedido, enquanto que as inelásticas devem manter um valor absoluto de largura de banda igual ou superior ao do pedido. Os pedidos podem ser formalizados como na Equação 3.1:

$$req = (type, duration, bandwidth, price, connections) \tag{3.1}$$

- *type* pode ser 0, representando o fim de um *slice*, 1 no caso do início de uma *slice* elástica ou 2 para uma *slice* inelástica. É importante referir que no caso de um pedido com *type 0*, todos os restantes campos do pedido devem ser considerados 0.
- *duration* corresponde à duração do *slice* em segundos.
- *bandwidth* é o valor da largura de banda em Mb/s.

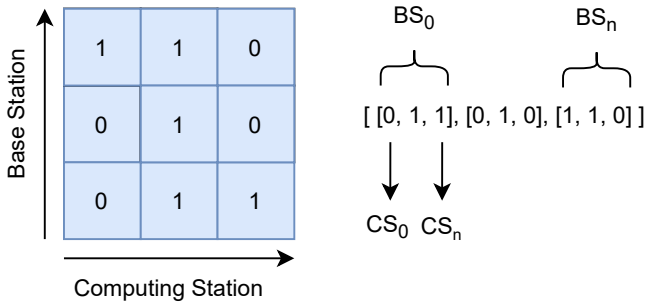

Figura 3.1: Vetor das interligações entre *base stations* e *computing stations*.

- *price* é o preço que o *tenant* pagará à operadora por cada segundo do seu *slice*.
- *connections* é um vetor bidimensional que representa as ligações pedidas entre todas as *base stations* e *computing stations*, sejam elas MEC ou *core*. Cada posição pode ser 0 ou 1, dependendo das ligações que o *tenant* considerar necessárias para atingir a cobertura rádio e poder computacional desejados. A Figura 3.1 exemplifica este vetor bidimensional.

A geração de pedidos de *slices* elásticas e inelásticas é feita segundo distribuições de probabilidade diferentes: o tempo entre a chegada de *slices* elásticas, em segundos, segue uma distribuição de *Poisson* com média λ_e, enquanto que a das *slices* inelásticas utiliza uma média λ_i. Estes parâmetros permitem regular a frequência da chegada de cada tipo de pedidos, bem como a proporção entre *slices* elásticas e inelásticas. Quanto à duração dos pedidos, ambos se baseiam numa distribuição exponencial de escala β_{dur}. A largura de banda de cada pedido pode ser 50, 100, 200 ou 300 *Mb*, sendo esta selecionada aleatoriamente. O número de interligações BS-CS no pedido provém também de uma distribuição exponencial de escala β_{con}. Finalmente, tendo em conta o número gerado, as ligações são escolhidas aleatoriamente.

O objetivo deste modelo é, através de um agente de DRL, melhorar o lucro obtido por uma operadora por responder e servir pedidos de *slices* vindos de múltiplos *tenants*, decidindo quando aceitar ou rejeitar os pedidos com base no lucro e no estado da rede.

3.1 Escolha de caminhos

Durante a duração de uma slice é necessário definir quais os caminhos utilizados para interligar os vários pontos de acesso e computação. De modo a este resolver problema, foi criado um algoritmo capaz de escolher um caminho entre cada par de nós. Este começa por calcular os ρ caminhos mais curtos entre cada par, segundo o número de *hops*. Com uma determinada periodicidade em segundos, o algoritmo mede a largura de banda disponível em cada *link* dos ρ caminhos selecionados. Com estes dados, é possível determinar qual o *link* com menor largura de banda em cada caminho, designado *bottleneck*. O

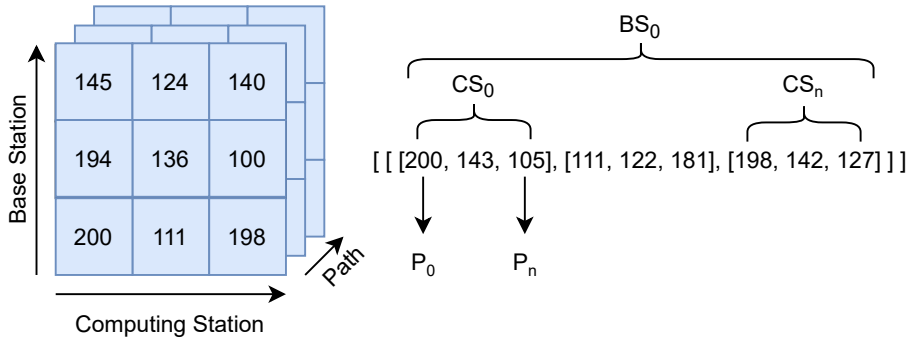

Figura 3.2: Exemplo de parte do vetor com os *bottlenecks* dos caminhos entre pares *base station-computing station*.

caminho selecionado para interligar um par de pontos é aquele, de entre os ρ caminhos, com o menor *bottleneck*, ou seja, maior largura de banda.

A informação dos *bottlenecks* deve ser recolhida por um controlador e pode ser representada através de um vetor tridimensional, em que na primeira dimensão os índices correspondem a *base stations*, os da segunda a *computing stations* de qualquer tipo, e os da terceira são os índices dos ρ caminhos disponíveis. O valor é o *bottleneck* do caminho. O vetor é exemplificado na Figura 3.2:

3.2 Modelação do Problema como um Markov Decision Process

Neste modelo, o agente é responsável por decidir que pedidos aceitar ou rejeitar. Processa um estado $s \in S$, em que S é o espaço de estados, composto pela informação do pedido e pelo estado da rede:

$$s = (req, elastic, inelastic, bottlenecks) \tag{3.2}$$

Nesta expressão, *req* é o pedido descrito na Equação 3.1, *elastic* é o número de *slices* elásticos ativos na rede e *inelastic* o número de *slices* inelásticas. Finalmente, *bottlenecks* é o vetor tridimensional representativo dos *bottlenecks* dos múltiplos caminhos entre nós na rede, descrito na Secção 3.1. Este estado é transformado num vetor unidimensional de modo a ser processado pelo agente.

O espaço de ações $A = 0, 1$ é composto por dois tipos de ação: (0) rejeitar ou (1) aceitar o pedido recebido. Quando o estado representa o final de uma *slice*, a ação tomada é irrelevante porque o pedido de término será sempre aceite.

Por último, a *reward* r é simplesmente $r = price \cdot duration$, ou seja, o produto entre o preço do *slice* por segundo e a duração do mesmo.

Quando um *slice* termina e produz um novo estado s_{t+1} com um pedido do tipo 0, ou seja, fim de uma *slice*, o seu desempenho ao longo da sua duração é avaliado. No caso

das *slices* elásticas, a média da largura de banda medida tem de ser igual ou maior que a requisitada, caso contrário, a *reward* produzida retira metade do preço pago por esta. Quanto às *slices* inelásticas, todas as medidas de largura de banda têm de ser iguais ou superiores à requisitada para que a nova *reward* não desconte o valor total pago pelo *slice*. Quando estes requisitos são cumpridos, a *reward* fica inalterada.

3.3 Agente

O funcionamento do agente começa na obtenção de um primeiro estado, s_t, correspondente ao primeiro pedido. O estado é o *input* da DQN, produzindo como *output* os *Q-values* de cada ação para o estado recebido. A ação de aceitar ou rejeitar o pedido é a com o maior *Q-value* associado. Ao tomar a ação escolhida, é produzido um novo estado s_{t+1} e uma *reward*. O processo é repetido até ser atingido um número máximo de pedidos por época, $maxRequests$. O Algoritmo 2 descreve o *loop* de funcionamento do agente em pseudocódigo.

Algorithm 1 Funcionamento do Agente

```
1:  i ← 0
2:  while i < maxRequests do
3:      done ← False
4:      state ← req
5:      while done ≠ True do
6:          Qvalues ← network(state)   ▷ Passar state pela rede para obter o Q-value de cada ação
7:          action ← maxIndex(Qvalues)   ▷ Ação com maior Q-value
8:          nextState, reward, done ← environment.step(action)
9:          state ← nextState
10:     i ← i + 1
```

O *loop* de treino é igual para ambos os agentes e o seu funcionamento começa na obtenção do primeiro estado s_t, correspondente ao primeiro pedido. A partir deste estado são calculados os *Q-values* para cada ação de rejeitar ou aceitar o pedido. De forma a introduzir exploração do ambiente, com probabilidade ϵ é escolhida uma ação aleatória, e com probabilidade $1-\epsilon$ é selecionada a ação com melhor *Q-value*. Uma vez tomada uma ação a_t, é produzido um novo estado s_{t+1} e uma *reward* r_t. Este grupo de resultados é agrupado numa experiência e o processo repetido até que exista um determinado número de experiências no *experience replay buffer*.

Quando existem experiências suficientes, um conjunto destas é recolhido aleatoriamente. Todos os estados s_t das experiências selecionadas são passados pela *Q-net* principal, obtendo os *Q-values* das ações para cada estado. Os estados seguintes, s_{t+1}, são passados pela *target network*. Estes últimos são utilizados para cálculo da Equação 3.3, em que *done* é um booleano que representa se o estado é o último de uma época:

$$Q(s_t, a_t) = r_t + \gamma * ((1 - done) * \max_{a_{t+1}}(\hat{Q}(s_{t+1}, a_{t+1}))) \tag{3.3}$$

De seguida, a *loss function* é utilizada para calcular o erro entre a previsão dos *Q-values* e os calculados na Equação 3.3. Sobre o resultado desta função é calculado o *gradient descent* em função dos parâmetros e estes são ajustados através de *backpropagation*.

