UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

GABRIEL JOB ANTUNES GRABHER

Um Estudo de Técnicas de Aprendizado por Reforço para Gerenciamento Consciente de Memória em Ambientes de Processamento de Streams

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. Cláudio R. Geyer
Co-orientador: MSc Kassiano J. Matteussi

Porto Alegre
2021

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões Mendes
Vice-Reitora: Profª. Patricia Helena Lucas Pranke
Pró-Reitor de Pós-Graduação: Profª. Cíntia Inês Boll
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Diretora da Escola de Engenharia: Profª. Carla Schwengber Ten Caten
Coordenador do Curso de Engenharia de Computação: Prof. Walter Fetter Lages
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro
Bibliotecária-chefe da Escola de Engenharia: Rosane Beatriz Allegretti Borges

RESUMO

A área de *Machine Learning* (ML) está em uma fase de grande expansão e sendo aplicada aos mais diversos contextos e ambientes, tais como visão computacional, compreensão de linguagem natural, direção de carros autônomos, entre outros. O gerenciamento dinâmico de recursos é uma das áreas que foi consideravelmente beneficiada por algoritmos de ML e ainda possui diversas possibilidades de inovação, pois permite o aprimoramento de inúmeras técnicas pré-existentes. No contexto *Stream Processing*, *frameworks* como Spark sofrem da falta de gerenciamento dinâmico de recursos no nível de seu gerenciador de memória. Quando o fluxo de dados é muito elevado e as regiões de armazenamento e execução do gerenciador de memória ficam cheias, podem ocorrer perdas de blocos de dados. Além disso, devido a sua prioridade, a memória de execução pode precisar de mais espaço e solicitar da região de armazenamento, automaticamente despejando dados e também causando perdas. Uma solução na forma de buffers de mensagens pode ser usada para orquestrar a pressão de dados enviados ao Spark, evitando os fluxos mais pesados. Porém, tal método precisa de um agente capaz de inferir o gerenciamento dinâmico da ingestão de dados na memória do Spark com base em estatística sobre a execução, se tornando um contexto ideal para a aplicação de um modelo de *Machine Learning*. Portanto, este trabalho busca aplicar o algoritmo Stacked-Autoencoder Q-Network (SAQN) que consiste em utilizar um método de *Deep Reinforcemente Learning* junto ao modelo de rede neural profunda *Stacked-Autoencoder* para gerenciar a ingestão de dados em memória durante a execução de aplicações utilizando o *Spark Streaming*. A solução desenvolvida tem como finalidade maximizar o *throughput* por um ajuste fino enquanto se minimiza o uso de memória. Tal controle é baseado em recompensas de performance e custo que levam em consideração a variação de *throughput* em determinado instante de tempo e o estado global de utilização da memória. Os resultados indicam que a solução proposta é capaz de aprimorar a ingestão de dados em memória para o Spark em diferentes ambientes de execução e aplicações, sem apresentar quedas no sistema e mostrando um controle mais fino entre o ganho de performance, chegando até 25%, e diminuição de custo quando comparado com uma solução heurística somada a um sistema de *backpressure*.

Palavras-chave: Aprendizado por Reforço. Deep-Q-Network. Stacked-Autoencoder. Gerenciamento de Recursos. Stream Processing. Spark. BigData.

A Study on Reinforcement Learning Techniques for Conscious Memory Management in Stream Processing Environments

ABSTRACT

The subject of Machine Learning (ML) is constantly growing and being implemented in various contexts and environments with success, such as computer vision, natural language comprehension, driving autonomous vehicles, etc. Dynamic resource management is one of the areas considerably favored by the surge of ML algorithms, while still having plenty of room to grow, due to the several opportunities to improve on existing techniques. In the context of Stream Processing, frameworks like Spark suffer from the lack of dynamic resource management on the level of its memory manager. The loss of data blocks loss can happen when the data influx is too high and the memory manager's storage and execution regions get full. Also, due to its priority, the execution memory may need more space and require it from the storage section, automatically dumping data and causing data loss. A solution in the shape of message buffers may be used to manage influx of data sent to Spark, avoiding higher pressures. However, such method requires an agent capable of infering the optimal orchestration of data ingestion in memory on Spark based on execution statistics, which is an ideal use case for applying a Machine Learning model. Thus, this work aims to apply the Stacked-Autoencoder Q-Network (SAQN) algorithm, which consists of a Deep Reinforcement Learning method coupled with a Stacked-Autoencoder deep neural network model, to manage the data ingestion in memory during Spark Streaming executions. The developped solution seeks to maximize throughput while minimizing memory usage. This kind of control is based on the usage of perfomance and cost rewards which take in consideration throughput variation in a determined time and the global state of memory utilization. The results indicates that the proposed solution is capable of successfully managing data ingestion on Spark memory for different execution environments and applications, without presenting system crashes and showing a finer control between performance gain, reaching up to 25%, and cost reduction when compared with a heuristic solution plus a backpressure system.

Keywords: Reinforcement Learning, Deep-Q-Network, Stacked-Autoencoder, Resource Management, Stream Processing, Spark, BigData.

LISTA DE FIGURAS

Figura 4.1 Distribuição de Memória de um Executor Spark..26
Figura 4.2 Arquitetura das Message Queues 27
Figura 4.3 Modelo da Rede SAQN 31
Figura 4.4 Diagrama de Sequência do Agente SAQN 33

Figura 5.1 Recompensas com SAQN sem treino prévio...37
Figura 5.2 Recompensas com SAQN e treino prévio 38
Figura 5.3 Utilização Global de Memória para Aplicação SUMServer sem Backpressure 39
Figura 5.4 Throughput Global para Aplicação SUMServer sem Backpressure 40
Figura 5.5 Delay Total para Aplicação SUMServer sem Backpressure 41
Figura 5.6 Cache Global para Aplicação SUMServer sem Backpressure 42
Figura 5.7 Comparação com Baselines em C3 para SUMServer 44
Figura 5.8 Comparação com Baselines em C3 para Global SUMServer 46
Figura 5.9 Comparação com Baselines em C3 para Stateless SUMServer 47

LISTA DE TABELAS

Tabela 3.1 Comparação de trabalhos relacionados 20
Tabela 3.2 Comparação de trabalhos relacionados (cont.)..21

Tabela 5.1 Clusters Usados 34
Tabela 5.2 Configurações de Ambiente 34
Tabela 5.3 Configurações de software do ambiente..35
Tabela 5.4 Variações do modelo SAQN...36
Tabela 5.5 Resultados de Aplicações para Configuração C1 48
Tabela 5.6 Resultados de Aplicações para Configuração C2 48
Tabela 5.7 Resultados de Aplicações para Configuração C3 49
Tabela 5.8 Resultados para SAQN com 4 Threads e configuração C3 e E1 49
Tabela 5.9 Comparação de*throughput* entre Adaptativo BP e SAQN em C3 e E1.........50
Tabela 5.10 Comparação de throughput do SAQN com 1 e 4 Threads em C3 e E1.......50

LISTA DE ABREVIATURAS E SIGLAS

ML	Machine Learning
ANN	Artificial Neural Network
DNN	Deep Neural Network
RNN	Recurrent Neural Network
CNN	Convolutional Neural Network
PCA	Principal Component Analysis
RL	Reinforcement Learning
MDP	Markov Decision Process
DQN	Deep-Q-Network
A3C	Asynchronous Advantage Actor-Critic Algorithm
LSTM	Long Short-Term Memory

SUMÁRIO

1 INTRODUÇÃO **9**
2 FUNDAMENTOS **12**
2.1 Aprendizado Supervisionado **12**
2.2 Aprendizado Não-Supervisionado **13**
2.3 Aprendizado por Reforço **13**
2.4 Framework Spark **15**
3 TRABALHOS RELACIONADOS **16**
3.1 Metodologia de Busca e Pesquisa de Artigos Relacionados **16**
3.1.1 Questões de Pesquisa 16
3.1.2 String de Busca 16
3.1.3 Critérios de Inclusão e Exclusão 17
3.1.4 Critérios Qualitativos 17
3.1.5 Seleção Final 19
3.2 Trabalhos Selecionados **22**
3.3 Discussão **24**
4 PROPOSTA **25**
4.1 Contexto do Problema **25**
4.2 Abordagem da Proposta **27**
4.3 Implementação **28**
4.3.1 Representação do Estado da Execução 28
4.3.2 Representação da Ação do Agente 30
4.3.3 Representação da Política de Recompensa 30
4.3.4 Modelo da Rede Neural 30
4.3.5 Integração do Agente nas Message Queues 32
5 AVALIAÇÃO DA PROPOSTA **34**
5.1 Análise da Função de Recompensa **36**
5.2 Comparação com Baselines **43**
5.2.1 Execuções com Múltiplas Threads 49
5.3 Discussão **50**
6 CONCLUSÃO **52**
6.1 Considerações Pessoais **52**
6.2 Direções Futuras **53**
REFERÊNCIAS **54**

1 INTRODUÇÃO

De acordo com a definição do "modelo padrão", a Inteligência Artificial (IA) é abordada como um agente racional que age com o objetivo de atingir o melhor resultado, ou em casos de incerteza, o melhor resultado esperado (NORVIG, 2020). Já o Aprendizado de Máquinas (*Machine Learning*), é uma sub-área da IA que pode ser definida como um conjunto de métodos que podem detectar automaticamente padrões em dados e, em seguida, usar os padrões descobertos para prever dados futuros (MURPHY, 2012).

Com o constante avanço de tecnologias na área da computação, vemos a incessante digitalização do mundo ao nosso redor: redes sociais, Internet das Coisas (IoT), redes de comunicação (WiFi, 5G, fibra ótica), *hardwares* (SSDs, GPUs, dispositivos móveis, *Smartphones*), etc. E devido às grandes quantidades de dados e o poder de processamento obtidos por essas tecnologias, a área de *Machine Learning* pode desenvolver-se consideravelmente, sendo capaz de permitir o surgimento de aplicações de compreensão da fala humana (AMODEI et al., 2016), reconhecimento facial (TAIGMAN et al., 2014), competição em jogos de alto nível estratégico (MNIH et al., 2013) (SILVER et al., 2016) , processo de controle sem motorista de carros autônomos (HUVAL et al., 2015), entre outros.

Muitas das novas tecnologias mencionadas utilizam ferramentas de *BigData*, como Apache Spark, para gerenciar a grande quantidade de dados com que elas trabalham e prover *insights* em tempo real. Então o melhor gerenciamento de recursos para esses processamentos em larga escala se torna um ponto importante onde podem ser usadas técnicas de ML para realizá-los mais efetivamente.

Ao realizar-se uma busca por artigos relacionados à utilização de aprendizado de máquina para gerenciamento de recursos nos últimos 5 anos, percebe-se a falta de um estudo ou algum trabalho que condense o estado da arte mais recente, principalmente as soluções após o fim do segundo "inverno"na área de *Machine Learning* quando as *Deep Neural Networks* se tornaram mais poderosas (NEWQUIST, 2018). E também, foram encontradas poucas técnicas de ML na literatura explicitamente relacionadas ao *frameworks* de *BigData*, então trabalhos que avaliem a viabilidade de usá-las nesse meio se fazem necessários. Além disso, tais *frameworks* sofrem com a falta de gerenciamento dinâmico de seus recursos em tempo de execução, o que pode levar à perda de dados e computação, ponto onde redes neurais poderiam facilmente trazer benefícios, sendo capazes de reconhecer padrões do estado mais atual do ambiente, ou até mesmo do histórico de execução

no caso de Recurrent Neural Networks (RNNs).

No contexto de *Stream Processing* utilizando o *framework* Spark, a memória é dividida em duas seções: uma para execução e outra para armazenamento. Porém, a memória de execução tem prioridade sobre a de armazenamento, de modo que se a memória de execução precisar de mais espaço por causa de um processamento muito intensivo, ela irá solicitar a seção da área de armazenamento, automaticamente despejando dados e causando a perda de dados. Uma solução na forma de filas de mensagens (*Message Queues*) pode ser usada para orquestrar e controlar o fluxo de dados enviados ao Spark, evitando esse despejamento de dados. Entretanto, tal método precisa de um agente capaz de inferir de forma dinâmica o gerenciamento da ingestão de dados na memória do Spark com base em estatística sobre a execução. Se tornando um contexto ideal para a aplicação de um modelo de *Machine Learning*.

Por conta disso, estão inclusas como objetivo deste trabalho as seguintes contribuições:

- Realizar uma pesquisa sistemática da literatura sobre soluções de *machine learning* para o escalonamento de recursos em ambientes distribuídos e apontar os modelos mais relacionados ao gerenciamento de ingestão de dados para memória e *Stream Processing*;
- Implementar o modelo de ML mais relevante para o contexto do trabalho encontrado na pesquisa, e adaptar para o ambiente de *Stream Processing*, caso necessário;
- Avaliar tal modelo em um contexto real de gerenciamento de dados inseridos na memória de aplicações *Spark Streaming*, comparando-a com com a solução heurística pré existente e observar as diferenças em performance.

No capítulo 2 são apresentados fundamentos sobre aprendizado de máquina para auxiliar na compreensão do contexto do trabalho. Em seguida, no capítulo 3 é explicado como foi feita a revisão sistemática da literatura e também são indicados os trabalhos relacionados obtidos. No capítulo 4, a proposta sobre a utilização de *Message Queues* com o Spark é explicada em mais detalhes, e como o algoritmo de *Reinforcement Learning* proposto se encaixa no contexto. Assim como o contexto do trabalho é apresentado formalmente. Ainda no mesmo capítulo na seção 4.3 são demonstrados detalhes da implementação da solução e como o ambiente foi representado. E na seção 4.4 e 4.5 estão presentes avaliações com o modelo e comparações de performance com *baselines* e uma discussão final sobre os resultados obtidos. Finalmente, no capítulo 5 são discutidas as

contribuições deste trabalho, assim como dificuldades enfrentadas e possíveis caminhos em como complementar os resultados obtidos.