Com uma dada frequência, ou seja, a cada $syncFreq$ episódios, os valores da *Q-net* são copiados para a *target network* de modo a atualizar os seus parâmetros com os da rede principal. Quando uma época termina, a probabilidade ϵ é diminuída por um valor até atingir um mínimo. Este ajuste é feito para que, à medida que conhece melhor o ambiente, o agente se foque em tomar as melhores ações em vez de explorar. O processo descrito é repetido durante um número de épocas $totalEpochs$. O Algoritmo 2 contém o pseudocódigo que descreve o *loop* de treino dos agentes.

Algorithm 2 Algoritmo DQN

```
1: i ← 0
2: while i < totalEpochs do
3:     step ← 1
4:     done ← False
5:     state ← req                                   ▷ req é o primeiro pedido
6:     while done ≠ True do
7:         step ← step + 1
8:         Qvalues ← network(state)   ▷ Passar state pela rede para obter o Q-value de cada ação
9:         if randomFloat(0, 1) < ϵ then
10:            action ← randomInt(0, 2)                ▷ Ação aleatória
11:        else
12:            action ← maxIndex(Qvalues)              ▷ Ação com maior Q-value
13:        nextState, reward, done ← environment.step(action)   ▷ action no ambiente retorna o novo estado, a reward e o indicador de se o estado é o último da época
14:        replayBuffer ← replayBuffer+(state, action, reward, nextState, done)
           ▷ Adicionar experiência ao replay buffer
15:        state ← nextState
16:        if size(replayBuffer) > batchSize then
17:            minibatch ← randomSelect(replayBuffer, batchSize)   ▷ Selecionam-se aleatoriamente batchSize experiências
18:            stateBatch ← minibatch.states
19:            actionBatch ← minibatch.actions
20:            rewardBatch ← minibatch.rewards
21:            nextStateBatch ← minibatch.nextStates
22:            doneBatch ← minibatch.dones
23:            X ← network(stateBatch)                 ▷ Passar todos os estados pela rede
24:            Qvals ← targetNetwork(nextStateBatch)   ▷ Passar todos os estados seguintes pela target network
25:            Y ← rewardBatch + γ ∗ ((1 − done) ∗ max(Qvals))
26:            loss ← lossFunction(X, Y)
27:            loss.backpropagate()
28:            if step%syncFreq == 0 then
29:                targetNetwork.parameters ← network.parameters
30:        if ϵ > minLimit then
31:            ϵ ← ϵ − 1/totalEpochs
32:    i ← i + 1
```

4

Ambiente

Neste capítulo são apresentadas as tecnologias utilizadas para criar o ambiente de simulação da rede de transporte utilizando o *Containernet* [33] e o controlador SDN *Ryu* aplicados sobre o *OpenAI Gym*. São também apresentadas as implementações do algoritmo de seleção de caminhos e do agente que recebe pedidos por *slices*.

4.1 Mininet e Containernet

De modo a simular esta topologia escolheu-se o *Mininet* [15]. Foi escolhido este simulador devido a utilizar elementos de rede virtuais em *software* com um funcionamento semelhante ao *hardware* físico e facilidade de configuração e de integração com *software-defined networks* [13]. Estas vantagens provém do facto deste ser executado sobre o *Linux kernel* e a sua pilha de protocolos de rede, utilizando o *switch* virtual *Open vSwitch (OVS)* que é imensamente utilizado em plataformas de virtualização de rede. Estas caraterísticas permitem simular o comportamento real de topologias, sendo o código de controlo desenvolvido facilmente portado para *hardware* físico com alterações mínimas.

O *Mininet* é um simulador de redes que permite a criação de *hosts*, *switches*, *links* e controladores virtuais. Através da sua extensa API em *Python*, oferece a oportunidade de criar topologias personalizadas, configurando os equipamentos virtuais e definindo a largura de banda, *delay* e percentagem de perda de pacotes nos *links*. Os seus *switches* OVS suportam a criação de SDNs através dos protocolos *OpenFlow* e *P4*.

Containernet [33] é uma bifurcação do repositório *Mininet* que surgiu com o objetivo de simplificar o desenvolvimento e teste de VNFs (*Virtual Network Functions*), substituindo os *hosts* por *Docket containers* no *Mininet* através de uma API em *Python*. Com esta funcionalidade é possível simular serviços de computação nas MECs e CSs, como servidores *web*, bases de dados, entre outros. O *Containernet* permite ainda emular a utilização de recursos CPU, RAM e memória, bem como partilhar volumes de armazenamento entre o simulador local e os *containers* através de *Docker volumes*, o que possibilita a extração e análise de resultados produzidos nos *containers*. Este conceito é ilustrado na Figura 4.1.

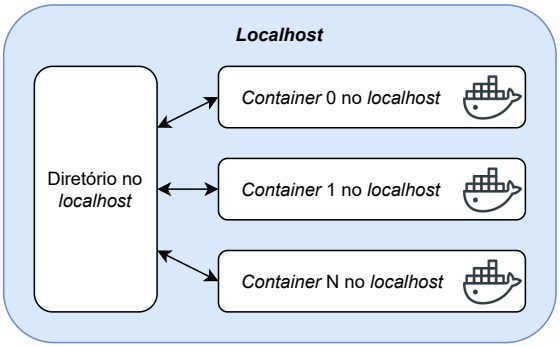

Figura 4.1: Esquemático do funcionamento de *Docker Volumes*.

No contexto deste trabalho, o *Containernet* foi utilizado para montar a topologia representada num ficheiro de texto, constituída pelas BSs, MECSs e CSs. Cada um destes elementos corresponde a um *Docker container* com o seu próprio espaço de armazenamento onde regista as estatísticas da *performance* de cada uma dos *slices* que o utilizarem. Este armazenamento é acessível ao simulador utilizando os *Docker volumes*, o que permite comparar os resultados das *slices* com os pedidos dos *tenants*. Para além dos *hosts*, recorre-se ao *Containernet* para criar os *switches Open vSwitch*, os *links* e preparar-se para uma ligação de um controlador SDN.

4.2 Slices

O tráfego produzido por cada *slice* criada no sistema é simulado pelo *software Iperf*. O *Iperf* [20] é uma ferramenta de geração e medição de tráfego para redes. Nesta dissertação, é utilizado com o propósito de criar ligações UDP, no caso de *slices* inelásticas, ou TCP, no caso de *slices* elásticas, entre pares de *hosts*, que funcionam como cliente e servidor. O *Iperf* foi utilizado neste contexto para injetar tráfego com uma determinada largura de banda, durante um período de tempo específico, entre um par de *hosts*.

Os resultados de cada *Iperf* são guardados em *log files* nos *containers* clientes e servidores da *slice* e utilizam o formato JSON (*JavaScript Object Notation*) para representar a largura de banda disponibilizada em cada segundo da *slice*. Desta forma, é possível avaliá-lo e decidir se a largura de banda foi suficiente para responder aos requisitos dos pedidos e ajustar a *reward* produzida para o agente de acordo com o resultado.

4.3 Controlador Ryu

O controlador é um bloco de *software* com uma visão global da rede, centralizando-se a tomada de decisões neste elemento enquanto que os *switches* tratam do encaminhamento dos pacotes até ao seu destino, de acordo com as instruções do controlador [31].

De modo a estabelecer uma ligação entre o controlador SDN e os *switches* OVS da topologia recorre-se ao protocolo *OpenFlow*. Este permite definir, para cada *switch*, o que

fazer com um pacote através de *matching* deste com parâmetros do pacote como endereço IP, MAC, porto, etc. No caso de *match*, a ação no *switch* pode ser enviar o pacote para outro porto, para o controlador, *flood* ou ignorá-lo (*drop*).

Neste trabalho é utilizado o controlador SDN *Ryu* [37]. Este é *open-source* e disponibiliza uma API em *Python* que facilita a criação de aplicações SDN através do protocolo *OpenFlow*, sendo facilmente integrável com a topologia simulada no *Containernet*.

A aplicação SDN desenvolvida começa com a leitura do ficheiro de texto da topologia para estabelecer proativamente os flows iniciais entre todos os hosts disponíveis. Este processo guarda no controlador o nome de cada *host*, os seus endereços MAC e IP, e o par *switch-port* a que está ligado. Guarda ainda um mapa de adjacências, indicando, para cada *switch*, que porta utilizar para aceder aos seus vizinhos, bem como a largura de banda inicial de cada *link* da topologia.

Reunida a informação necessária sobre a topologia, utiliza-se o *NetworkX* [21], pacote *Python* para análise de estruturas em grafo, para calcular os caminhos mais curtos (com menor número de *hops*) entre cada BS e todas as MECSs e CSs. Cada caminho é representados da seguinte forma:

$$path = [(switch_1, in-port_1, out-port_1), ..., (switch_n, in-port_n, out-port_n)] \tag{4.1}$$

Para cada um destes caminhos é calculado o seu *bottleneck*, ou seja, o *link* do caminho com menor largura de banda, e é selecionado o caminho com o maior *bottleneck* entre cada par de *hosts*. Estes são instalados nos *switches* que compõem os caminhos, criando um *match* com a porta de entrada, MAC origem e destino dos pacotes, e uma ação de *output* para a porta de saída correspondente.

Uma vez estabelecida a comunicação entre os *hosts*, o controlador inicia uma *thread* responsável pela medição da largura de banda de todos os *links* da topologia, enviando periodicamente, neste caso, de 5 em 5 segundos, um pedido a cada *switch* pelo estado das suas portas. Quando a resposta de cada *switch* é recebida no controlador, desperta um evento *OpenFlow PortStatsReply*, e é ajustado o valor de largura de banda de cada *link* guardado no controlador para corresponder à nova medição.

Após terem sido recebidas todas as respostas e feitos os respetivos ajustes, são recalculados os *bottlenecks* dos caminhos e enviados para o *ambiente OpenAIGym* através de um *socket*, para que possam ser selecionados os melhores caminhos entre *hosts* de acordo com o necessário para responder aos pedidos de *slices*. Esta decisão é passada para o ambiente porque é este que gere as *slices*, o que lhe permite escolher os melhores caminhos no começo de cada *slice*. Cada mensagem enviada através deste *socket* tem um total de elementos correspondente à Equação 4.2:

$$length = BSs \cdot (MECSs + CSs) \cdot Paths \tag{4.2}$$

O controlador possui uma outra *thread* em que espera pela ligação do ambiente ao seu *server socket* para assim receber a informação sobre que caminho foi selecionado entre

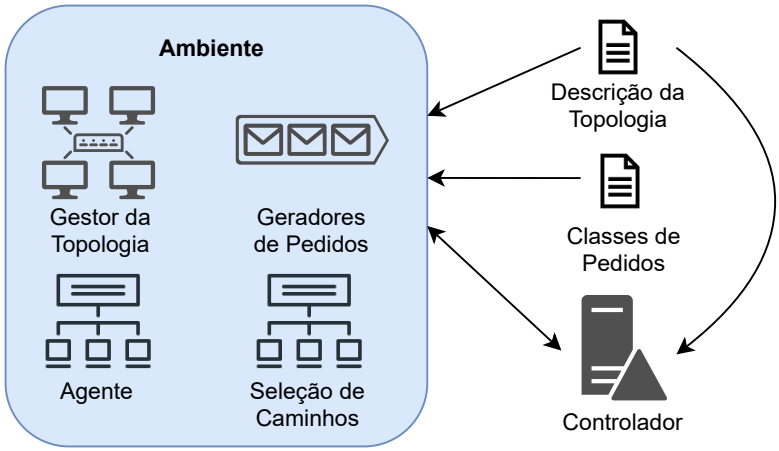

Figura 4.2: Componentes da simulação e respetivas ligações.

cada par *BS-MECS* ou *BS-CS*. O caminho é representado por um inteiro, neste caso de 0 a 6, que corresponde ao índice do caminho a utilizar. Se o índice do caminho recebido for diferente do atualmente em uso, o antigo é removido e o novo instalado através de *flows* nos *switches*. Quando um caminho deixa de ser utilizado, o seu índice tem o valor -1 e é mantidoo o caminho atual entre os seus *hosts*. Cada mensagem recebida através deste *socket* tem um número de índices igual ao produto entre o número de *base stations* e *computing stations*.

4.4 Ambiente OpenAI Gym

O ambiente inclui o *Containernet* que simula a topologia e comunica com o controlador SDN *Ryu*. No entanto, é necessária uma camada que estabeleça a ligação entre estes elementos e o agente DRL. Para isso, criou-se um ambiente baseado no *OpenAI Gym*, um *toolkit* com uma API em *Python* que permite a criação de ambientes através da definição de um espaço de estados, um espaço de ações, um método *reset* que produz o estado inicial de cada época e de um método *step* que aceita uma ação, produz um novo estado e uma *reward*. Um esquema do ambiente de simulação com as suas múltiplas componentes é apresentado na Figura 4.2.

O ambiente começa por carregar os *templates* de pedidos disponíveis a partir de um ficheiro de texto. Estes são os pedidos para os *tenants* enviarem ao agente e representam o tipo de pedido (*e* para elásticos ou *i* para inelásticos), a largura de banda em *Mbps* e o preço por segundo. Os valores de largura de banda escolhidos devem ser variados e suficientes para criar competição pelas *slices*, ou seja, não deve ser possível para o agente aceitar todos os pedidos e receber a melhor *reward* possível, caso contrário não aprenderá em que situações rejeitar pedidos. O preço por segundo pode ser qualquer unidade e a sua principal função é ajustar a preferência por responder a *slices* inelásticas em vez de elásticas.

De seguida, o ambiente inicia uma *thread* com um *server socket* através do qual recebe os *bottlenecks* dos caminhos enviados pelo controlador. No sentido inverso é ligado um *socket* ao controlador para lhe enviar o índice do caminho selecionado entre cada par de *hosts* ou o valor -1 se o caminho não estiver em uso por nenhuma *slice*.

No ambiente em questão, cada época consiste num número máximo de pedidos gerados para o agente. Para simular esta geração de pedidos, no método *reset* de cada época são iniciadas duas *threads*, uma para cada tipo de pedidos, que funcionam até ser atingido o número máximo de pedidos, e duas *Queues*: uma para pedidos de inicialização de *slices* e outra para pedidos de término. O estado devolvido por este método resulta do primeiro pedido gerado pelas *threads* e adicionado à *queue*.

A forma como cada *thread* gera um pedido começa pela amostragem de um valor de uma distribuição de *Poisson*, de média 4 para elásticos e 8 para inelásticos. Estes valores foram escolhidos para que o número de pedidos elásticos seja aproximadamente o dobro dos inelásticos e existam pedidos suficientes para criar competição por *slices* na rede. Cada *thread* fica em espera durante o valor de tempo obtido, criando de seguida o pedido como descrito na Secção 3. Este pedido tem a sua duração baseada numa distribuição exponencial de escala 15, limitada por um mínimo de 1 e máximo de 60 segundos. A largura de banda e o preço são retirados do *template* selecionado aleatoriamente. Finalmente, é amostrado um número de ligações BS-CS a partir de uma distribuição exponencial de escala 2, dando maior probabilidade de um menor número de ligações, com um mínimo de 1 e, neste caso, máximo de 7 ligações por *slice*.

Com o primeiro estado produzido pelo método *reset*, o agente pode tomar a ação de aceitar ou rejeitar o pedido de início da *slice*. Se for aceite, a *reward* é o produto entre a duração da *slice* e o seu preço por segundo. São calculados os melhores caminhos entre cada par BS-CS do pedido e enviados os índices correspondentes para o controlador. Uma vez estabelecidos os caminhos, iniciam-se os *iperfs* entre os pares e uma *thread* encarregue da avaliação da *performance* da *slice*.

As *threads* de avaliação de cada *slice* começam por aguardar a duração estipulada no pedido. De seguida, são enviados os novos caminhos em uso para o controlador, considerando que as ligações entre alguns pares podem já não ser necessárias. Finalmente é feita a análise dos *log files* criados pelos *hosts* da respetiva *slice*. Numa *slice* elástica, se a média da largura de banda medida de segundo em segundo em todas as ligações requisitadas for inferior à pedida, é produzida uma *reward* igual a $-\frac{reward}{2}$. Por outro lado, numa *slice* inelástica, se o valor da largura de banda medida em todas as ligações for inferior à pedida, a *reward* resultante é $-reward$. No caso de os requisitos de largura de banda do *slice* serem cumpridos, a *reward* corresponde a 0. Esta *reward* é adicionada à *queue* dos pedidos de término e é adicionado um pedido do tipo 0 à *queue* de pedidos.

Após a tomada de decisão sobre o pedido, o ambiente gera um novo estado a partir do pedido na primeira posição da *queue*. Se esta estiver vazia é bloqueada até que uma das *threads* crie um novo pedido. No caso de o pedido obtido ser de término de uma *slice*, é consultada a *queue* de fim de pedidos e obtida a *reward* dependente do seu sucesso.

```
q_net = torch.nn.Sequential(
    torch.nn.Linear(INPUT_DIM, HL1),
    torch.nn.ReLU(),
    torch.nn.Linear(HL1, HL2),
    torch.nn.ReLU(),
    torch.nn.Linear(HL2, OUTPUT_DIM)
)
```

Figura 4.3: Código para declarar e inicializar uma DQN.

Quando é atingido o número máximo de pedidos de *slices* (não contando os de término), são paradas as *threads* geradoras de pedidos e espera-se que todas as *threads* avaliadoras terminem, resultando cada uma num novo estado de término de *slice*.

4.5 Agente

Para a sua dos agentes recorreu-se à biblioteca de *deep learning PyTorch* [32]. Esta suporta diferenciação automática e otimização sobre a forma de uma das mais rápidas implementações entre as bibliotecas deste tipo, principalmente devido à sua lógica em C++. A sua unidade fundamental é o tensor, muito semelhante ao *array* multidimensional do *NumPy*. Os tensores *PyTorch* têm um mecanismo de cálculo de gradientes e posterior otimização para o cálculo da *loss* no processo de aprendizagem.