2 FUNDAMENTOS

2.1 Aprendizado Supervisionado

Segundo a definição de (NORVIG, 2020), em Aprendizado Supervisionado, um agente observa pares de "entrada-saída"e aprende uma função para mapear cada entrada para uma saída. Tais tipos de saídas podem ser chamados de classes, ou *labels*. Em geral, fazem parte desse grupo redes neurais como *Deep Neural Networks* (DNNs), *Convolutional Neural Networks* (CNNs), *Recurrent Neural Networks* (RNNs), *Multi-layer Perceptron* (MLP), *Support Vector Machines* (SVM), aprendizado por Kernels, Naive-Bayes e Florestas Aleatórias. Porém, no contexto de gerenciamento de recursos os algoritmos encontrados foram: DNNs e redes *Long Short Term Memory* (LSTM) que são um tipo específico de RNN.

Uma rede neural profunda ou *Deep Neural Network* (DNN) é uma rede neural artificial com múltiplas camadas internas (ocultas) entre suas entradas e saídas, por isso são chamadas de "profundas"(SCHMIDHUBER, 2015). Assim como as rede neurais artificiais, DNNs foram inspiradas em redes neurais biológicas em suas camadas formadas por neurônios e sinapses. Redes de arquitetura profunda possuem diversas variações com adaptações para domínios específicos, como RNNs para reconhecimento de linguagem ou CNN para reconhecimento de imagens.

LSTMs são um tipo especial de *Deep Recurrent Neural Network* (RNN), capazes de armazenar informações por um longo período de tempo com o intuito de aprender a dependência a longo-prazo de uma sequência de entradas (HOCHREITER; SCHMIDHUBER, 1997). LSTMs processam uma sequência de tamanho variável $y = (y_1, y_2, ..., y_m)$ adicionando novos dados de maneira incremental em uma memória, com um limite definido de até onde dados antigos devem ser armazenados. Como as redes LSTM possuem a capacidade de disponibilizar resultados em um certo *time-step* considerando os resultados dos *time-steps* anteriores. Elas se encaixam bem com as necessidades de gerenciamento de recursos onde possa ser útil lembrar do histórico de utilização de uma máquina por exemplo.

2.2 Aprendizado Não-Supervisionado

Por (NORVIG, 2020), em Aprendizado Não-Supervisionado, um agente aprende padrões em uma entrada sem nenhum tipo de indicação explícita por parte do programador. Dois dos principais métodos utilizados são PCA (*Principal Component Analysis*) e Clusterização (*Cluster Analysis*). A intuição é identificar um k número (pré-definido) de grupos ou componentes e associar os dados de *input* ao grupo/componente mais "similar". Outros métodos incluem detecção de anomalias entre pares de inputs e recriação de dados com base em um conjunto de inputs (por exemplo criar imagem de um gato a partir de um *dataset* de imagens de gatos). No contexto de gerenciamento de recursos, *autoencoders* são amplamente utilizados.

Um *autoencoder* é uma rede neural capaz de copiar seus dados de entrada para sua saída, de forma que tais dados sejam "codificados"e a rede aprenda a nova representação desejada (KRAMER, 1991). Geralmente é utilizado para redução de dimensionalidade ao ignorar os ruídos presentes na entrada. O *autoencoder* possui duas partes principais: um *encoder* responsável por mapear a entrada para código, e um *decoder* que mapeia o código para uma reconstrução da entrada.

2.3 Aprendizado por Reforço

Os agentes de Aprendizado por Reforço ou Reiforcement Learning (RL) aprendem por um conjunto de recompensas e punições. Fica a critério do agente de decidir quais ações que levaram até uma recompensa foram responsáveis por ela, como alterar as ações tomadas de forma a receber mais recompensas maiores no futuro (NORVIG, 2020). Abordagens por RL se encaixam muito bem com gerenciamento de dados. Um dos motivos é que eles não precisam de um modelo preciso definido a priori, frequentemente chamados de independentes de modelo ou *modelfree*, facilitando o treinamento para objetivos que são difíceis de optimizar. Outro motivo vem do fato que geralmente decisões feitas nesses sistemas são repetitivas, gerando uma grande quantidade de dados de treinamento para algoritmos de RL. Além disso, RL consegue modelar sistemas complexos e protocolos de tomada de decisão como DNNs, assim dados "brutos"e repletos de interferências (*noise*) podem ser utilizados como entrada nessas redes neurais, e a estratégia resultante pode ser usada em um ambiente online e estocástico. Por último, como o agente RL aprende continuamente no sistema, ele pode ser optimizado para objetivos diferentes

(e.g., *jobs* pequenos, poucas tarefas, periodicidade). As soluções encontradas na literatura implementavam alguma forma de *Q-learning*, *Deep Reinforcement Learning* (DRL), *Deep-Q-Network* (DQN), *Markov Decision Process* (MDP) ou *Asynchronous Advantage Actor-Critic Algorithm* (A3C).

Um processo de decisão de Markov, ou *Markov Decision Process* (MDP), é um *framework* matemático para modelar sistemas de decisão, onde para cada instante t o processo se encontra em um estado s, e o agente responsável por tomada de decisões escolhe um ação a possível em s, onde $s \in S, a \in A$, de forma que resulte em uma mudança a um novo estado s' e uma recompensa R ao agente (HOWARD, 1960). A probabilidade de mudança de estado é influenciada pela ação escolhida e pelo estado atual, porém independente de todos estados e ações anteriores.

Q-learning é um algoritmo de aprendizado por reforço que funciona com base em um agente, um conjunto de estados S, um conjunto de ações por estado A e uma Tabela de *Q-values* para cada par (s, a), representando a expectativa de recompensa (WATKINS, 1989). Ao executar uma ação, o agente muda de um estado para outro, onde cada estado gera uma recompensa (um valor numérico) e o valor de $Q(s, a)$ é atualizado através da equação de Bellman:

$$Q(s,a)^{new} = Q(s,a) + \underbrace{\alpha}_{\textit{Learnig rate}}[\underbrace{R(s,a)}_{\textit{Reward}} + \underbrace{\gamma}_{\textit{Discount rate}} \overbrace{\max Q'(s',a')}^{\substack{\textit{Maximum predicted reward, given} \\ \textit{new state and all possible actions}}} - Q(s,a)] \tag{2.1}$$

Além disso, para cada processo de decisão de Markov finito ou *Finite Markov Decision Process* (FMDP), o *Q-learning* consegue encontrar a solução ótima para maximização da recompensa total esperada para todos os estados, começando de um dado estado inicial (MELO, 2001).

Em vários caso reais de problemas de tomada de decisão, os estados e ações de um MDP podem apresentar uma dimensionalidade muito alta (*e.g.* imagens de um jogo ou dados brutos de um sensor), tornando a solução impossível para algoritmos RL tradicionais como Q-learning. Então, algoritmos de *Deep Reinforcement Learning* (DRL) surgiram como uma solução tais MDPs, ao representar as Tabelas de *Q-values* (no caso de *Deep-Q-Networks*) ou outras funções aprendidas como redes neurais, aumentando a capacidade de representação de altas dimensões (MNIH et al., 2015).

Avanços significativos foram feitos na área de DRL ao jogar jogos como AlphaGo

(SILVER et al., 2016) ou Atari (MNIH et al., 2013) em níveis superiores a humanos, demonstrando o poder de trabalhar com controle de um grande espaço de estados e ações.

2.4 Framework Spark

O Apache Spark é um motor unificado para processamento de dados em larga escala, se baseando em um *driver* que controla em alto-nível o fluxo das aplicações e a execução de operações em paralelo (ZAHARIA et al., 2010) em diferentes executores de um dado *cluster*. O Spark implementa duas abstrações para programação paralela: os *resilient distributed datasets* (RDDs), e operações paralelas em cima desses *datasets*. O primeiro é uma coleção *read-only* de objetos particionada em um conjunto de máquinas, permitindo que ela possa ser reconstruída caso umas das partições seja perdida ou ocorra uma falha nos nós. Já as operações que podem ser efetuadas em RDDs são:

- **reduce:** Combina elementos do *dataset* a partir de uma dada função associativa para produzir um resultado no nível do *driver;*
- **collect:** Envia todos elementos do *dataset* para o *driver*;
- **foreach:** Executa uma função definida pelo usuário em cada elemento do *dataset*.

Além disso, o Spark disponibiliza o uso de dois tipos de variáveis compartilhadas para serem usadas em funções executando no *cluster*:

- **Variáveis de Broadcast:** Objeto que encapsula o valor de um dado e garante que ele só sera copiado uma vez para cada executor;
- **Variáveis Acumuladoras:** São variáveis que só podem ser incrementadas a partir de uma função, e também apenas o *drive* pode ler o seu valor. Acumuladores podem ser definidos para qualquer tipo que possuam uma função de incremento e um valor inicial.

3 TRABALHOS RELACIONADOS

3.1 Metodologia de Busca e Pesquisa de Artigos Relacionados

Esta seção descreve o método utilizado para identificar e resumir os trabalhos relacionados. Este trabalho adota as diretrizes para execução de pesquisas sistemáticas apresentadas por (KITCHENHAM, 2004). Além disso, a pesquisa e as análises dos critérios de inclusão, exclusão e qualitativos são auxiliadas por uma ferramenta[1] em Python desenvolvida durante a execução deste trabalho. Em seguida, serão apresentados as Questões de Pesquisa (QP), a String de pesquisa empregada, os critérios de exclusão (EC), de inclusão (IC) e qualitativos (QC). E por fim, o padrão utilizado para a seleção final dos artigos que seriam efetivamente lidos.

3.1.1 Questões de Pesquisa

As Questões de Pesquisa são definidas para ajudar a estabelecer quais são os principais atributos buscados nos trabalhos relacionados. São elas:

- QP.1: Qual é o estado-da-arte no uso de Machine Learning para o gerenciamento de recursos e quais suas vantagens com relação as métodos clássicos?
- QP.2: Como tais soluções são avaliadas, quais métricas são utilizadas?
- QP.3: Qual técnica de Machine Learning é empregada?
- QP.4: Qual é o tipo de recurso gerenciado por essa solução e como ele é representado?

3.1.2 String de Busca

Nossa String de Pesquisa contém termos bem definidos para auxiliar na definição do escopo da busca entre as soluções existentes, assim como indicar lacunas e tendências no conhecimento do uso de aprendizado de máquina para gerenciamento de recursos.

Foi utilizado o Google Scholar como principal motor de busca na ferramenta em Python mencionada acima, baseando-se na biblioteca *scholarly*, pois ele é capaz de classificar as melhores conferências e periódicos em diversas áreas, padronizando a pesquisa

[1] https://github.com/gabrijob/paperworm

entre elas. Facilitando o uso de uma mesma String de Pesquisa para cada repositório digital disponível. A string definida foi:

allintitle: "* learning"("resource*"OR "task*") ("management"OR "scheduling"OR "orchestration"OR "provisioning"OR ”tuning"OR "placement"OR "optimization") +site:<repositório digital>

Repositório digital estando presente no conjunto *[acm.org, ieee.org, sciencedirect.com, onlinelibrary.wiley.com, springer.com, mdpi.com]*.

3.1.3 Critérios de Inclusão e Exclusão

Os Critérios de Inclusão (IC) e Exclusão (EC) foram aprimorados conforme a seleção era feita e filtros eram aplicados até chegarem no estado aqui estabelecido. Os critérios de inclusão são:

- IC1: Está escrito em Inglês;
- IC2: Foi publicado entre 2015 e 2020;
- IC3: Apresenta um modelo de machine learning para gerenciamento de recursos;
- IC4: Foi publicado em alguma das revistas do conjunto {IEEE, ACM, ELSEVIER, WILEY, MDPI, SPRINGER}.

Os critérios de exclusão são:

- EC1: É uma patente;
- EC2: É uma duplicata;
- EC3: Possui menos de 5 páginas;
- EC4: Não possui um ano de publicação informado.

3.1.4 Critérios Qualitativos

Medir a qualidade de um conjunto de artigos não é simples, exige tempo e o processo varia de acordo com a experiência do autor, as fontes buscadas, vieses, o contexto da busca, entre outros. Nesta seção, será proposta uma abordagem de classificação com o intuito de selecionar os artigos relevantes da literatura de aprendizado de máquina para gerenciamento de recursos. Os critérios de avaliação utilizados buscam balancear entre as soluções mais bem estabelecidas e as mais novas. Os critérios são:

i) Ano: aqui quanto mais atual, maior a nota. Assim, o ano atual recebe a nota máxima de 1 (um) ponto e o ano mais antigo, a menor. Então, a nota é normalizada pelo intervalo de anos (de 2015 à 2020 no caso deste trabalho), na forma:

$$Score = PaperYear \div (2020 - 2015 + 1) \tag{3.1}$$

Nesse caso, o ano de 2020 também é considerado na classificação, por isso o "+1".

ii) Número de citações: este critério é dividido em duas partes de 0,5 pontos cada. Inicialmente, a média do número de citações é calculada e é utilizada para a primeira parte da nota, na qual um artigo recebe 0,5 pontos caso ele possua um número de citações maior que a média, senão a nota é a divisão entre os dois valores:

$$Score1 = 0.5\ ,\ \{PaperCitationNb > AvgCitationNb\} \tag{3.2}$$

senão,

$$Score1 = 0.5 \times PaperCitationNb\ \div AvgCitationNb \tag{3.3}$$

A segunda parte é obtida a partir do cálculo do desvio padrão, e só é aplicada aos artigos que possuem um número de citações maior do que a média. Nesse caso, um artigo recebe 0,5 pontos caso alcance a média de citações mais dois desvios padrões, senão a nota é a diferença entre o número de citações obtido pelo artigo dividida por 2 desvios padrões:

$$Score2 = 0.5\ ,\ \{(PaperCitationNb - AvgCitationNb) > 2 \times StdDeviation\} \tag{3.4}$$

senão,

$$Score2 = 0.5 \times (PaperCitationNb - AvgCitationNb) \div 2StdDeviation \tag{3.5}$$

Finalmente, o critério é calculado pela soma das duas notas, dessa forma os artigos são avaliados de maneira mais justa de acordo com a distribuição dos números de citações.

iii) Número de páginas: este critério é calculado a partir da média de páginas em todo o conjunto de artigos. De forma que um artigo que contenha uma quantidade igual

ou superior a média recebe 1 ponto, caso contrário o valor é normalizado pela média:

$$Score = 1\ ,\ \{PaperPagesNb > AvgPagesNb\} \tag{3.6}$$

senão,

$$Score = PaperPagesNb\ \div AvgPagesNb \tag{3.7}$$

iv) Google Rank: o último critério é baseado na ordem em que os artigos aparecem no Google Scholar no momento da pesquisa. Como explicado em (BAEZA-YATES; RIBEIRO-NETO, 2013), é sabido que o Google considera diversas métricas para apresentar os resultados, tais como seu próprio h-index, pagerank, número de cliques em uma página, entre outros. Então, o critério é calculado considerando a ordem original do Google, de forma que o primeiro artigo receba 1 ponto e os outros recebam uma nota normalizada pelo número total de artigos:

$$Score = (papersTotal - googleRank + 1) \div papersTotal \tag{3.8}$$

Finalmente, a pontuação final de um dado artigo é a soma de todos os scores acima, onde o valor máximo possível sendo uma pontuação de 4,0. Após calculada a classificação para todos os artigos, foi selecionado um ponto de corte referente a 80% da nota máxima observada (3,65), no qual todos artigos com pontuação igual ou maior ao ponto de corte foram selecionados para uma análise manual. Nessa etapa, foram selecionados no total 50 trabalhos para a análise final.

3.1.5 Seleção Final

Para escolher os artigos que serão lidos na íntegra, foi realizado uma leitura dos títulos e resumos dos obtidos pela nota de corte na etapa anterior, ao mesmo tempo que eles eram categorizados em alta, média ou baixa relevância. Finalmente, os 16 artigos classificados com altamente relevantes foram escolhidos (Tabelas 3.1 e 3.2), com os médios na reserva, caso necessários nas próximas fases do trabalho.

Como pode-se observar nas Tabelas 3.1 e 3.2, 10 dos 16 artigos selecionados

Tabela 3.1: Comparação de trabalhos relacionados

Trabalho	*Citações*	*Algoritmo*	*Ambiente*
(MAO et al., 2016)	448	REINFORCE, Deep Reinforcement Learning	Cluster como um único conjunto de recursos.
(SUN et al., 2017)	148	WMMSE (Weighted Minimum Mean Square Error), Deep Neural Network	Gerenciamento de energia em redes wireless.
(WEI et al., 2017)	119	Actor-Critic Reiforcement Learning	Escalonamento de usuários e alocação de recursos energéticos em redes heterogêneas.
(LIU et al., 2017)	101	LSTM (Long Short Term Memory), Deep Reinforcement Learning	Alocação de VMs e gerenciamento de energia em ambientes de nuvem.
(ZHANG; YAO; GUAN, 2017)	45	Deep-Q-Network (DQN), Stacked-Autoencoder (SA)	Alocação de recursos computacionais em ambientes de nuvem Apache Hadoop.
(CHALLITA; DONG; SAAD, 2018)	97	LSTM (Long Short Term Memory), Multi-Layer Perceptron (MLP)	Escolha de canais para comunicações em espectros não-licenciados de redes do tipo *licensed assisted access long term evolution* (LTE-LAA).
(LUONG et al., 2018)	88	Deep Neural Network (DNN)	Alocação de mineradores em Edge Computing Service Provider (ECSP) de redes blockchain móveis.
(BENIFA; DEJEY, 2019)	23	RL-SARSA (RLPAS)	Alocação e dimensionamento de VMs em pequena, média e grande para ambientes de nuvem.
(YANG et al., 2019)	23	Post-Decision State Experience Replay and Transfer (PDS-ERT), Deep Reinforcement Learning	Seleção de rede, atribuição de sub-canais e gerenciamento de energia para redes IoT industriais.
(MANDAL et al., 2019)	15	Imitation Learning (IL)	Controle de tipo, número e frequência de núcleos ativos em plataformas mobile heterogêneas de SoCs (system-on-chip).
(CHEN et al., 2019)	18	Echo State Networks (ESNs), Transfer Learning	Otimização da alocação de blocos e transmissão de dados em uplink e downlink para redes de realidade virtual wireless.
(CHEN; SAAD; YIN, 2019)	53	Liquid State Machine (LSM)	Gerenciamento de associação de usuários, alocação de espectro e caching em redes de Unmaned Aerial Vehicles (UAVs) e Licensed Assisted Access (LTE).

Fonte: Os Autores

Tabela 3.2: Comparação de trabalhos relacionados (cont.)

(CHEN et al., 2019)	26	Deep-Q-Network (DQN)	Gerenciamento de descarga de computação e escalonamento de pacotes em um ambiente de jogo estocástico não-cooperativo entre servidores Mobile-Edge para usuários mobile.
(CHEN et al., 2020)	23	Asynchronous Advantage Actor-Critic (A3C)	Gerenciamento de computação, comunicação e cache em uma rede de veículos composta por redes virtuais mobile, Base Stations e servidores Mobile Edge Computing.
(SHAHIDINEJAD; GHOBAEI-ARANI, 2020)	2	Learning Automata, Long Short Term Memory (LSTM), Q-Learning	Orquestração dinâmica de tarefas e nós em redes com camadas mobile, edge e cloud .
(LU et al., 2020)	10	Deep-Q-Network (DQN), Long Short Term Memory (LSTM), Candidate Networks	Orquestração da descarga de tarefas em redes de Mobile Edge Computing (MEC).

Fonte: Os Autores

implementam algum modelo de aprendizado por reforço, que pode ser explicado pelo fato do contexto de gerenciamento de recursos geralmente lidar com aplicações difíceis de serem representadas em *datasets* de treinamento, sendo necessário que o aprendizado seja feito durante a execução. Outro fator que favorece o uso de aprendizado por reforço é a necessidade de métodos *modelfree* que possam ser flexíveis com as variações de uma execução para outra por conta do ambiente não-determinístico do contexto.

Também pode-se verificar que a visão geral do estado-da-arte presente nas Tabelas 3.1 e 3.2 e a tendência explicada no parágrafo anterior respondem a questão de pesquisa QP.1, de modo que a principal vantagem com relação aos métodos clássicos é a flexibilidade que o aprendizado de máquina oferece. De acordo com a leitura dos trabalhos, a QP.2 pode ser condensada de maneira geral nas métricas de performance e custo, porém cada ambiente considera diferentes representações dessas métricas. Já para a QP.3, cada técnica de ML utilizado está presente na terceira coluna das Tabelas 3.1 e 3.2. Finalmente, os tipos de recursos gerenciados para a QP.4 estão indicados na quarta coluna das tabelas e como eles são representados depende do algoritmo utilizado, dificultando a inserção de todos eles no texto deste trabalho.

3.2 Trabalhos Selecionados

Nesta seção serão apresentadas diversas soluções encontradas pela pesquisa sistemática da literatura da utilização de *Machine Learning* para gerenciamento de recursos. Vale notar que caso o leitor esteja interessado em casos mais específicos, já foram feitos *surveys* de ML para redes IoT em (HUSSAIN et al., 2020) ou para redes Edge-Cloud em (DUC et al., 2019). Neste trabalho, o objetivo é de uma análise mais generalizada do estado atual da área e possíveis tendências.

Em (MAO et al., 2016) foi apresentado um modelo generalizado para representar recursos como CPU, memória, I/O, etc e *jobs* pelos requisitos desses recursos, onde um *cluster* é uma coleção de todos os recursos, com o objetivo de treinar uma DRL com base nessa representação. Em (SUN et al., 2017) foi proposto uma DNN para aproximar o mapeamento do algoritmo Weighted Minimum Mean Square Error (WMMSE) já utilizado para o gerenciamento de energia, melhorando a performance e obtendo o resultado mais rapidamente.

No ambiente *Cloud*, (LIU et al., 2017) propuseram a utilização de uma DRL atuando no *job broker* da *Cloud* para a alocação de VMs em conjunto com uma rede LSTM localizada localmente em cada máquina para predição de *workload* e gerenciamento do consumo de energia. O trabalho (ZHANG; YAO; GUAN, 2017) desenvolve uma solução de uma DQN combinada com um *stacked autoencoder* (SA) para estimar a função Q, chamado de *Stacked-Autoencoder Q-Network* (SAQN), o qual é utilizado para ajustes na alocação de um único recurso por vez, como CPU ou memória. Além de ser avaliado com base em SmartYARN, uma extensão do sistema operacional distribuído em larga-escala para BigData implementado pelo Apache Hadoop(YARN) equipado com RL. Em (BENIFA; DEJEY, 2019) foi apresentado uma solução de escalonamento automático de VMs em ambientes *Cloud* utilizando um algoritmo SARSA de aprendizado por reforço com base em utilização de CPU, tempo de resposta e *throughput*. Em (CHEN et al., 2019) propõe-se um algoritmo de *Transfer Learning* baseado em *Echo State Networks* (ESNs), um tipo de *Recurrent Neural Network* (RNN), para alocação de blocos de recursos e otimização da transmissão de dados em *uplink* e *downlink* em redes *wireless* para realidade virtual (VR).

Já no ambiente Edge, (CHEN et al., 2019) propõe um algoritmo DQN para otimizar o *offloading* de recursos computacionais e as políticas de escalonamento de pacotes num jogo estocástico não-cooperativo entre empresas provedoras de serviço (*Service Pro-*

viders) para usuários mobile. Em (LU et al., 2020) apresenta-se um modelo de DQN junto com LSTM e *Candidate Networks* para o escalonamento de tarefas em plataformas *Mobile Edge Computing* (MEC), reduzindo latência, custo, energia e utilização da rede. Em (SHAHIDINEJAD; GHOBAEI-ARANI, 2020) propõe-se uma abordagem com *Learning Automata* para transferência das cargas dinâmicas de trabalho de aplicações mobile para servidores *Edge* ou *Cloud*, em conjunto com uma combinação de um modelo LSTM para prever as cargas futuras e *Q-learning* para determinar o número apropriado de servidores *Edge* para lidar com flutuações na carga. Ainda em *Edge* mas no contexto de *Blockchain*, (LUONG et al., 2018) desenvolve-se um algoritmo DNN para alocação de mineradores de um *Edge Computing Service Provider* (ECSP) em redes *blockchain mobile*, considerando principalmente a receita da ECSP. Mais especificamente na área de redes de veículos, (CHEN et al., 2020) utilizou um modelo de aprendizado por reforço *Asynchronous Advantage Actor-Critic* (A3C) para gerenciar recurso de computação, comunicação e cache.

Considerando redes LTE, (CHALLITA; DONG; SAAD, 2018) implementa um modelo LSTM *encoder-decoder* junto com um Multi-Layer Perceptron (MLP) para prever a *workload* de *Small Base Stations* (SBS) e alocar os canais apropriados de uma LTE-LAA de forma justa utilizando um modelo de jogo não-cooperativo, de Teoria dos Jogos. Enquanto (CHEN; SAAD; YIN, 2019) propõe duas implementações de agentes Liquid State Machines (LSM) para gerenciar recursos de *Unmaned Aerial Vehicles* (UAV) em redes LTE, um que executa na *cloud* responsável pela gerência dos dados nas caches dos UAVs, e outro para seleção de canal LTE, que roda nas UAVs. No contexto de redes heterogêneas (HetNets) como fornecimento de energia híbrida entre fontes renováveis ou não, (WEI et al., 2017) apresenta uma solução para escalonamento de usuários e alocação de recursos para fazer uso de energia renovável coletada por pequenas células usando um algoritmo de RL *Actor-Critic*.

Dentro do contexto de IoT, (YANG et al., 2019) propôs um algoritmo DRL com *Post-Decision State Experience Replay and Transfer* (PDS-ERT) para otimizar a política de seleção de rede, atribuição de sub-canais e gerenciamento de energia em redes IoT na indústria 4.0. E com SoCs em mente, (MANDAL et al., 2019) apresentou um *framework* de *Imitation Learning* (IL) para controlar dinamicamente o tipo (Big/Little), quantidade e frequência de *cores* ativos em plataformas heterogêneas mobile.

3.3 Discussão

As soluções propostas por (CHEN et al., 2020), (ZHANG; YAO; GUAN, 2017) e (CHEN; SAAD; YIN, 2019) foram escolhidas como as relevantes, pois os dois primeiros utilizam representações mais abrangentes de recursos, sem dependência em um tipo específico como consumo de energia ou comunicação, podendo ser mais facilmente adaptadas ao gerenciamento de ingestão de dados em memória. Além disso, (CHEN; SAAD; YIN, 2019) foi o único trabalho entre os analisados a especificamente gerenciar caches ou memória, que apesar de ser proposto para redes UAV, a solução será consideravelmente interessante se adaptada ao contexto *Stream Processing* em *Clouds*. Por outro lado, (ZHANG; YAO; GUAN, 2017) já apresenta comparações com o SmartYarn do Apache Hadoop, que são altamente relevantes com o contexto deste trabalho. E também, como o algoritmo utilizado é um DQN com Stacked-Autoencoder, portanto um aprendizado por reforço, a implementação será livre de modelo (*modelfree*), de maneira que treinamento prévio não seja necessário para a rede funcionar e que o agente seja mais flexível para diferentes níveis de pressão de dados. Tornando-o o melhor indicado para esta etapa de implementação. Assim as outras duas abordagens serão consideradas como possíveis trabalhos futuros.

4 PROPOSTA

A proposta deste trabalho consiste em implementar o modelo *Stacked-Autoencoder Q-Network* (SAQN) selecionado pela pesquisa nos trabalhos relacionados, e realizar adaptações necessárias para utilizá-los em um ambiente de *Spark Streaming* para gerenciamento dinâmico de ingestão de dados em memória. Esse tema foi escolhido tomando em conta a tese de doutorado do coorientador Kassiano J. Matteussi sobre gerência de memória baseado no controle de ingestão de dados para aplicações de Spark Streaming, cujo contexto será melhor explicada na sub-seção seguinte. A tese também conta com um ambiente propriamente configurado na Grid5000[1] e aplicações para testes, que serão utilizados na seção de avaliação. Além disso, nos trabalhos avaliados, foram encontradas poucas soluções voltadas a Stream Processing ou gerenciamento de memória em especial, tornando o contexto ainda mais interessante. Então, o impacto na performance do processamento pelo modelo implementado é avaliado e comparado a uma solução heurística, destacando os pontos positivos e negativos do uso do método de ML.