O *PyTorch* possui ainda uma *package* que permite a declaração de camadas de neurónios, funções de ativação e a posterior combinação sequencial destas. Através desta funcionalidade criaram-se duas redes neuronais: uma para a DQN e outra para a *dueling* DQN. A Figura 4.3 apresenta o código *PyTorch* para criar a DQN. Utiliza-se a função *Linear* para declarar uma camada *feed-forward* que processa um *input* de dimensão x e produz um *output* de dimensão y. A função *ReLU* representa a função de ativação *Rectified Linear Unit (ReLU)* e a função *Sequential* cria a sequência de camadas e funções de ativação que constituem a rede.

Para criar a rede *dueling DQN* é necessário criar uma classe que herde as propriedades do módulo NN do *PyTorch* para se ter maior controlo sobre a sequência dos dados, permitindo a divisão em valor do estado e *advantage*, posteriormente somados para obter o *Q-value*. A Figura 4.4 apresenta o código necessário para criar a arquitetura.

Ao tomar uma ação no ambiente, transita-se para o novo estado e guarda-se a experiência no *replay buffer*. Caso este já tenha experiências suficientes, são selecionadas aleatoriamente um número *batchSize* destas e é calculado o erro entre as previsões e os valores teóricos, fazendo *backpropagation* do erro para ajustar devidamente os parâmetros da rede. A Figura 4.5 contém o código utilizado para selecionar as experiências, passá-las pelas redes para obter as previsões e os *Q-values* teóricos, seguindo-se o cálculo do erro entre estes e a sua *backpropagation*.

```
class DuelingDQN(nn.Module):
    def __init__(self):
        super(DuelingDQN, self).__init__()

        self.fc1 = nn.Linear(INPUT_DIM, HL1)
        self.relu = nn.ReLU()
        self.fc_value = nn.Linear(HL1, HL2)
        self.fc_adv = nn.Linear(HL1, HL2)

        self.value = nn.Linear(HL2, 1)
        self.adv = nn.Linear(HL2, 2)

    def forward(self, state):
        y = self.relu(self.fc1(state))
        value = self.relu(self.fc_value(y))
        adv = self.relu(self.fc_adv(y))

        value = self.value(value)
        adv = self.adv(adv)

        avg_adv = torch.mean(adv, dim=1, keepdim=True)
        Q = value + adv - avg_adv

        return Q
```

Figura 4.4: Código para declarar e inicializar uma *dueling* DQN.

```python
if len(replay) > BATCH_SIZE:
    minibatch = random.sample(replay, BATCH_SIZE)
    state_batch = torch.cat([s1 for (s1, a, r, s2, d) in minibatch])
    action_batch = torch.Tensor([a for (s1, a, r, s2, d) in minibatch])
    reward_batch = torch.Tensor([r for (s1, a, r, s2, d) in minibatch])
    next_state_batch = torch.cat([s2 for (s1, a, r, s2, d) in minibatch])
    done_batch = torch.Tensor([d for (s1, a, r, s2, d) in minibatch])
    Q1 = q_net(state_batch)
    with torch.no_grad():
        Q2 = target_net(next_state_batch)

    Y = reward_batch + GAMMA * ((1 - done_batch) * torch.max(Q2, dim=1)[0])
    X = Q1.gather(dim=1, index=action_batch.long().unsqueeze(dim=1)).squeeze()
    loss = loss_fn(X, Y.detach())
    optimizer.zero_grad()
    loss.backward()
    losses.append(loss.item())
    optimizer.step()

    if step % SYNC_FREQ == 0:
        target_net.load_state_dict(q_net.state_dict())
```

Figura 4.5: Seleção de experiências e *backpropagation* no agente.

5

Resultados

Este capítulo apresenta os parâmetros utilizados durante o treino e teste dos agentes, incluindo a topologia, as várias classes de pedidos e as constantes utilizadas no ambiente e agentes. Por último, são introduzidos e discutidos os resultados obtidos.

5.1 Configurações

Esta secção inclui as configurações utilizadas no ambiente durante as etapas de treino e teste de ambos os agentes (DQN e *dueling* DQN). Começa por apresentar a topologia utilizada, as classes de pedidos existentes e os parâmetros das redes neuronais dos agentes.

5.1.1 Topologia

O primeiro passo para a simulação da rede é a escolha de uma topologia. Este é um passo importante que influencia o tamanho do estado consumido pelo agente durante o processo de aprendizagem.

A rede de transporte que se pretende simular estabelece ligação entre as partes rádio, *core* e *mobile edge* da rede. Estas três entidades são representadas por BSs (*base stations*), CSs (*Computing Stations*) e MECSs (*Multi-access Edge Computing Stations*), respetivamente.

A quantidade de BSs é decidida de acordo com os objetivos de cobertura rádio da operadora, mas neste trabalho não são tidas em conta considerações geográficas ou a simulação da RAN, apenas se considera uma lista das BSs necessários nos pedidos enviados pelos *tenants*. As MECs são utilizadas para aproximar poder de computação e armazenamento da camada de acesso da rede, efetivamente reduzindo o congestionamento desta e a latência para os utilizadores por se encontrarem mais perto dos recursos. Neste caso decidiu-se considerar uma MEC por cada BS, possibilitando o suporte das suas operações mais simples. Finalmente, as CSs são as *core computing stations*, necessárias para computações mais exigentes. Apesar de existir uma MECS e CS por cada BS, as BSs podem ligar-se aos nós que necessitarem durante a execução de um pedido.

Outra consideração importante na rede de transporte é uma elevada redundância nos caminhos entre os nós, que permite *load balancing*, aumentando a estabilidade da rede

em situações de falha ou congestão. Esta questão implica a utilização de um número considerável de *switches* na rede, bem como a utilização de *links* mais poderosos em zonas da rede onde se concentra mais tráfego, nomeadamente junto ao *core*.

A topologia resultante, representada na Figura 5.1 considera os pontos anteriores utilizando 7 BSs, 7 MECs próximas das BSs e 7 CSs no *core* da rede. Para interligar estes *hosts* empregam-se 65 *switches*. Todos os *links* na topologia são bidirecionais, mas têm capacidades diferentes:

- entre *hosts* (BS, MEC ou CS) e *switches* têm uma largura de banda de 1 Gbps para garantir que na simulação todo o tráfego dos *slices* é enviado e recebido pelos *hosts*.
- na parte de acesso da rede os *links* têm uma largura de banda de 300 Mbps.
- na camada de distribuição os *links* funcionam com uma largura de banda de 500 Mbps.
- no *core* da rede os *links* possuem uma largura de banda de 700 Mbps.

A topologia é guardada num ficheiro de texto para ser carregada para o ambiente. Neste são representadas as ligações entre os nós da rede e as suas características de largura de banda em *Mbps*, *delay* em *ms* e perda de pacotes em percentagem. Como os *links* são bidirecionais, não se deve repetir a ligação entre dois nós nas duas direções. O ficheiro segue o formato da Tabela 5.1.1.

nome1	nome2	bandwidth	delay	loss
BS1	S1	1000	1	0
S1	S2	1000	1	0

5.1.2 Classes de Pedidos

As classes de pedidos disponíveis são representados num ficheiro de texto para serem carregadas para o ambiente. Decidiu-se que as *slices* inelásticas são 4 vezes mais caras que as elásticas de modo a promover a preferência pelas primeiras. As classes utilizadas são apresentadas na Tabela 5.1.2.

Tipo	Largura de Banda	Preço
1	50	5
1	100	10
1	200	20
1	300	30
2	50	20
2	100	40
2	200	80
2	300	120