4.1 Contexto do Problema

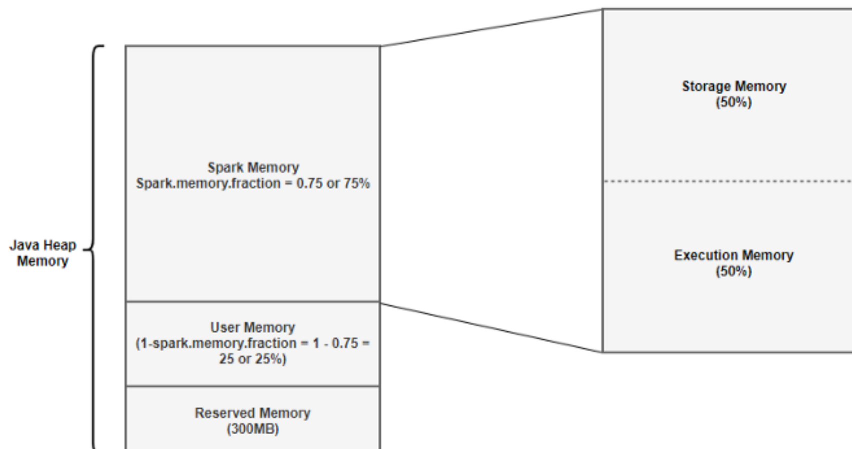
Frameworks de processamento *in-memory*, como Spark, sofrem da falta de gerenciamento dinâmico de recursos no nível de seu gerenciador de memória, especialmente na região de *storage*. Por exemplo, o *Unified Memory Manager* (UMM) pode sofrer com a perda de blocos de dados quando o fluxo de dados é muito elevado. As regiões de armazenamento e computação da UMM ficam cheias, sobrecarregando o processo *Master* do Spark, gerando erros *Out of Memory* (OOM), quedas no sistema e perdas de dados.

A UMM do Spark permite que as regiões de memória alocadas dentro da máquina virtual Java (JVM) sejam dinamicamente ajustadas em percentuais para as áreas de execução e armazenamento, conforme apresentado na Figura 4.1. A memoria de execução tem prioridade sobre a memória de armazenamento, ou seja, em um cenário de processamento intensivo, caso a memória de execução precise de mais espaço, irá solicitar a região de armazenamento que automaticamente ira despejar os dados da memória.

A abordagem adotada na tese do Kassiano Matteussi limita o envio de dados, utilizando um valor global máximo de memória de armazenamento, visando manter o uso consciente da área *Storage Memory*. O procedimento é feito em conjunto a um sistema de

[1] https://www.grid5000.fr/

Figura 4.1: Distribuição de Memória de um Executor Spark



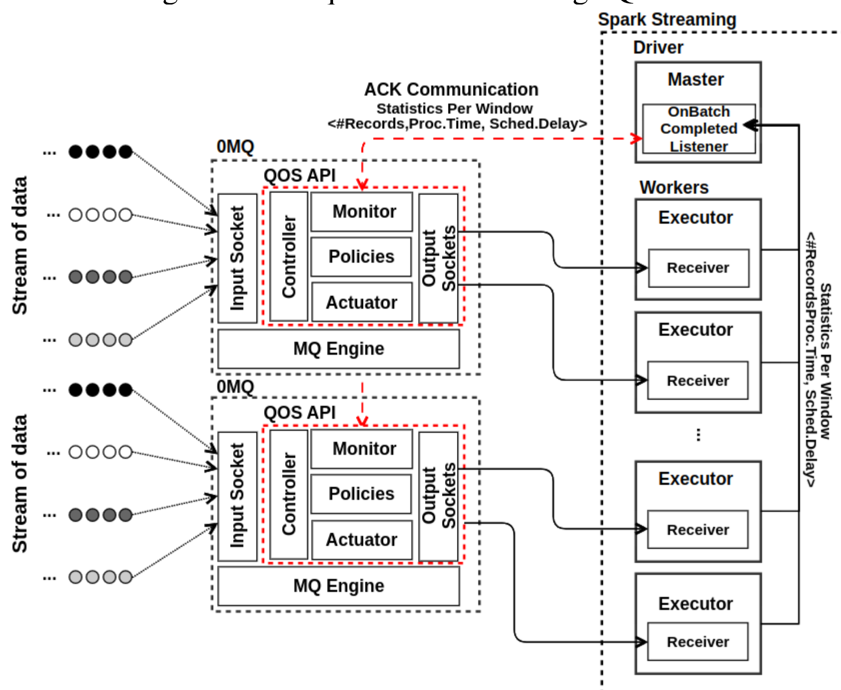
Fonte: Documentação Spark

Message Queues (MQs), no qual as MQs são responsáveis pela orquestração e sincronização dos dados com base em informações recebida do estado do último *batch* processado pelo Spark, como total registros processados, tempo de execução e atrasos. A arquitetura da solução contém dois atores principais: o módulo de Spark Streaming e o sistema de Message Queueing, conforme na Figura 4.2. O primeiro controla o ciclo de vida da execução e fornece, através de *listeners*, as estatísticas recém mencionadas sobre o estado de execução, representado pelo OnBatchCompletedListener na Figura 4.2. Já o segundo controla o escalonamento de dados ao receber as estatísticas sobre a execução do Spark por uma QoS API, a qual mede a carga de trabalho no sistema e encaminha as mensagem ao módulo Spark Streaming em uma taxa mais controlada.

A QoS API possui quatro módulos:

- **Monitor**: recebe periodicamente informações sobre o estado de execução no Spark Processing;
- **Controller**: supervisiona todo o sistema;
- **Policies**: identifica se o valor global máximo de memória de armazenamento definido foi ultrapassado;
- **Actuator**: decide se as mensagens recebidas nas MQs continuam sendo enviadas ao Spark ou não.

Figura 4.2: Arquitetura das Message Queues



Fonte: Os Autores

4.2 Abordagem da Proposta

Portanto, de forma que o melhor desempenho possível seja atingido com o menor uso de memória, assim como evitar erros de falta de memória, perdas de dados, *overheads*, e quedas no sistema, o escopo de implementação deste trabalho consiste de:

1. Uma adaptação do algoritmo SAQN para o ambiente de *Message Queues*;
2. Integração do agente SAQN adaptado no MQ mestre como uma alternativa a outras *policies*;
3. Que executa mais especificamente no módulo *Policies* da API de MQs;
4. Com o objetivo de controlar o limite máximo de utilização da área *Storage Memory* dentro da JVM;
5. Com base nas estatísticas obtidas do módulo *Monitor*.

4.3 Implementação

As implementações neste capítulo foram desenvolvidas majoritariamente em Python utilizando a biblioteca Keras do Tensorflow 2, tomando como base soluções genéricas dos algoritmos e adaptando-as ao contexto do trabalho. Além disso, foi necessário implementar uma API-SAQN[2] para que os Message Queues pudessem executar o código dos agentes, visto que as MQs são feitas em C. A implementação dessa API foi feita através de Cython[3], uma linguagem usada principalmente para facilitar a escrita de extensões em C para Python, pois durante a pesquisa pelas implementações genéricas dos algoritmos de ML, foi encontrado uma maior quantidade de exemplos em Python do que em C/C++, seja DQNs, A3Cs ou LSTMs. Ou seja, o desenvolvimento de APIs em Cython se prova mais flexível para diferentes soluções de ML do que reescrever os códigos em C/C++.

Desse modo, cada agente possui sua própria API que reflete como eles devem se comunicar com o ambiente, porém o processo de criação dessas interfaces não influenciam no código original em Python. Mais detalhes sobre a comunicação dos agentes com o ambiente serão explicados nas seções seguintes.

Tomando como base o algoritmo SAQN proposto em (ZHANG; YAO; GUAN, 2017), foi-se implementado uma versão adaptada para o ambiente Spark e de Message Queues (Alg. 1), tentando-se manter o mais fiel possível ao algoritmo proposto no artigo. As modificações abrangem principalmente as definições de estado, recompensa e ação, que por consequência influenciaram na arquitetura da rede neural e na comunicação do agente com o ambiente. Tais alterações serão explicadas em mais detalhe ao longo desta seção.

4.3.1 Representação do Estado da Execução

Originalmente no artigo, o estado do ambiente era representado pelas configurações de valores como CPU e memória de cada máquina. No caso deste trabalho, foram considerados valores médios provenientes do Spark ou obtidos das instâncias de MQs,

[2]https://github.com/gabrijob/SAQN_mq
[3]https://cython.org/

Algorithm 1 SAQN on MQ

1: // Inicialização de variáves da rede pelo MQ $[create_agent(state_{start})]$
2: Inicializar modelo Stacked-Autoencoder com pesos aleatórios de uma distribuição uniforme (Glorot initializer)
3: Inicializar replay memory
4: $\epsilon = 1;\ episode = 1;\ done = false;\ step = 1$
5: Inicializa ação para 'sem alterações' $action = 0$
6: **while** done is false **do**
7: **for** step < EPISODE_SIZE **do**
8: // Pedido de uma ação pelo MQ $[infer(state_{curr}, done = false)]$
9: // Ou fim da execução $[finish(state_{end}, done = true)]$
10: Calcula vetor de recompensas para o último estado $rewards[2]$ (4.2)
11: Atualiza a replay memory com a última observação
12: Treina a rede neural
13: **if** done is false **then**
14: Diminui ϵ
15: Calcula nova ação a partir do estado atual $action = max(Q - values)$
16: **end if**
17: **end for**
18: $episode = episode + 1$
19: **end while**

sendo representado como:

$$state(t) = [thpt_{tot}, thpt_{var}, t_{proc}, t_{sche}, msgs_{spark}, msgs_{gb}, mem_{ready}, qos_{thresh}] \tag{4.1}$$

Onde $thpt_{tot}$ é o*throughput* médio global da execução no instante t, $thpt_{var}$ é a variação de *throughput* entre t e $t-1$, t_{proc} e t_{sche} são o tempo de processamento médio e tempo de escalonamento médio em cada máquina no tempo t, respectivamente. Em termos de armazenamento, $msgs_{spark}$ e $msgs_{gb}$ são a quantidade de mensagens processadas pelo Spark e enviadas pelo MQ em Gb, mem_{ready} é o estado atual da memória global (quantos dados estão no Spark esperando para serem processados no instante de tempo t), e finalmente, qos_{thresh} representa o estado global da memória em GB dos dados distribuídos entre a memória dos executores do Spark.

4.3.2 Representação da Ação do Agente

Na implementação base do SAQN, uma ação representava uma mudança de uma unidade em um tipo de recurso, por exemplo um acréscimo de 1 Gb de memória ou 1 core de CPU. Decidiu-se manter essa lógica de alterações de uma unidade, porém apenas no tamanho do buffer das MQs, de forma que $action(t) \in (-1, 1)$, com a unidade sendo equivalente a 1 Gb. Então, existem três possíveis ações: acréscimos de 1 Gb na memória global dos MQs, decréscimos de 1 Gb, ou ainda nenhuma alteração.

4.3.3 Representação da Política de Recompensa

Foi mantido a representação da recompensa como um vetor bi-dimensional envolvendo performance e custo, evitando-se uma combinação linear desses valores devido a complexidade de se definir manualmente como pesar corretamente dois objetivos diferentes. No contexto deste trabalho, a performance foi representada como o *throughput* global médio / variação de *throughput* normalizado entre -1 e 1, e o custo, o tamanho do buffer global dos MQs normalizado entre valores mínimos e máximos de memória disponíveis. De forma que:

$$\begin{aligned} &R(t) = [r_perf(t), r_cost(t)], \text{ onde} \\ &r_perf(t) = 2 * (thpt_{tot} - thpt_{min})/(thpt_{max} - thpt_{min}) - 1, \\ &r_cost(t) = -2 * (qos_{thresh} - qos_{min})/(qos_{max} - qos_{min}) - 1 \end{aligned} \tag{4.2}$$

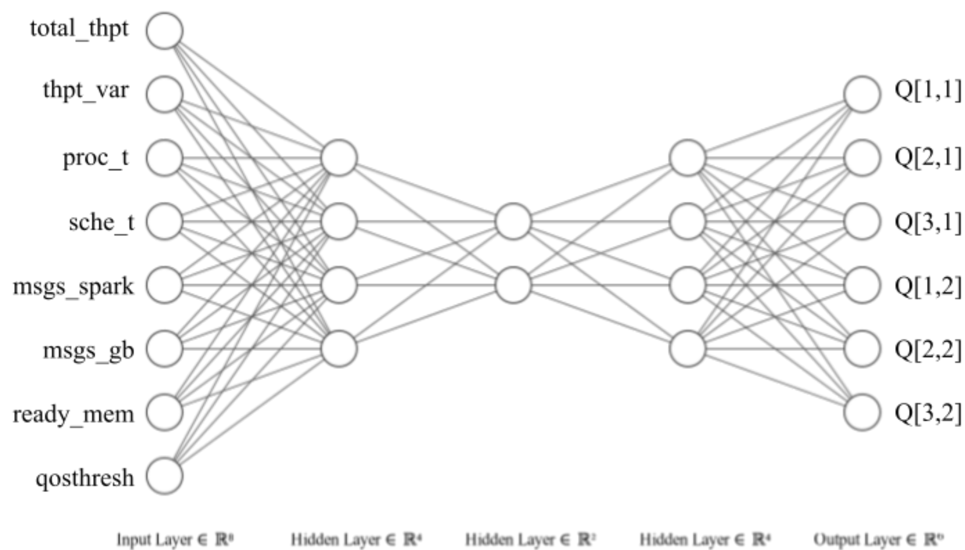
4.3.4 Modelo da Rede Neural

O modelo proposto em (ZHANG; YAO; GUAN, 2017) específica apenas o uso de um *Stacked Autoencoder* como a rede profunda do algoritmo clássico de DQN, com o uso de *replay memory* para lidar com a natureza sequencial das entradas, e a recompensa bi-dimensional que influencia no treino da rede. Porém, o artigo não indica a arquitetura do SAQN originalmente.

Seguindo as diretrizes do artigo base de se usar um *Stacked-Autoencoder*, foi implementado uma rede com camadas **8-4-2-4-6** como demonstrado na Figura 4.3. As ca-

madas externas foram decididas com base no tamanho de entrada, definida por state(t) acima, e o tamanho de saída, por Q-value, que possui uma dimensão de duas vezes a quantidade de ações possíveis (bi-dimensionalidade), cujo motivo será explicado no próximo parágrafo. Por fim, as camadas ocultas foram deduzidas para que a rede tomasse a forma básica de um AutoEncoder.