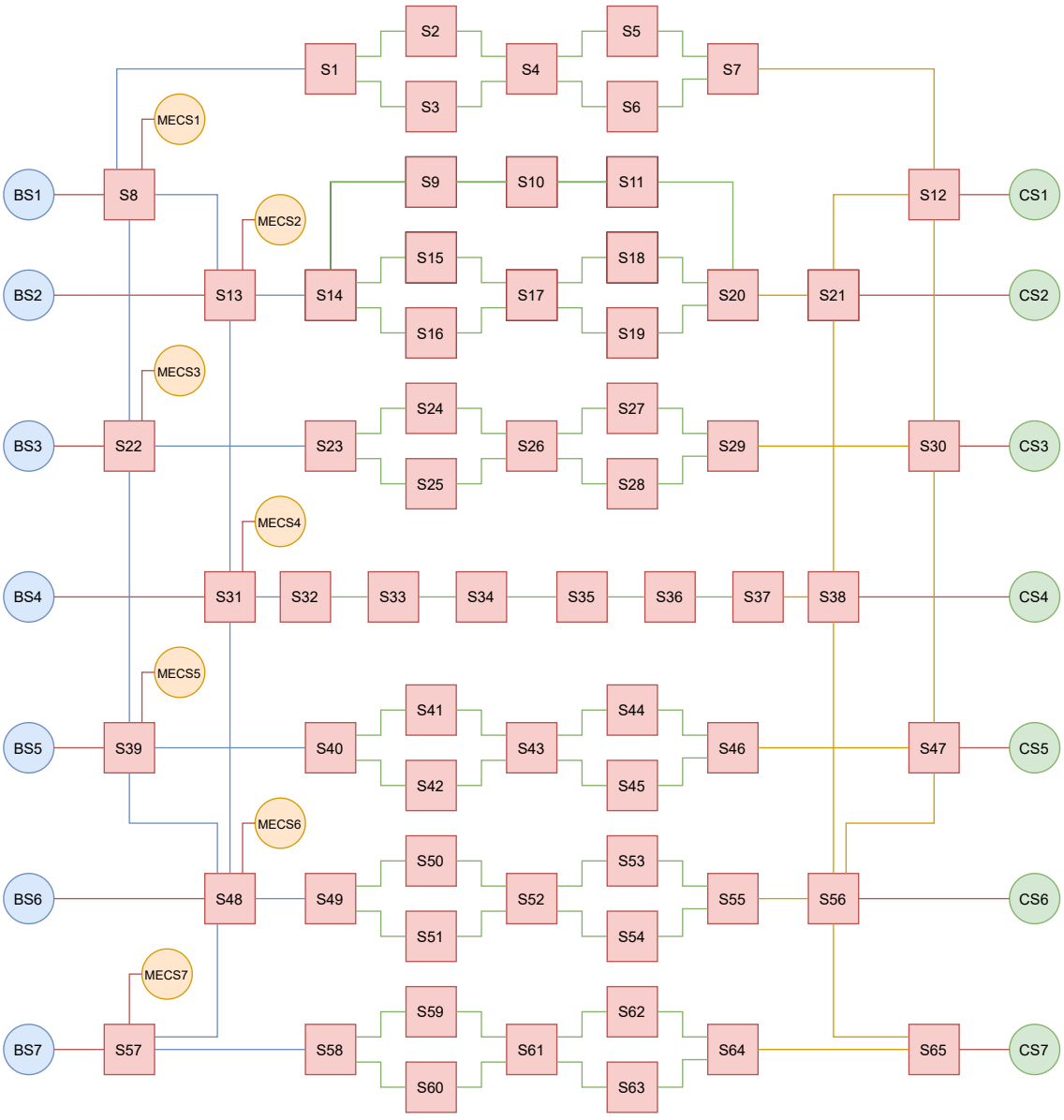

Figura 5.1: Topologia da rede de transporte. Os *links* azuis são de 300 Mbps, os verdes de 500 Mbps, os amarelos de 700 Mbps e os vermelhos de 1000 Mbps.

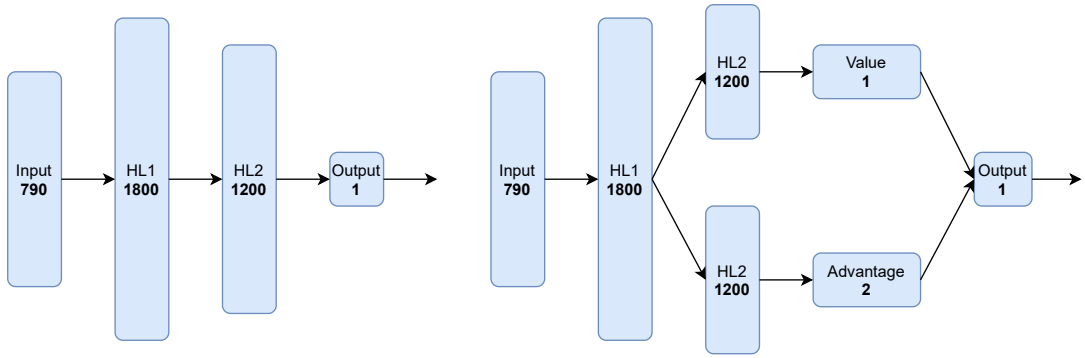

Figura 5.2: Arquitetura DQN simples à esquerda e *dueling* DQN à direita.

5.1.3 Agente

O número máximo de pedidos por época é igual a 50 e cada agente treinou durante 10000 épocas. Tendo em conta o estado descrito na Secção 3.2 e que se estão a utilizar 7 *base stations*, 14 *computing stations* e 7 caminhos entre cada *base station* e *computing station*, o espaço de estados tem um total de 790 índices.

Em ambos os casos as redes têm uma camada de *input* de dimensão 790, correspondente ao estado produzido pelo ambiente, e *output* de dimensão 1 que representa aceitar ou rejeitar um pedido. No caso da DQN, existem duas *hidden layers* com 1800 e 1200 neurónios. A *dueling DQN* tem uma *hidden layer* de 1800, seguida de uma ramificação para duas camadas de 1200 neurónios cada: uma para cálculo do *value* do estado e outra para a advantage de cada ação possível no estado. As duas arquiteturas podem ser vistas na Figura 5.2. A escolha do número e dimensão das *hidden layers* é um problema sem uma resposta universal [35], sendo os parâmetros em uso baseados nos trabalhos estado da arte discutidos. Entre cada camada é aplicada a função de ativação ReLU, a mais comum em *deep learning*.

O *experience replay buffer* é uma *queue* com 1000 posições e o *batch size* é definido como 200, o que significa que a *backpropagation* só é iniciada a partir do momento em que existem 200 experiências no *replay buffer*.

Para calcular o *Q-value* de cada par estado-ação é utilizado um parâmetro γ igual a 0.9 de modo a ter em consideração as *rewards* futuras mas não até ao infinito. Quanto à *loss function*, neste caso utilizou-se a *MSE loss* por ser a mais comum em artigos relacionados. O resultado da *loss function* é utilizado para calcular os gradientes da rede inteira, e estes são aplicados pelo *optimizer Adam*, disponível no *PyTorch* utilizando um *learning rate* de 0.001.

A *target Q-network* é uma cópia da *Q-network* descrita e sincroniza os valores dos seus parâmetros com a original a cada 500 *steps*. Para permitir exploração do ambiente é utilizado um *decaying epsilon*, inicialmente igual a 0.3 até ao mínimo de 0.1, atingido ao fim de 1000 épocas.

5.2 Resultados

Esta secção contém os resultados da aplicação de uma DQN e uma *dueling* DQN divididos em treino e teste, terminando com uma discussão destes.

5.2.1 Treino

Nesta secção são mostrados os resultados do treino das soluções propostas para o problema da admissão de novas slices na rede de transporte entre RAN e CN, otimizando conjuntamente a utilização da rede e o lucro do operador. As métricas em estudo são a *reward* produzida, os pedidos de *slices* elásticas e inelásticas aceites e satisfeitos ao longo do treino.

Em primeiro lugar, para o agente que implementa uma DQN, são relacionadas as *rewards* com os pedidos de *slices* aceites e satisfeitas. Estes resultados podem ser observados na Figura 5.3. No lado esquerdo desta imagem estão representadas as *rewards* médias em grupos de 500 épocas obtidas durante o seu treino. Do lado direito encontram-se as médias dos pedidos aceites e satisfeitos, em grupos de 500 épocas.

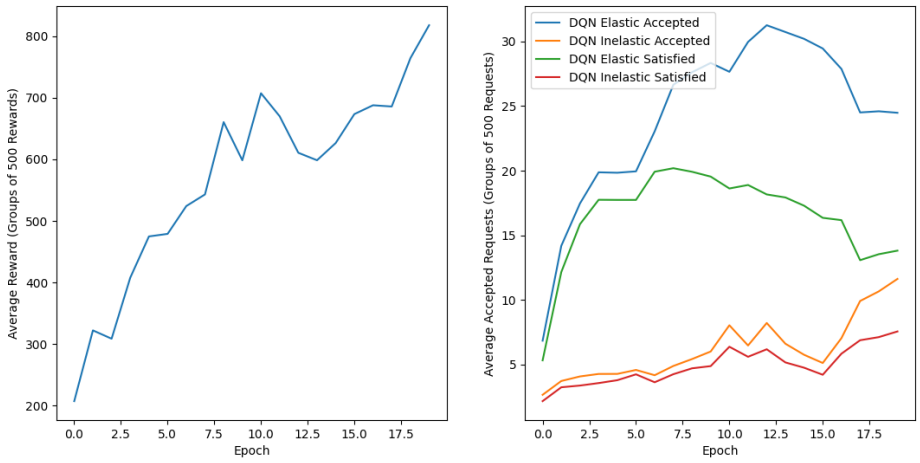

Figura 5.3: Algoritmo com DQN: *Rewards* médias à esquerda, pedidos de *slices* elásticas, inelásticas aceites e satisfeitas à direita.

Inicialmente, com a rede inicializada com parâmetros aleatórios, os pedidos são aceites ou rejeitados sem qualquer critério, resultando numa baixa *reward* e baixa utilização dos recursos da rede, o que permite a satisfação da maioria dos pedidos. À medida que o treino avança é de esperar, e comprovado pelos resultados, que o agente aprenda a aceitar progressivamente mais pedidos. Uma razão pela qual o agente dá inicialmente preferência aos pedidos de *slices* elásticas em vez de inelásticas, apesar de estas produzirem maiores *rewards*, é o facto de as inelásticas gerarem *rewards* negativas com maior frequência devido à exigência com que os seus pedidos são avaliados.

Eventualmente o agente aprende a aceitar a maioria dos pedidos porque o valor absoluto da *reward* de os aceitar é superior à de não os satisfazer. No entanto, aceitar todos os pedidos de *slices* leva à saturação dos *links* da rede, resultando na satisfação de muito poucos pedidos e forçando o agente a compreender em que situações os rejeitar de modo a aumentar a *reward*. Este cenário é observável por volta da época 5000 onde existe uma queda nas *rewards* e pedidos satisfeitos em cada época. Tendo em conta que os resultados apresentados são médias em grupos de 500 épocas, a época 5000 é marcada pelo valor 10 no eixo das épocas.

Por volta das 6000 épocas é possível observar que o número de pedidos aceites é reduzido, fazendo com que a *reward* volte a aumentar porque os caminhos na rede voltam a ter capacidade para os satisfazer. Ao reduzir o número de pedidos de *slices* elásticas aceites, o agente liberta algum espaço na rede para *slices* inelásticas, resultantes numa maior *reward* se os seus requisitos forem cumpridos. Segundo os resultados, esta corresponde à última fase deste processo de treino, a partir das 8500 épocas: o número de *slices* elásticas aceites é reduzido e o de inelásticas aumentado, provocando uma subida nas *rewards* e uma melhoria na satisfação de pedidos inelásticos.

De seguida, é feita uma análise semelhante mas desta vez sobre os resultados produzidos durante o treino do agente que implementa uma *dueling* DQN. Os seus resultados são apresentados na Figura 5.4. Tal como na figura 5.3, o lado esquerdo contém as *rewards* médias do agente ao longo das épocas em grupos de 500, enquanto que o lado direito apresenta as *slices* elásticas e inelásticas médias admitidas, e os pedidos satisfeitos médios.

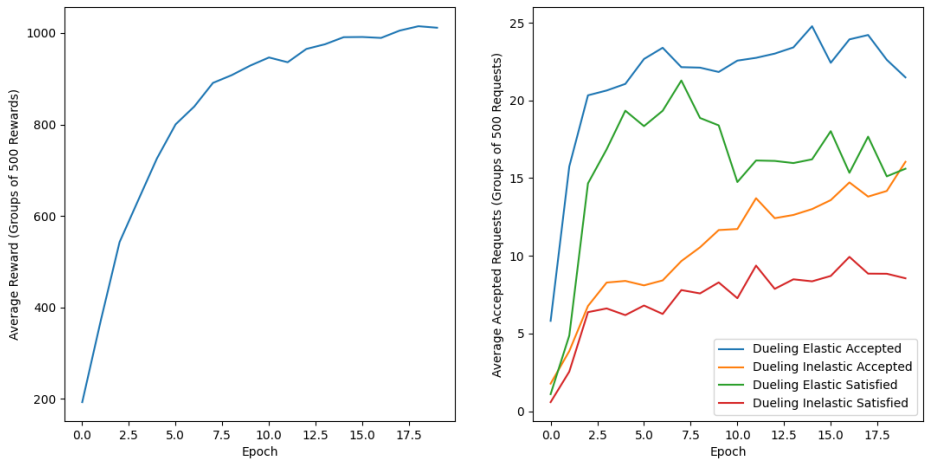

Figura 5.4: Algoritmo com *dueling* DQN: *Rewards* médias à esquerda, pedidos aceites de *slices* elásticas, inelásticas e satisfeitas à direita.

Da mesma forma que na arquitetura anterior, inicialmente com os parâmetros aleatórios, o agente aceita apenas uma pequena parte dos pedidos. Esse valor rapidamente

aumenta à medida que compreende que quanto maior o número de pedidos aceites, maior a *reward*. No entanto, o agente com a *dueling* DQN parece aprender mais rapidamente que saturar a rede não é uma boa política, aceitando um número estável de pedidos de *slices* elásticas e aumentando lentamente o número de inelásticas aceites. Perto do final do treino é possível observar uma diminuição na inclinação dos pedidos satisfeitos, indicadora da proximidade da zona de saturação da rede.

Uma das principais diferenças entre o treino das duas arquiteturas é evidenciada na Figura 5.5. Corresponde ao facto de os resultados do treino com a arquitetura *dueling* não terem momentos em que a *reward* diminui devido à saturação da rede. Este facto também

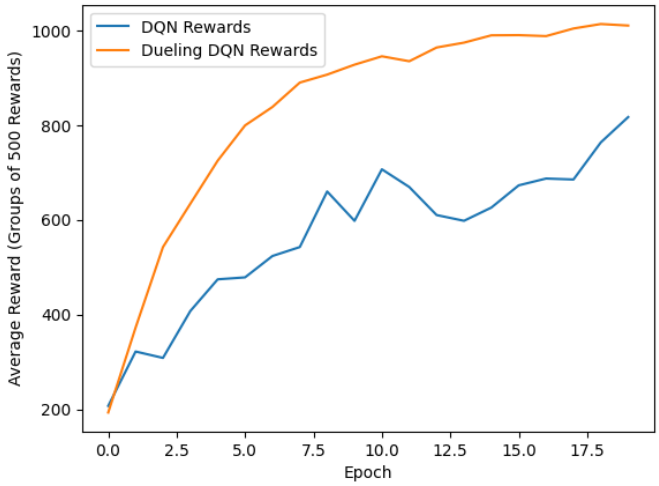

Figura 5.5: *Reward* DQN vs. *dueling* DQN.

é comprovado pela comparação da satisfação dos pedidos na Figura 5.6: no treino da DQN existe um momento, por volta das 5000 épocas, em que os pedidos satisfeitos reduzem devido à saturação dos *links* da rede, que não é tão intensa no treino da *dueling* DQN. O treino desta tem um progresso mais estável, sem exagerar no número de pedidos de *slices* elásticas aceites e dando desde mais cedo maior preferência às inelásticas.

5.2.2 Teste

Esta secção contém os resultados dos testes dos algoritmos. O teste é feito durante 500 épocas, cada uma com um máximo de 50 pedidos. São comparadas 4 políticas de aceitação de pedidos de *slices* e os dois tipos de agentes anteriormente discutidos. As políticas são aceitar pedidos aleatoriamente (com 50% de probabilidade de aceitar/rejeitar pedidos), aceitar apenas pedidos de *slices* elásticas, aceitar apenas pedidos de *slices* inelásticas ou aceitar todos os pedidos. Os agentes são os discutidos anteriormente: DQN e *dueling DQN*.

A Figura 5.7 contém as médias das *rewards* de cada método em grupos de 50 épocas.

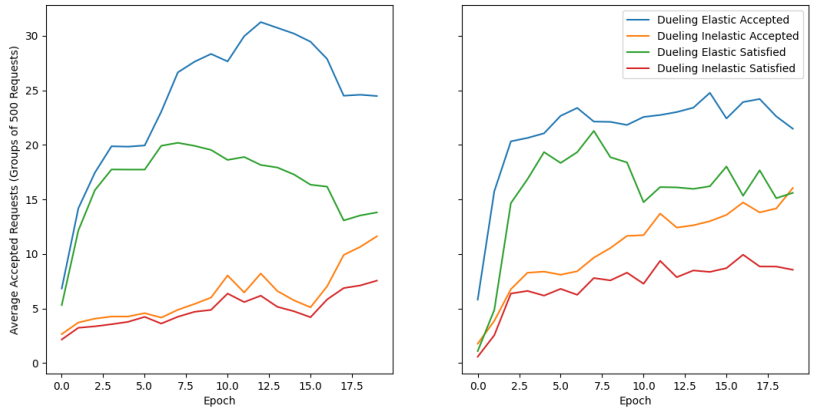

Figura 5.6: Pedidos aceites de *slices* elásticas, inelásticas e satisfeitos DQN vs. *dueling* DQN.

Este gráfico prova a superioridade dos algoritmos com DQN e *dueling* DQN sobre os restantes métodos. Aceitar os pedidos aleatoriamente ou aceitar apenas os pedidos de *slices* elásticas produz uma *reward* semelhante e relativamente baixa, por volta dos 200. Aceitar os pedidos de *slices* inelásticas produz uma *reward* com aproximadamente o dobro do valor das elásticas. Isto deve-se ao facto de as *slices* inelásticas valerem 4 vezes mais que as elásticas mas as *rewards* que produzem quando os pedidos não são satisfeitos retiram a totalidade da sua *reward* inicial. Finalmente, a *reward* de aceitar todos os pedidos, cerca de 500, não atinge a soma das *rewards* de aceitar os pedidos de *slices* elásticas com as de aceitar os de *slices* inelásticas porque ao serem aceites mais pedidos na rede maior se torna a carga nos seus *links* e portanto menos pedidos serão satisfeitos.

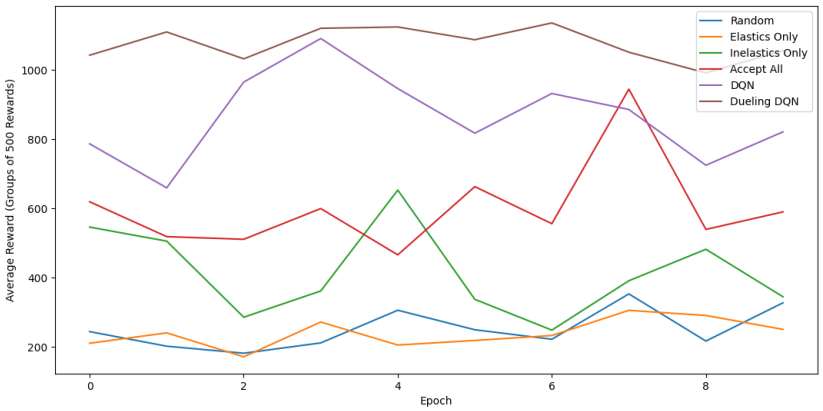

Figura 5.7: *Rewards* dos testes das 4 políticas e 2 algoritmos DRL.