Figura 4.3: Modelo da Rede SAQN



Fonte: Os Autores

Como mencionado, a bi-dimensionalidade do vetor de recompensas impacta outro fatores no modelo, de forma que os Q-values na SAQN também sejam vetores bidimensionais e a função de perda precise acomodar essa mudança. Como nenhum detalhe dessa implementação é dada no artigo base, a solução implementada foi de "planificar" os vetores Q-value, então se originalmente é necessário um Q-value por ação, nesse caso utilizou-se dois Q-values por ação, que também pode-se entender como seis possíveis ações. Dessa maneira, cada Q-value representa uma ação (acréscimo, decréscimo, sem alteração) para um objetivo diferente (performance ou custo), e o maior Q-value indica a melhor ação e por quê, sem alterar o treinamento efetivamente, pois para a rede, é como se houvessem seis ações diferentes, utilizando a função de perda como na Equação 4.3 a

seguir.

$$L = [r_t + \gamma * Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]^2 \tag{4.3}$$

Onde r_t é a recompensa obtida no instante de tempo t, γ é a taxa de aprendizado, e $Q(s_{t+1}, a_{t+1})$ e $Q(s_t, a_t)$ são os Q-values, ou seja expectativas de recompensa, para determinados estado e ação nos instantes de $t + 1$ e t, respectivamente. Então, a função de perda é basicamente um erro médio ao quadrado (*mean squared error*) entre a última expectativa de recompensa Q calculada para (s_t, a_t), e o valor obtido durante a execução $r_t + \gamma * Q(s_{t+1}, a_{t+1})$.

4.3.5 Integração do Agente nas Message Queues

O agente SAQN se encontra instanciado dentro da MQ mestre, o qual possui dados globais do estado corrente da execução e é capaz de alimentar o agente com o formato de entrada definido logo acima. Como já explicado na seção de contexto, ele foi inserido no módulo policies, fazendo parte dos possíveis algoritmos a serem utilizados para redefinir o limite de memória.

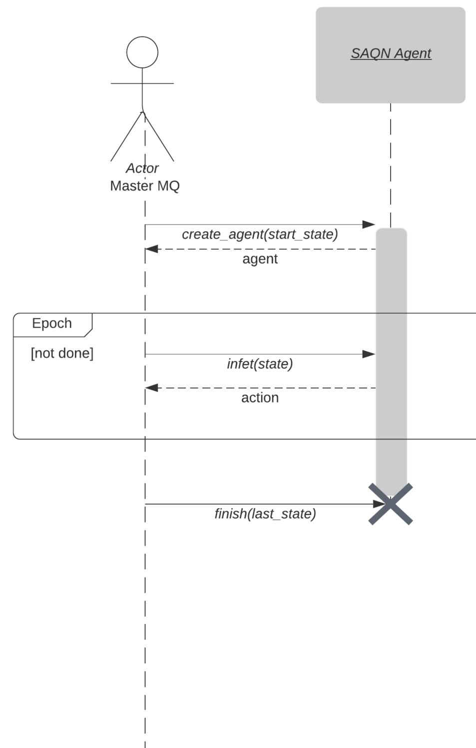
Diferente de implementações mais típicas de DQNs, como por exemplo em cima de jogos de Atari[4], que exibem um comportamento mais “ativo”[5], ou seja, o agente pede o estado atual para o ambiente, geralmente uma imagem da tela do jogo, e após decidir a próxima ação, ele faz um pedido de um novo estado. Já neste trabalho, foi necessário uma abordagem mais “passiva”, de modo que o MQ mestre pudesse enviar ao agente o estado global da execução e receber uma decisão em como o limite de memória deve ser alterado, visto que o mestre necessita de tais ações em momentos exatos do seu fluxo de execução, não podendo ser interrompido pelo agente para envio de estado ou recebimento de decisão. Dessa forma, a comunicação entre as duas partes ocorre conforme representado na Figura 4.4.

Como demonstrado na Figura 4.4, a API do agente SAQN possui três funções principais: *create_agent*, *infer* e *finish*. Onde a primeira é responsável por inicializar as variáveis do agente para a execução, a segunda função é utilizada para pedir que a rede infira uma nova ação com base no estado atual, conforme especificados nas subseções

[4]https://gym.openai.com/

[5]Notar que ativo não refere-se às técnicas de aprendizado por reforço ativa e passiva, como apresentadas em (NORVIG, 2020), apenas à natureza da comunicação entre o agente e a MQ.

Figura 4.4: Diagrama de Sequência do Agente SAQN



Fonte: Os Autores

4.3.1 e 4.3.2. Finalmente, o terceiro método finaliza o agente e salva o modelo final e dados da execução. Além disso, vale notar que cada *infer*, que é chamada mais vezes, pode levar entre 0,03s e 0,55s, dependendo se forem efetuados geração de logs e salvamento do modelo em tempos intermediários, não se tornando um valor considerável de *overhead*.

5 AVALIAÇÃO DA PROPOSTA

A avaliação foi realizada em várias etapas, com o objetivo de validar o aprendizado do SAQN através dos valores de recompensa obtidos, comparar com outras soluções como *baselines* e analisar se é possível aumentar a performance através do uso de múltiplas *threads*. Os testes foram executados em dois *clusters* diferentes da Grid5000, indicados na Tabela 5.1.

Tabela 5.1: Clusters Usados

Nome	*CPU*	*Cores*	*RAM*	*Memória*	*Ethernet*
E1	2x CPUs Intel Xeon E5-2630 v3	8 por CPU	128GB	2x 558GB HDD	10Gb
E2	2x Intel Xeon Gold 6130	16 por CPU	192GB	223GB SSD + 447GB SSD + 3726GB HDD	10Gb

Fonte: Os Autores

Além dos *clusters*, também foram utilizadas diferentes configurações do ambiente de execução, conforme tabela 5.2.

Tabela 5.2: Configurações de Ambiente

Nome	*Gerador de Dados*	*Message Queue*	*Master*	*Worker*
C1	8 nós	1 nó	1 nó	8 nós
C2	8 nós	4 nó	1 nó	8 nós
C3	8 nós	8 nó	1 nó	8 nós

Fonte: Os Autores

Para a preparação do ambiente de execução foram utilizados os *softwares* e configurações dos parâmetros do Spark conforme Tabela 5.3.

Tabela 5.3: Configurações de software do ambiente

Operating System	Debian 4.9.0-11 amd64
Hadoop	3.1.2
Java	1.8.081
Scala	2.13
OpenMpi	4.0.1
0MQ	3.1.1
Tensorflow	2.3.1
Keras	2.4.3
Cython	0.29
Apache Spark	2.4.3
Window (batch interval)	2000ms
Block interval	400ms
#Driver instances	1
#Executors instances	8
Driver's JVM Memory (E1)	117GB
Driver's JVM Memory (E2)	174GB
Executors' JVM Memory (E1)	117GB
Executors' JVM Memory (E2)	174GB
Cores per Executors (E1)	32
Cores per Executors (E2)	64
Parallelism	default

A seguir, são listadas as aplicações *streaming* utilizadas nas avaliações.

- Stateless SUMServer: aplicação não intrusiva em nível de memória, recebe os dados enviados pelo MQ, desserializa as mensagens numéricas de 10000KB e as imprime;
- SUMServer: aplicação minimamente intrusiva em nível de memória, recebe os dados enviados pelo MQ, desserializa as mensagens numéricas de 10000KB, calcula a média entre o estado atual sobre os últimos 10 estados retidos em memória, se e somente se os estados possuam algum índice relacionado ao estado atual. Em seguida, os valores são impressos;
- Global SUMServer: aplicação intrusiva em nível de memória, recebe os dados enviados pelo MQ, desserializa as mensagens numéricas de 10000KB, calcula a media entre o estado atual sobre os últimos 10 estados retidos em memória para o executor

atual e todos os seus vizinhos, caso os estados possuam algum índice relacionado ao estado atual. Em seguida, os valores são impressos.

Neste capítulo são realizadas três avaliações principais: uma primeira entre diferentes modelos de implementação do SAQN para encontrar o mais representativo para o contexto, em seguida uma comparação do modelo escolhido com *baselines* para analisar quais são as vantagens com relação a performance e custo, e por fim uma análise dos *speedups* obtidos ao se usar múltiplas *threads* Spark durante a execução.

5.1 Análise da Função de Recompensa

Inicialmente, com o objetivo de decidir o modelo mais representativo do algoritmo SAQN, foi necessário analisar diferentes funções de recompensa e como elas se comportam para redes com ou sem treino prévio à execução, no qual o pré-treino é definido como o modelo resultante de uma execução não treinada. Foram realizados testes sem treino de todas as variações presentes na Tabela 5.4 para aplicação **SUMServer** com a configuração **C3**, cujos resultados estão representados na Figura 5.1.

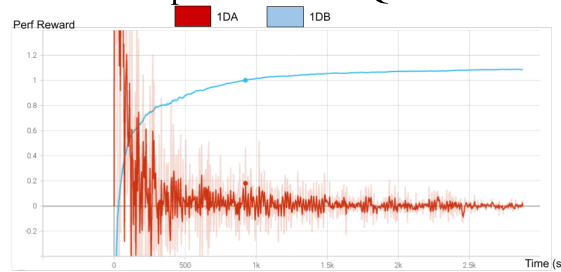
Tabela 5.4: Variações do modelo SAQN

Nome	*Tipo de Recompensa*	*Valores da Recompensa*
1DA	Vetor de uma dimensão: **performance**.	Variação de *throughput* entre t e $t-1$.
1DB	Vetor de uma dimensão: **performance**.	*Throughput* normalizado em t.
2DA	Vetor de duas dimensões: **performance e custo**.	Variação de *throughput* entre t e $t-1$, e utilização de memória normalizada em t.
2DB	Vetor de duas dimensões: **performance e custo**.	*Throughput* e utilização de memória em t normalizados.

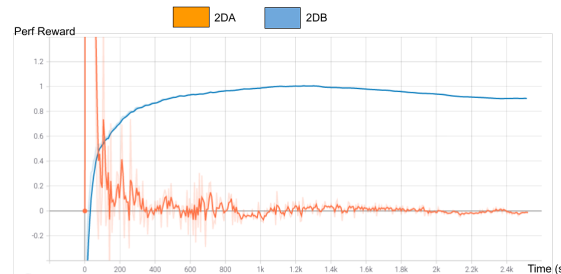
Fonte: Os Autores

Em relação aos hiper-parâmetros da rede, durante o treinamento utilizou-se o otimizador *Adam* para controlar a taxa de aprendizado γ e o algoritmo ϵ-*greedy* para balancear a taxa de exploração ϵ do agente, de forma que ele varie entre ações aleatórias e conscientes. O valor de ϵ começa num valor de 1 e decresce em 1,25% a cada 5 *steps* (chamadas da função *infer* na API), diminuindo as chances do agente tomar ações aleatórias para exploração. Vale notar que o valor percentual de decrescimento de ϵ influencia na velocidade de convergência do modelo, porém o mal controle desse valor pode gerar problemas como *overfitting*, convergência em máximos locais apenas ou não-convergência em um modelo final.

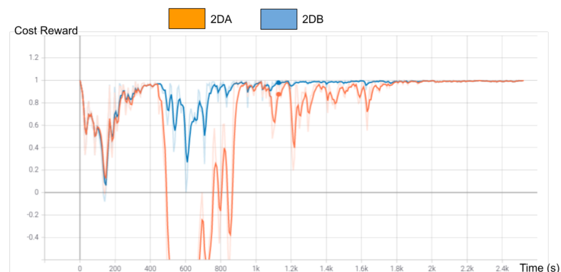
Figura 5.1: Recompensas com SAQN sem treino prévio



(a) Performance para 1DA e 1DB sem treino



(b) Performance para 2DA e 2DB sem treino



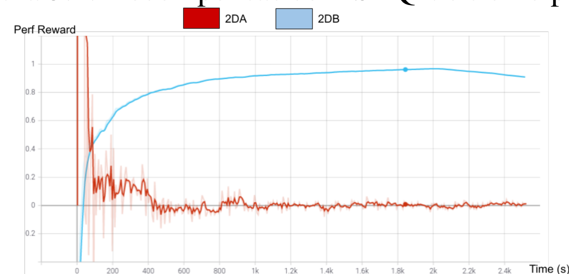
(c) Custo para 2DA e 2DB sem treino

Fonte: Os Autores

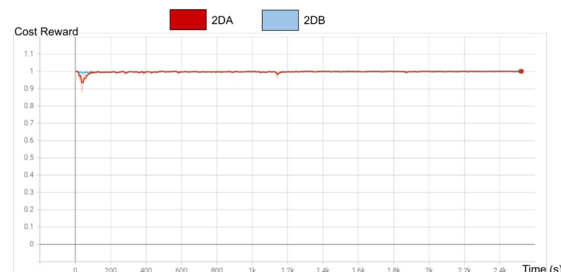
Como pode-se analisar nos dois gráficos de performance na Figura 5.1 , nos casos 1DA (Fig. 5.1(a) em vermelho) e 2DA (Fig. 5.1(b) em laranja), onde é considerado a variação de *throughput* como recompensa, ela cresce bastante no início da execução e depois fica variando em torno de zero, indicando que o *throughput* aumenta rapidamente quando os dados começam a chegar nas MQs, e estabiliza em um valor durante o resto da execução. Nesse caso o comportamento ideal seria uma recompensa sempre igual ou maior que zero, sinalizando que não houveram variações negativas, ou seja, quedas de *throughput*, em nenhum momento. Vale notar que os picos positivos no início dos gráficos da Figuras 5.1(a) e 5.1(b) representam o aumento quase instantâneo de vasão quando os dados começar a chegar pela primeira vez, de forma que esse valor máximo não está representado pois ele impossibilitaria o foco nos intervalos de recompensas em $[-0.2, 1]$, onde ocorrem as variações mais importantes nesses gráficos. Outro ponto importante nesses dois gráficos são os casos 1DB (Fig. 5.1(a) em azul) e 2DB (Fig. 5.1(b) em azul),

que utilizam o *throughput* normalizado como recompensa, que cresce continuamente até atingir um valor máximo de aproximadamente 996 MBps, e no caso específico de 2DB ocorre uma perda de performance ao fim da execução, causado pelo uso mínimo de memória global gerenciado pelo agente que aprendeu a reduzir os custos. Já no gráfico de custo na Figura 5.1(c), as recompensas são notavelmente mais instáveis no começo até estabilizarem no valor máximo, ou seja, o menor nível de utilização de memória global entre *Message Queues*. Análogo ao caso dos gráficos de performance, o pico negativo na Figura 5.1(c) não está totalmente representado pois impediria a visualização das variações menores.

Figura 5.2: Recompensas com SAQN e treino prévio



(a) Performance para 2DA e 2DB com treino

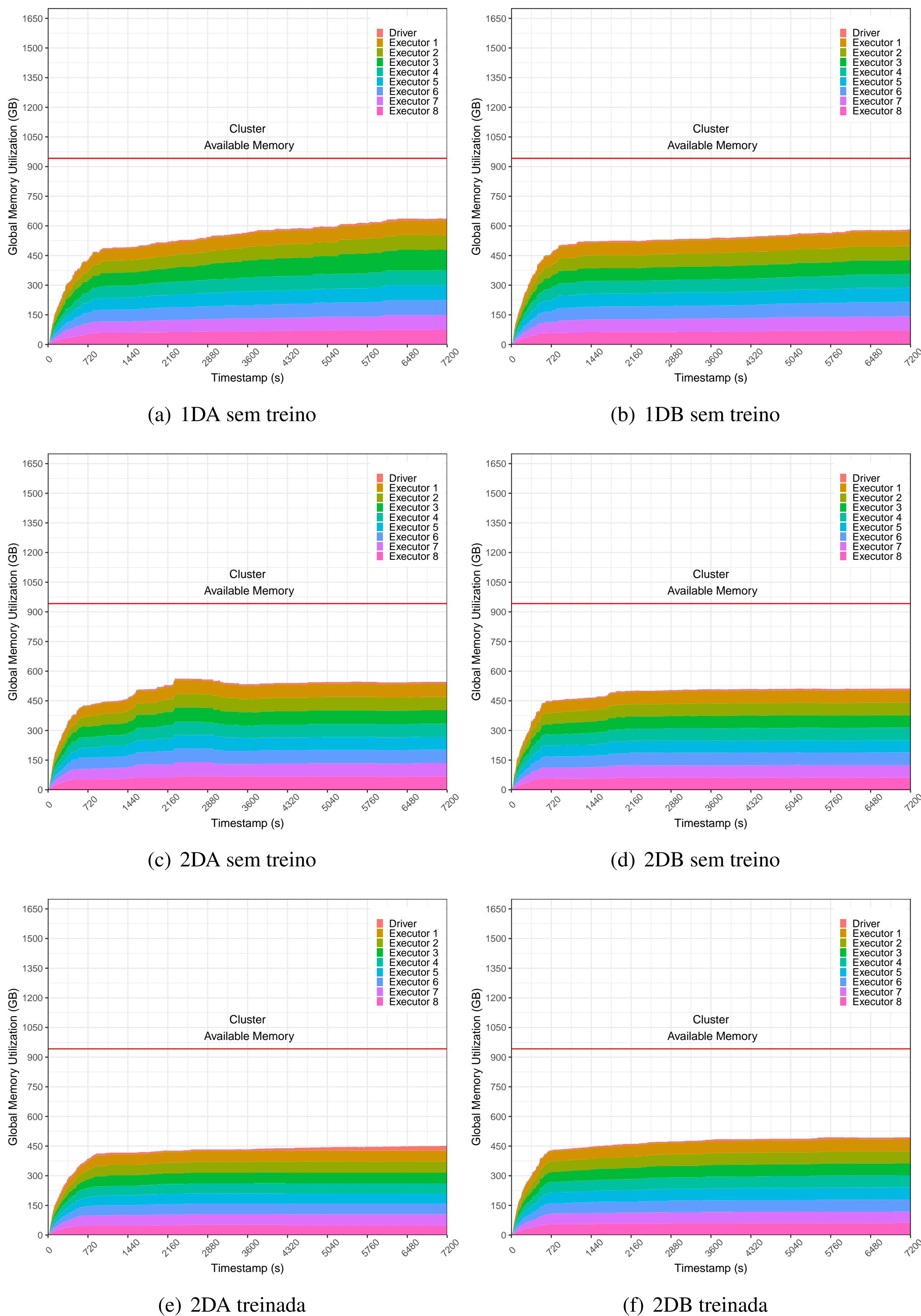


(b) Custo para 1DA e 1DB com treino

Fonte: Os Autores

Com base nos resultados anteriores, foi feito um experimento mantendo-se a configuração anterior **C3** e apenas os modelos **2DA** e **2DB** com treinamento prévio, visto que o controle do nível de utilização da memória é uma característica essencial do funcionamento desejado da solução, além de continuar mais fiel a implementação original do SAQN com recompensas de vetores de duas dimensões. Ao observar os gráficos de performance na Figura 5.2(a), pode-se notar um comportamento parecido das recompensas de performance com os testes sem treinamento, tanto para 2DA (em vermelho) quanto 2DB (em azul), porém agora as variações são bem menores. Além disso, o gráfico de custo em 5.2(b) mostrou um resultado consideravelmente diferente, com a recompensa se mantendo estavelmente no valor máximo, ou seja uso mínimo de memória global de aproximadamente 1 Gb, para os dois modelos.

Figura 5.3: Utilização Global de Memória para Aplicação SUMServer sem Backpressure



(a) 1DA sem treino

(b) 1DB sem treino

(c) 2DA sem treino

(d) 2DB sem treino

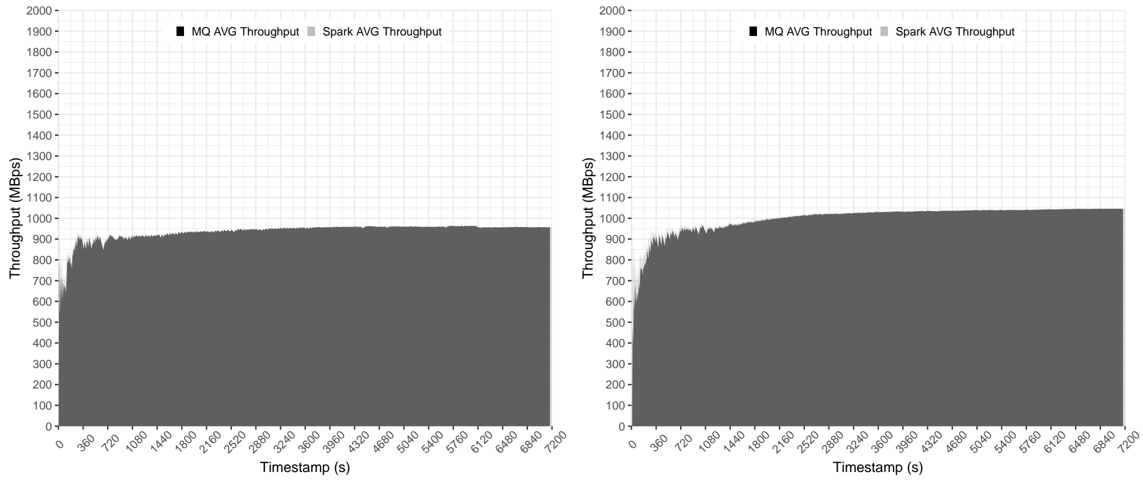
(e) 2DA treinada

(f) 2DB treinada

Fonte: Os Autores

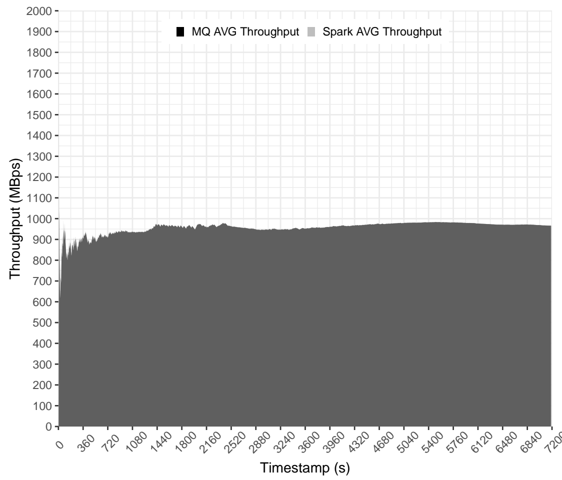
Ao analisar diretamente os dados de execução, é possível perceber que a utilização de memória global e ao longo dos nós diminui gradativamente conforme é utilizado a

Figura 5.4: Throughput Global para Aplicação SUMServer sem Backpressure

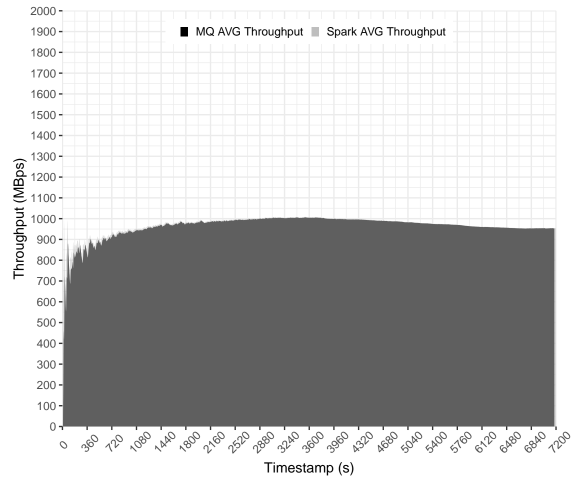


(a) 1DA sem treino

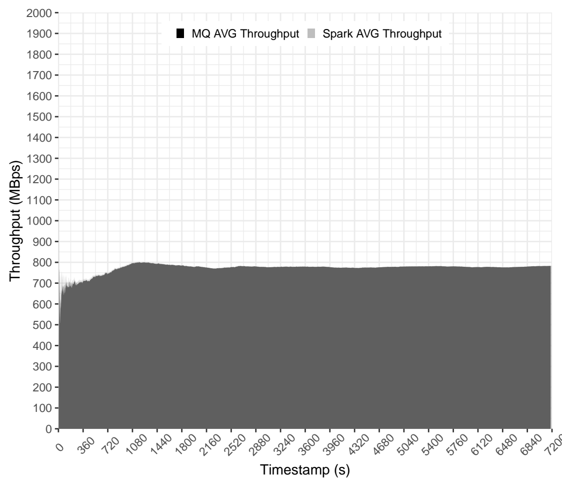
(b) 1DB sem treino



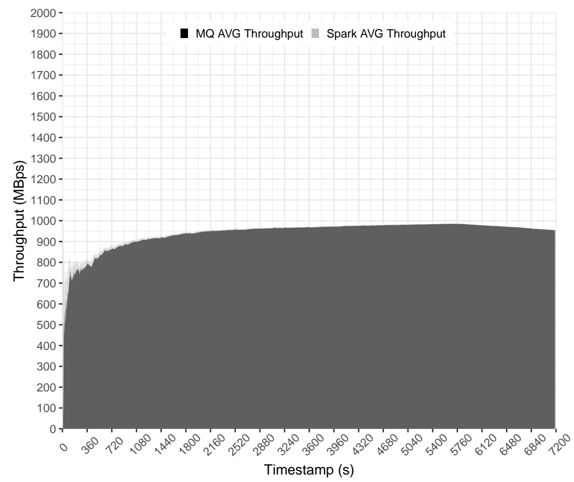
(c) 2DA sem treino



(d) 2DB sem treino



(e) 2DA treinada

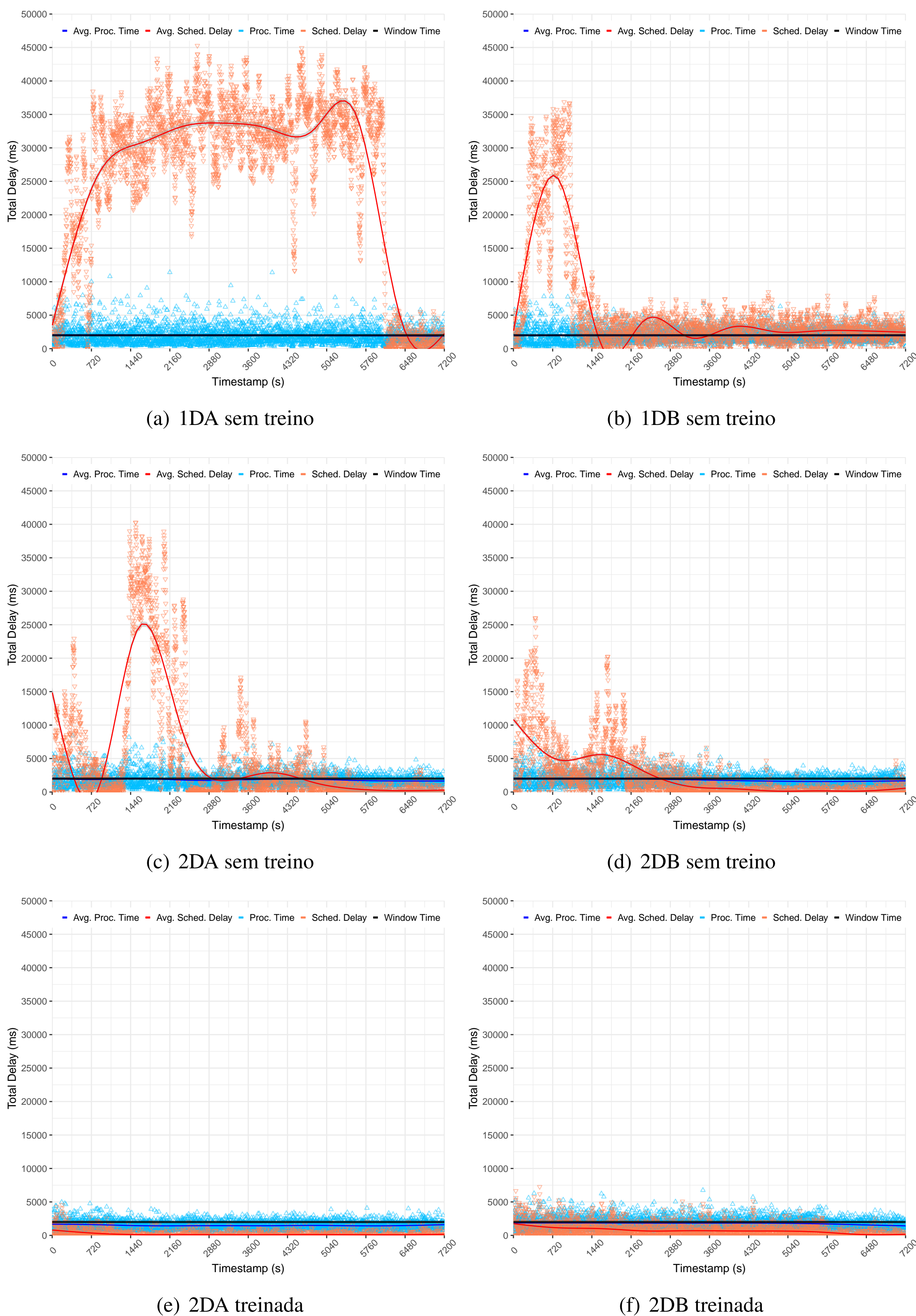


(f) 2DB treinada

Fonte: Os Autores

recompensa como vetores 2D e considerado uma rede treinada, como visto na Figura 5.3. Porém, o valor de *throughput* médio não varia tanto (em torno de 900MBps) entre