Os algoritmos restantes, DQN e *dueling DQN*, apresentam uma *performance* superior, em termos de *reward*, a todos os algoritmos anteriores. No primeiro caso, é obtida uma

reward média cerca de 4 vezes superior à de admitir *slices* aleatoriamente e aproximadamente 1.5 vezes superior à de admitir todos os *slices*. Com o segundo modelo são obtidos resultados ainda superiores ao primeiro, com uma *reward* média quase 5 vezes superior à de aceitar pedidos de forma aleatória e perto de 2 vezes superior à de aceitar todos os pedidos. No geral, este último algoritmo apresenta uma *performance* 20% melhor que a do primeiro.

As *slices* admitidas e satisfeitas por cada algoritmo podem ser vistas na Figura 5.8. Cada painel desta representa, da esquerda para a direita, os pedidos de *slices* elásticas aceites, inelásticas aceites e pedidos satisfeitos, respetivamente, para todos os algoritmos apresentados.

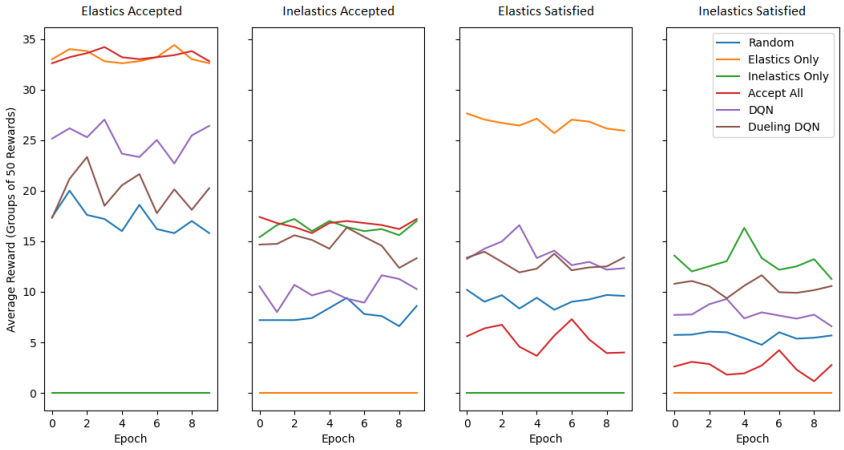

Figura 5.8: Pedidos aceites de *slices* elásticos, inelásticos e satisfeitos durante o teste, da esquerda para a direita.

A partir da figura anterior e relacionando-a com as *rewards* obtidas durante os testes, é possível concluir que, mesmo que o algoritmo com a *dueling* DQN tenha descoberto uma política de admissão de *slices* que leva a uma *reward* superior à dos restantes algoritmos, é inferior na satisfação de pedidos relativamente ao algoritmo com DQN, mesmo que por pouco.

5.2.3 Discussão

Tendo em conta os resultados apresentados nas Secções 5.2.1 e 5.2.2, pode-se concluir que a aplicação de algoritmos com DQNs para a resolução do problema do controlo da admissão de *slices* na rede de transporte, otimizando conjuntamente a utilização da rede e o lucro da operadora é um sucesso. Validam também que o uso de uma arquitetura *dueling* permite atingir resultados superiores em ambientes com um espaço de estados grande.

As solução apresentadas, DQN e *dueling* DQN, apresentam uma *reward* média cerca de 50% e 100%, respetivamente, melhor que a política de aceitar todos os pedidos. A DQN

obteve também uma média de satisfação de pedidos aproximadamente 120% superior ao algoritmo que aceita todos os pedidos. Por outro lado, a satisfação média de pedidos da *dueling* DQN é 100% melhor que a de aceitar todos os pedidos, 20% inferior à DQN padrão. A diferença resulta do facto de a *dueling* DQN dar maior prioridade a *slices* inelásticas que são mais exigentes no cumprimento dos seus SLAs.

Apesar da ligeira inferioridade da *dueling* DQN em satisfazer pedidos elásticos relativamente à DQN original, a sua satisfação de pedidos inelásticos superior, o seu ritmo e estabilidade de aprendizagem e *reward* produzida levam a concluir que esta variação do algoritmo se comporta melhor neste tipo de situação, principalmente considerando a possibilidade de escalar a topologia.

Algo também a considerar nos resultados é o facto de nestas experiências ter sido considerado que os pedidos das *slices* inelásticas valem 4 vezes mais que os das elásticas. Alterar este valor influenciará a prioridade dada aos tipos diferentes de pedidos, podendo levar os agentes a aceitar pedidos de apenas um tipo. Este parâmetro poderia ser alterado de modo a otimizar conjuntamente a *reward* obtida pela operadora e a justiça de resposta a diferentes classes de pedidos.

6

Conclusão

Network slicing é uma das componentes mais importantes para cumprir os KPI propostos para as redes 5G futuras. Espera-se que este paradigma traga novas oportunidades de negócio: as operadoras vendem os seus recursos de rádio, *mobile edge* e *data centers* a *tenants*, que por sua vez vendem serviços aos utilizadores. Um problema a resolver para cumprir estes objetivos é a admissão de novas *slices* na rede de forma a maximizar o lucro e a utilização da rede cumprindo com os SLAs de cada *slice*.

Nesta dissertação propôs-se substituir as técnicas de otimização heurísticas tradicionais já aplicadas a este problema sem grande sucesso, por métodos de DRL. Inicialmente foi feito um estudo do estado-da-arte de algoritmos DRL aplicados a cenários semelhantes, concluindo-se que a utilização de DRL é viável para resolver este problema, e tendo-se optado por fazer a sua aplicação na rede de transporte por esse cenário ser pouco estudado.

No desenvolvimento da solução, começou por se formalizar o problema, os pedidos, a escolha de caminhos entre pontos na rede e finalmente modelou-se o cenário como um MDP. De seguida, utilizando o simulador de redes *Containernet*, o controlador SDN *Ryu*, o simulador de tráfego *Iperf* e o *OpenAI Gym*, criou-se um ambiente de simulação da rede com os qual os agentes podem interagir, obtendo estados e tomando ações de modo a controlar a admissão de *slices*. Os agentes implementam uma DQN e uma *dueling* DQN, respetivamente, e foram criados utilizando a *package* de DL para *Python*, *PyTorch*.

Os resultados obtidos mostram que os algoritmos criados têm um bom desempenho no cenário estabelecido, superando as *rewards* e pedidos satisfeitos de políticas simples. Como esperado do estudo estado-da-arte, a *dueling* DQN obteve uma *performance* ainda superior e com muito maior estabilidade e ritmo de aprendizagem que a DQN.

Apesar dos resultados, existe ainda muito a explorar na aplicação de algoritmos DRL a este cenário. Um dos estudos consistiria na aplicação de uma *double* DQN para melhoria do problema da sobrestimação de *Q-values* inerente à DQN padrão. Poderia também ser combinada uma CNN que, segundo o estudo do estado-da-arte, tem potencial na aprendizagem de relações complexas entre dados próximos no estado, neste caso os *bottlenecks* dos caminhos da rede.

Bibliografia

[1] Y. Abiko et al. "Flexible Resource Block Allocation to Multiple Slices for Radio Access Network Slicing Using Deep Reinforcement Learning". Em: *IEEE Access* 8 (2020), pp. 68183–68198. ISSN: 21693536. DOI: 10.1109/ACCESS.2020.2986050 (ver p. 17).

[2] I. Afolabi et al. "Network slicing and softwarization: A survey on principles, enabling technologies, and solutions". Em: *IEEE Communications Surveys and Tutorials* 20.3 (jul. de 2018), pp. 2429–2453. ISSN: 1553877X. DOI: 10.1109/COMST.2018.2815638 (ver p. 1).

[3] M. Agiwal, A. Roy e N. Saxena. *Next generation 5G wireless networks: A comprehensive survey*. Jul. de 2016. DOI: 10.1109/COMST.2016.2532458 (ver p. 1).

[4] L. L. Ankile, M. F. Heggland e K. Krange. *Deep convolutional neural networks: A survey of the foundations, selected improvements, and some current applications*. 2020. arXiv: 2011.12960 (ver p. 9).

[5] D. Bega et al. "A Machine Learning Approach to 5G Infrastructure Market Optimization". Em: *IEEE Transactions on Mobile Computing* 19.3 (2020), pp. 498–512. DOI: 10.1109/TMC.2019.2896950 (ver p. 20).

[6] D. Bega et al. "Optimising 5G infrastructure markets: The business of network slicing". Em: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 2017, pp. 1–9. DOI: 10.1109/INFOCOM.2017.8057045 (ver p. 20).

[7] M. G. Bellemare, W. Dabney e R. Munos. "A distributional perspective on reinforcement learning". Em: *34th International Conference on Machine Learning, ICML 2017*. Vol. 1. 2017, pp. 693–711. ISBN: 9781510855144. arXiv: 1707.06887 (ver pp. 14, 17).