Figura 5.5: Delay Total para Aplicação SUMServer sem Backpressure



(a) 1DA sem treino

(b) 1DB sem treino

(c) 2DA sem treino

(d) 2DB sem treino

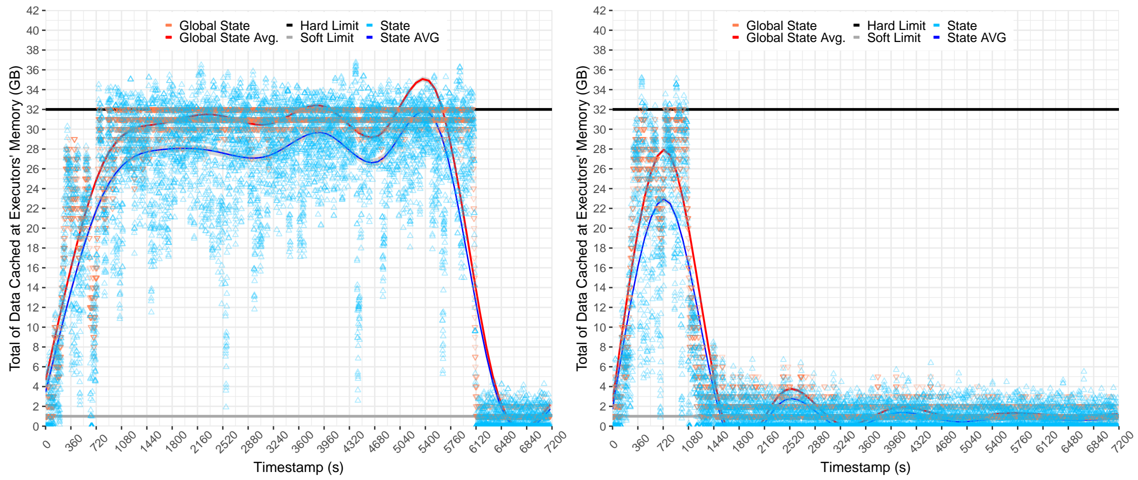
(e) 2DA treinada

(f) 2DB treinada

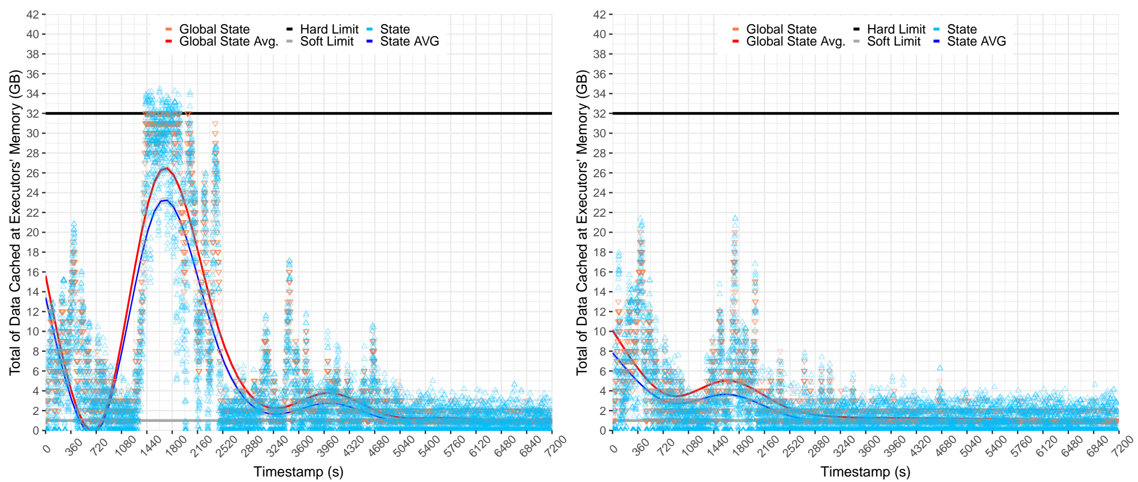
Fonte: Os Autores

modelos (Fig. 5.4), exceto para o 2DA com treino (Fig. 5.4(e)), no qual se atingiu um *throughput* bem menor (800MBps). Já nas Figuras 5.5 e 5.6, fica evidente a relação entre

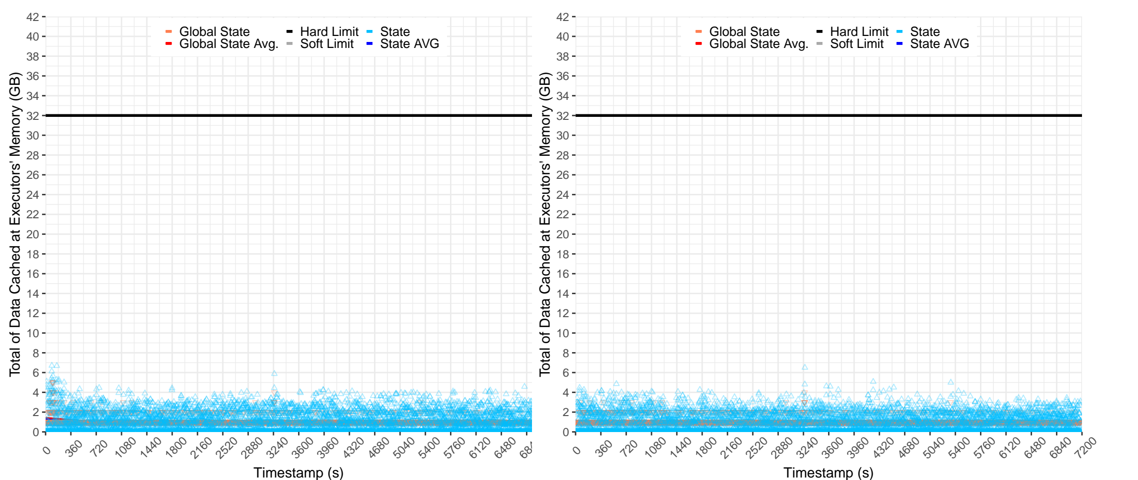
Figura 5.6: Cache Global para Aplicação SUMServer sem Backpressure



(a) 1DA sem treino (b) 1DB sem treino



(c) 2DA sem treino (d) 2DB sem treino



(e) 2DA treinada (f) 2DB treinada

Fonte: Os Autores

o nível de utilização da memória cache e o *delay* total para escalonar e executar uma tarefa, de forma que os modelos treinados já começam com a mínima utilização de cache,

e portanto com o menor *delay*. Outra conclusão que pode ser tirada das Figuras 5.6(a) e 5.6(b) é que mesmo sem considerar diretamente o custo na sua função de recompensa, eles ainda acabam diminuindo o tamanho da cache eventualmente.

Como ambos os modelos 2DA e 2DB obtiveram resultados igualmente positivos, qualquer um dos dois poderia ser usado nos próximos testes presentes neste capítulo, porém optou-se pelo 2DA, já que naturalmente ela prioriza mais as variações de *throughput* do que o uso de cache, pois sua recompensa de performance não é normalizada e pode ultrapassar os valores da recompensa de custo em casos chave quando o *throughput* cresce ou decresce rapidamente, portanto concedendo uma nuância desejada da solução e que demonstrou melhores resultados em outros testes.

5.2 Comparação com Baselines

Nesta sub-seção serão apresentadas comparações entre o modelo escolhido anteriormente com dois casos *baselines*, avaliando principalmente a performance através das medidas de *throughput*, atraso de escalonamento, além da utilização de memória global entre os nós. Os *baselines* utilizados foram:

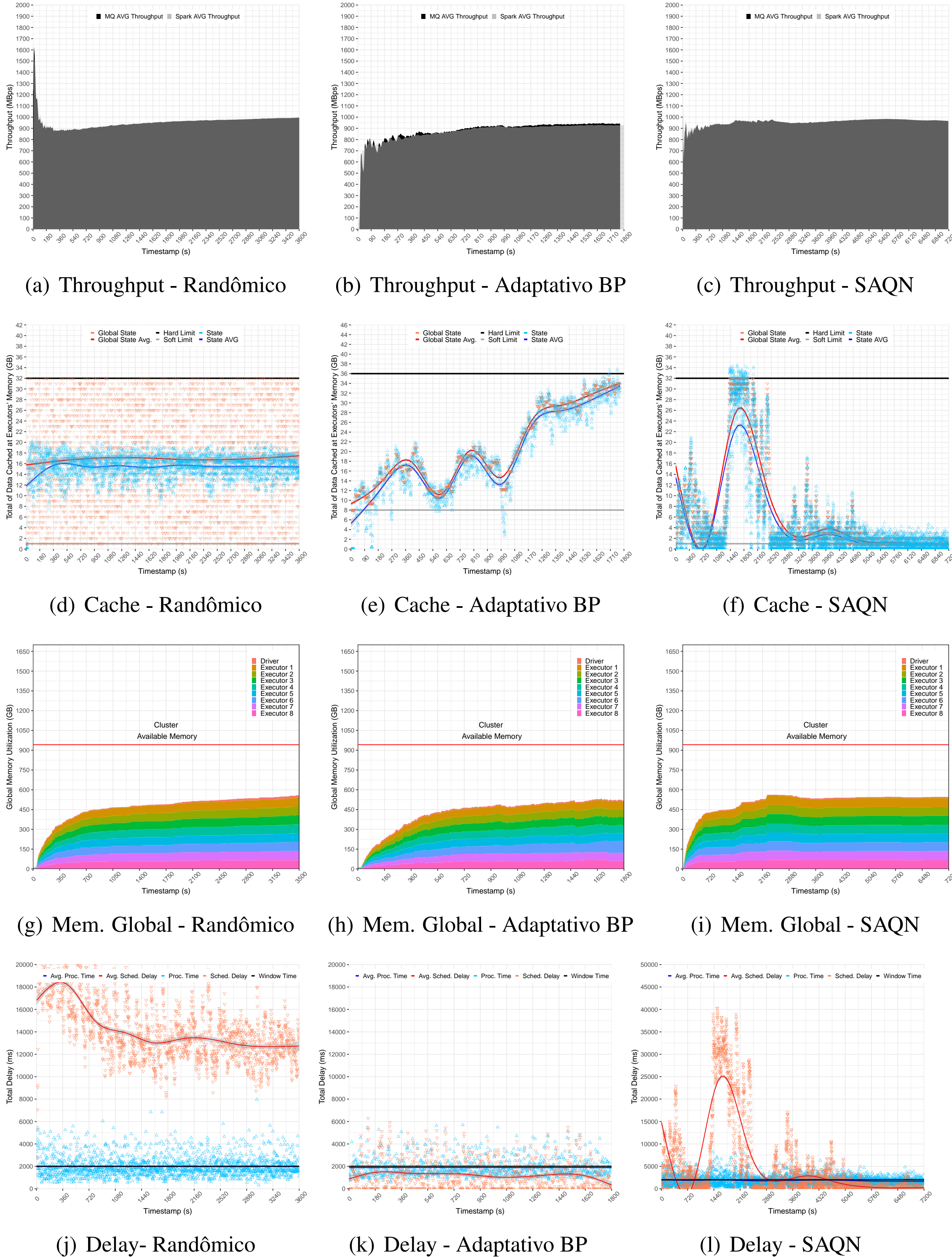
- **Randômico:** solução sem *backpressure* do Spark que altera o valor de cache global aleatoriamente entre valores de mínimo e máximo em Gb pré-definidos, da forma $qos_{thresh} \in (1, 32)$;
- **Adaptativo BP:** solução[1] com *backpressure* do Spark ativado que controla o número de mensagens processadas por segundo em conjunto de um algoritmo heurístico que gerencia dinamicamente a ingestão de dados em memória. O cache global varia entre um limite mínimo e máximo em Gb pré-definidos, da forma $qos_{thresh} \in (8, 32)$;

Na Figura 5.7 pode-se observar em detalhes as diferenças entre as implementações, onde o método randômico varia a cache em todo o intervalo entre 1 e 32 Gb (Fig. 5.7(d)), deixando a memória utilizada na média desses valores, visível na Figura 5.7(g) . Já para os métodos adaptativo e SAQN, se percebe um controle maior da cache, com o primeiro (Fig. 5.7(e)) aumentando o uso de memória ao longo da execução uma vez que o armazenamento não interfere no*throughput* da aplicação, ainda esse valor flutua conforme o desempenho da aplicação. Já o segundo (Fig. 5.7(f)), apresenta alto uso de cache

[1] Solução desenvolvida pelo coorientador Kassiano J. Matteussi para sua tese de doutorado.

inicialmente, para depois cair até o mínimo de 1 Gb, mostrando o aprendizado da rede. Além disso, por conta do melhor gerenciamento da cache, o SAQN soluciona o problema de crescimento contínuo do uso de memória global, estabilizando uma vez que a cache atinge o valor mínimo (Fig. 5.7(i)), enquanto nos métodos randômico e adaptativo, o uso de memória nunca estabiliza, Figuras 5.7(g) e 5.7(h), respectivamente.