[8] S. Chen e J. Zhao. "The Requirements, Challenges, and Technologies for 5G of Terrestrial Mobile Telecommunication". Em: *IEEE Communications Magazine* (2014) (ver p. 1).

[9] X. Chen et al. "Multi-Tenant Cross-Slice Resource Orchestration: A Deep Reinforcement Learning Approach". Em: *IEEE Journal on Selected Areas in Communications* 37.10 (jul. de 2019), pp. 2377–2392. ISSN: 15580008. DOI: 10.1109/JSAC.2019.2933893. arXiv: 1807.09350. URL: http://arxiv.org/abs/1807.09350 (ver p. 21).

[10] S. D'Oro et al. "Low-complexity distributed radio access network slicing: Algorithms and experimental results". Em: *IEEE/ACM Transactions on Networking* 26.6 (2018), pp. 2815–2828. ISSN: 10636692. DOI: 10.1109/TNET.2018.2878965. arXiv: 1803.07586 (ver pp. 2, 15).

[11] W. Dabney et al. "Distributional reinforcement learning with quantile regression". Em: *32nd AAAI Conference on Artificial Intelligence, AAAI 2018* (2018), pp. 2892–2901. arXiv: 1710.10044 (ver p. 14).

[12] E. M. Dogo et al. "A Comparative Analysis of Gradient Descent-Based Optimization Algorithms on Convolutional Neural Networks". Em: *2018 International Conference on Computational Techniques, Electronics and Mechanical Systems (CTEMS)*. 2018, pp. 92–99. DOI: 10.1109/CTEMS.2018.8769211 (ver p. 9).

[13] M. Erel et al. "Scalability analysis and flow admission control in mininet-based SDN environment". Em: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. 2015, pp. 18–19. DOI: 10.1109/NFV-SDN.2015.7387396 (ver p. 30).

[14] X. Foukas et al. *Network Slicing in 5G: Survey and Challenges*. Mai. de 2017. DOI: 10.1109/MCOM.2017.1600951 (ver p. 1).

[15] O. N. Foundation. *Mininet: Emulator for rapid prototyping of computer networks*. URL: https://github.com/mininet/mininet (ver p. 30).

[16] I. Goodfellow, Y. Bengio e A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (ver pp. 2, 8, 10).

[17] I. Goodfellow et al. "Generative adversarial networks". Em: *Communications of the ACM* 63.11 (out. de 2020), pp. 139–144. ISSN: 0001-0782. DOI: 10.1145/3422622. URL: https://dl.acm.org/doi/10.1145/3422622 (ver p. 10).

[18] I. Grondman, L. Buşoniu e R. Babuška. "Model learning actor-critic algorithms: Performance evaluation in a motion control task". Em: *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*. 2012, pp. 5272–5277. DOI: 10.1109/CDC.2012.6426427 (ver p. 15).

[19] S. Gu et al. "Continuous Deep Q-Learning with Model-based Acceleration". Em: *CoRR* abs/1603.00748 (2016). arXiv: 1603.00748. URL: http://arxiv.org/abs/1603.00748 (ver p. 13).

[20] V. Gueant. *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. URL: https://iperf.fr/ (ver p. 31).

[21] A. A. Hagberg, D. A. Schult e P. J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". Em: *Proceedings of the 7th Python in Science Conference*. Ed. por G. Varoquaux, T. Vaught e J. Millman. Pasadena, CA USA, 2008, pp. 11–15 (ver p. 32).

[22] Y. Hua et al. *GAN-powered Deep Distributional Reinforcement Learning for Resource Management in Network Slicing*. Rel. téc. 2019. arXiv: 1905.03929v3 (ver p. 17).

[23] M. Jiang, M. Condoluci e T. Mahmoodi. "Network slicing management & prioritization in 5G mobile systems". Em: *European Wireless Conference 2016, EW 2016* (2016), pp. 197–202 (ver pp. 2, 15).

[24] Y. Kim, S. Kim e H. Lim. "Reinforcement learning based resource management for network slicing". Em: *Applied Sciences (Switzerland)* 9.11 (jun. de 2019). ISSN: 20763417. DOI: 10.3390/app9112361 (ver p. 20).

[25] J. Koo et al. *Deep reinforcement learning for network slicing with heterogeneous resource requirements and time varying traffic dynamics*. 2019. arXiv: 1908.03242 (ver p. 22).

[26] M. Lapan. *Deep Reinforcement Learning Hands-On*. 2nd Edition. Birmingham: Packt, 2020 (ver pp. 7, 8, 10–12, 14, 15).

[27] R. Li et al. "Deep Reinforcement Learning for Resource Management in Network Slicing". Em: *IEEE Access* 6 (2018), pp. 74429–74441. ISSN: 21693536. DOI: 10.1109/ACCESS.2018.2881964. arXiv: 1805.06591 (ver pp. 2, 16, 17).

[28] S. Mahon et al. "Performance Analysis of Distributed and Scalable Deep Learning". Em: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 2020, pp. 760–766. DOI: 10.1109/CCGrid49817.2020.00-13 (ver p. 9).

[29] V. Mnih et al. "Human-level control through deep reinforcement learning". Em: *Nature* 518.7540 (2015), pp. 529–533. ISSN: 14764687. DOI: 10.1038/nature14236 (ver p. 12).

[30] V. Mnih et al. "Playing Atari with Deep Reinforcement Learning". Em: (2013), pp. 1–9. arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602 (ver p. 11).

[31] M. S. Olimjonovich. "Software Defined Networking: Management of network resources and data flow". Em: *2016 International Conference on Information Science and Communications Technologies (ICISCT)*. 2016, pp. 1–3. DOI: 10.1109/ICISCT.2016.7777384 (ver p. 31).

[32] A. Paszke et al. "Automatic differentiation in PyTorch". Em: (2017) (ver p. 35).

[33] M. Peuster, H. Karl e S. van Rossem. "MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments". Em: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. Nov. de 2016, pp. 148–153. DOI: 10.1109/NFV-SDN.2016.7919490 (ver p. 30).

[34] C. Qi et al. *Deep reinforcement learning with discrete normalized advantage functions for resource management in network slicing*. 2019. DOI: 10.1109/lcomm.2019.2922961. arXiv: 1906.04594 (ver pp. 13, 17).

[35] M. I. C. Rachmatullah, J. Santoso e K. Surendro. "A Novel Approach in Determining Neural Networks Architecture to Classify Data With Large Number of Attributes". Em: *IEEE Access* 8 (2020), pp. 204728–204743. DOI: 10.1109/ACCESS.2020.3036853 (ver p. 41).

[36] D. E. Rumelhart, G. E. Hinton e R. J. Williams. "Learning representations by back-propagating errors". Em: *Nature* 323 (1986), pp. 533–536 (ver p. 8).

[37] *Ryu - Component-based Software Defined Networking Framework*. URL: https://ryu-sdn.org/ (ver p. 32).

[38] G. Sun et al. "Dynamic Reservation and Deep Reinforcement Learning Based Autonomous Resource Slicing for Virtualized Radio Access Networks". Em: *IEEE Access* 7 (2019), pp. 45758–45772. ISSN: 21693536. DOI: 10.1109/ACCESS.2019.2909670 (ver p. 18).

[39] Y. Sun et al. "User Access Control and Bandwidth Allocation for Slice-Based 5G-and-Beyond Radio Access Networks". Em: *IEEE International Conference on Communications* 2019-May (2019), pp. 1–5. ISSN: 15503607. DOI: 10.1109/ICC.2019.8761841 (ver pp. 2, 15).

[40] H. Van Hasselt, A. Guez e D. Silver. "Deep reinforcement learning with double Q-Learning". Em: *30th AAAI Conference on Artificial Intelligence, AAAI 2016*. 2016, pp. 2094–2100. ISBN: 9781577357605. arXiv: 1509.06461. URL: www.aaai.org (ver pp. 12, 17).

[41] N. Van Huynh et al. "Optimal and Fast Real-Time Resource Slicing with Deep Dueling Neural Networks". Em: *IEEE Journal on Selected Areas in Communications* 37.6 (jun. de 2019), pp. 1455–1470. ISSN: 15580008. DOI: 10.1109/JSAC.2019.2904371. arXiv: 1902.09696 (ver p. 21).

[42] H. Wang et al. "Data-driven dynamic resource scheduling for network slicing: A Deep reinforcement learning approach". Em: *Information Sciences* 498 (2019), pp. 106–116. ISSN: 00200255. DOI: 10.1016/j.ins.2019.05.012. URL: http://hdl.handle.net/10871/37260 (ver p. 19).

[43] Z. Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". Em: *33rd International Conference on Machine Learning, ICML 2016*. Vol. 4. 2016, pp. 2939–2947. ISBN: 9781510829008. arXiv: 1511.06581 (ver pp. 13, 17).

[44] G. Yang et al. "Two-tier resource allocation in dynamic network slicing paradigm with deep reinforcement learning". Em: *2019 IEEE Global Communications Conference, GLOBECOM 2019 - Proceedings*. 2019. ISBN: 9781728109626. DOI: 10.1109/GLOBECOM38437.2019.9014254 (ver p. 17).

[45] A. Zai e B. Brown. *Deep Reinforcement Learning in Action*. Shelter Island: Manning Publications Co., 2020 (ver pp. 2, 11, 14, 15).