Figura 5.7: Comparação com Baselines em C3 para SUMServer



(a) Throughput - Randômico

(b) Throughput - Adaptativo BP

(c) Throughput - SAQN

(d) Cache - Randômico

(e) Cache - Adaptativo BP

(f) Cache - SAQN

(g) Mem. Global - Randômico

(h) Mem. Global - Adaptativo BP

(i) Mem. Global - SAQN

(j) Delay- Randômico

(k) Delay - Adaptativo BP

(l) Delay - SAQN

Fonte: Os Autores

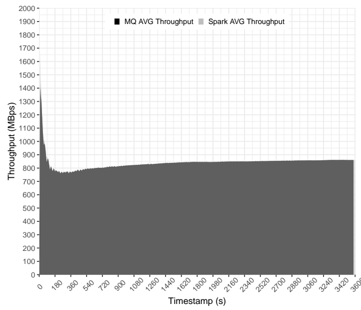
Com relação ao *delay* total, vale notar que para o caso randômico ele mantém em um valor alto durante toda a execução e que isso pode levar a problemas de performance ao longo do tempo (Fig. 5.7(j)), e para o SAQN (Fig. 5.7(l)) o atraso demonstra um comportamento parecido com o gráfico de cache, crescendo no início e depois decaindo, enquanto o adaptativo se mantém com praticamente sem atrasos durante todo o tempo (Fig. 5.7(k)). O baixo nível de *delay* remete a uma execução saudável da aplicação e ajuda a manter a alta performance. Por último, o crescimento de *throughput* se manteve constante em todos os métodos testados (Figs. 5.7(a), 5.7(b) e 5.9(c)), demonstrando que nenhuma das gerências causou perdas notáveis, com o randômico atingindo em média 992MBps, o adaptativo em 934.41MBps, e o SAQN em 966.45MBps.

Quando as soluções são comparadas em uma aplicação mais pesada como a Global SUM Server na Figura 5.8, os resultados não são tão unilaterais, de modo que o SAQN apresenta dificuldades em manter o uso de memória baixo (Figs. 5.8(f) e 5.8(i)) e consequentemente aumenta o atraso de escalonamento (Fig. 5.8(l)), visto que o ganho de *throughput* foi priorizado para alcançar um ganho considerável (Tabela 5.9). Porém, esse tipo de comportamento reflete que o algoritmo pode estar decrescendo a taxa de exploração muito rápido para esta aplicação, de forma que ele não tem tempo de obter observações o suficiente em níveis mais baixos de cache, e consequentemente "se desencorajar"com as altas variações de *throughput* da aplicação. Além disso, vale notar que o impacto do sistema de *backpressure* fica mais evidente nos gráficos de *delay* (Fig. 5.8(k)) e uso de memória global (Fig. 5.8(h)) da solução adaptativa, controlando o aumento desses valores se comparados as outras soluções.

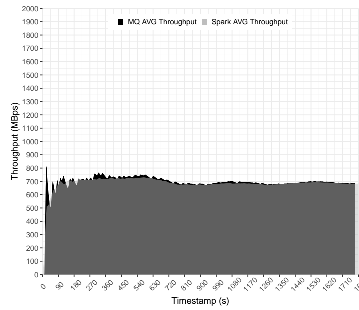
Para a aplicação Stateless SUM Server (Fig. 5.9) que é naturalmente mais leve, todas as aplicações apresentam performances satisfatórias, visto que o Spark consegue processar rapidamente todas as tarefas enviadas. Entretanto, pode-se observar que o atraso da solução randômica (Fig. 5.9(j)) se mantém constantemente elevado durante toda a execução, mostrando que a média entre os valores de cache (Fig. 5.9(d)) são exagerados para uma aplicação como esta, onde valores menores são o suficiente, como vistos nas Figuras 5.9(e) e 5.7(f).

Também foram feitas comparações mais abrangentes entre o algoritmo adaptativo com *backpressure* e o SAQN para todas configurações, *clusters* e aplicações. Na Tabela 5.5 estão os dados referentes a execuções para C1, pode-se notar que a solução adaptativa obtém um *throughput* maior que o SAQN em todas aplicações e clusters, indicando que o SAQN apresentou dificuldades para aprender em um ambiente com apenas 1 MQ, onde

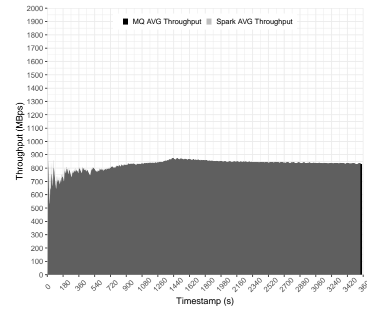
Figura 5.8: Comparação com Baselines em C3 para Global SUMServer



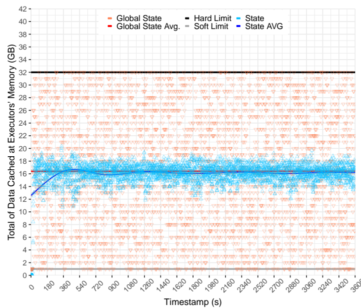
(a) Throughput - Randômico



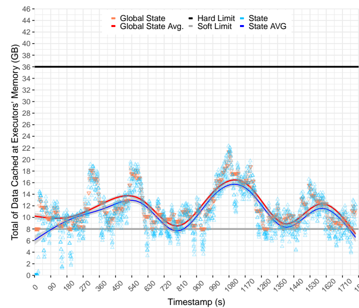
(b) Throughput - Adaptativo BP



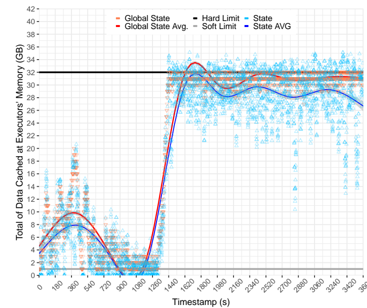
(c) Throughput - SAQN



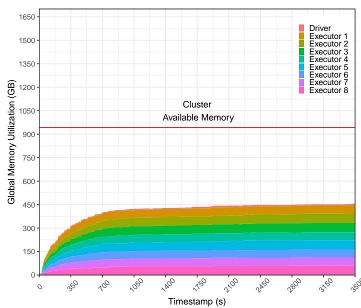
(d) Cache - Randômico



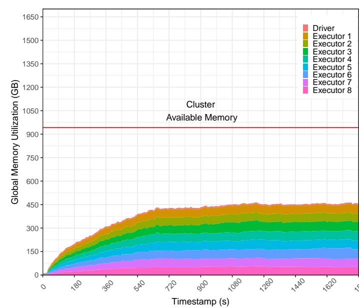
(e) Cache - Adaptativo BP



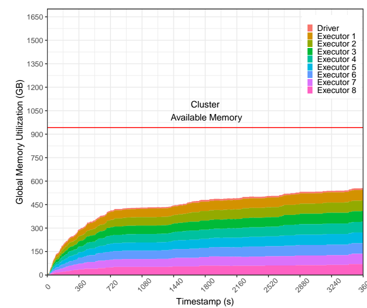
(f) Cache - SAQN



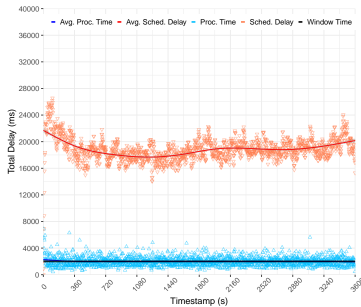
(g) Mem. Global - Randômico



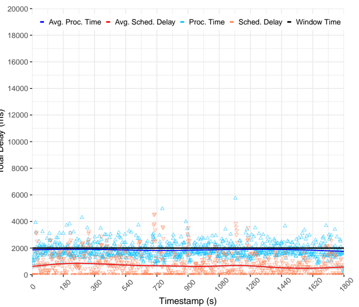
(h) Mem. Global - Adaptativo BP



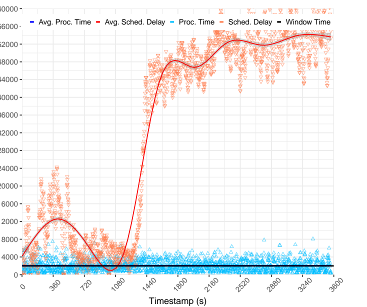
(i) Mem. Global - SAQN



(j) Delay- Randômico



(k) Delay - Adaptativo BP

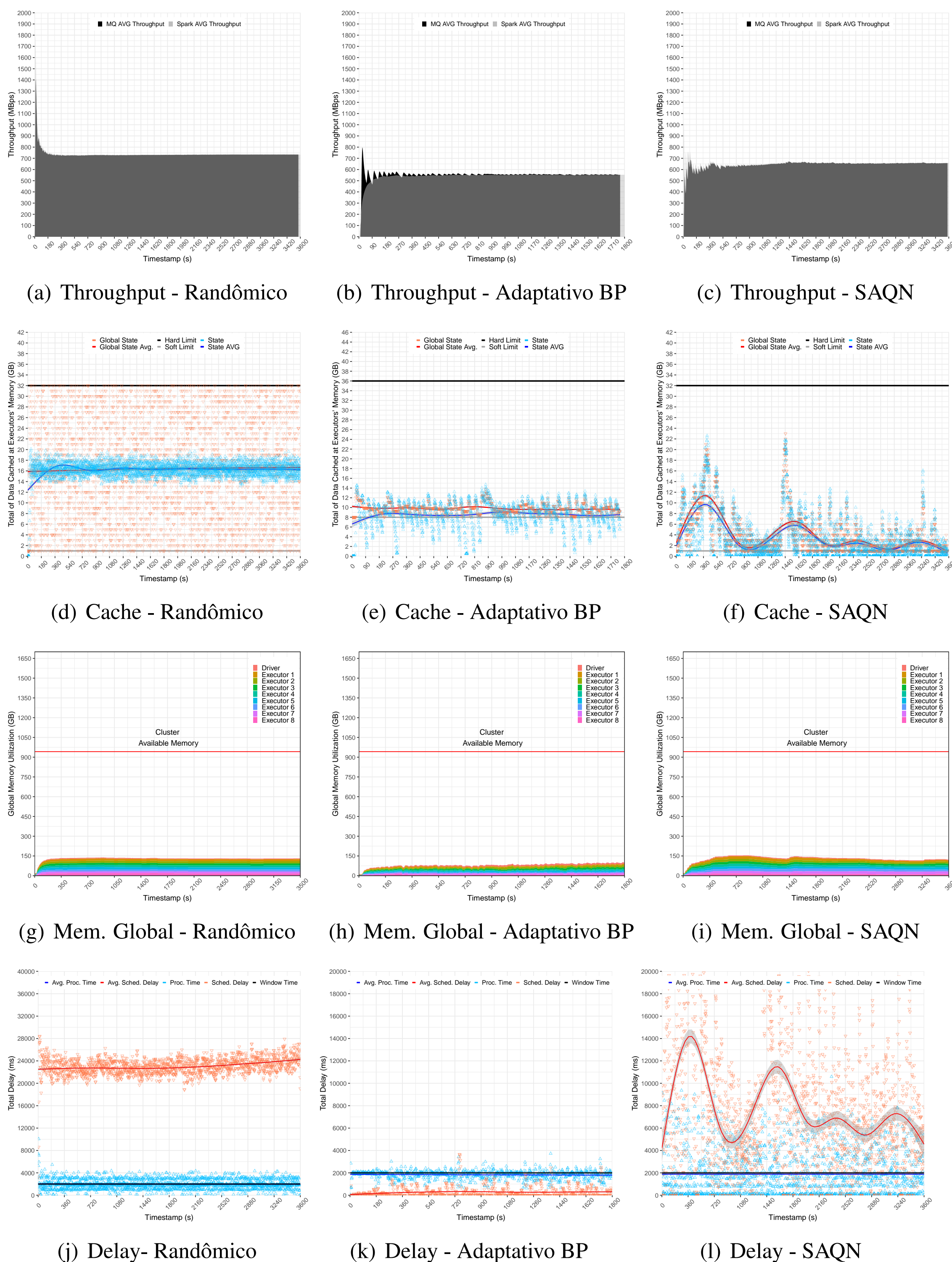


(l) Delay - SAQN

Fonte: Os Autores

existe apenas uma cache gerenciada e a ingestão de dados é baixa. Além disso, os tempos de processamento do SAQN também ficaram menores do que os do adaptativo, que idealmente deveriam estar mais próximos do tempo de janela de 2000ms, para garantir que o Spark esteja continuamente processando.

Figura 5.9: Comparação com Baselines em C3 para Stateless SUMServer



(a) Throughput - Randômico (b) Throughput - Adaptativo BP (c) Throughput - SAQN

(d) Cache - Randômico (e) Cache - Adaptativo BP (f) Cache - SAQN

(g) Mem. Global - Randômico (h) Mem. Global - Adaptativo BP (i) Mem. Global - SAQN

(j) Delay- Randômico (k) Delay - Adaptativo BP (l) Delay - SAQN

Fonte: Os Autores

Para a configuração C2 (Tabela 5.6), o SAQN obteve melhores resultados de *throughput* e tempo de processamento que o algoritmo adaptativo, apesar do *delay* ter aumentado consideravelmente. Porém, com o tempo de processamento mais próximo da janela de 2000ms, o SAQN consegue enviar mais dados continuamente para o Spark,

Tabela 5.5: Resultados de Aplicações para Configuração C1

Algoritmo	*Aplicação*	*Cluster*	*Throughput (MBps)*	*Delay (ms)*	*T. Proc. (ms)*
SAQN	**Stateless SUMServer**	E1	461.16	547	1329
Adaptativo	**Stateless SUMServer**	E1	581.54	80	1984
SAQN	**SUMServer**	E1	664.17	80	1335
Adaptativo	**SUMServer**	E1	730.83	111	1385
SAQN	**Global SUMServer**	E1	630.39	79	1539
Adaptativo	**Global SUMServer**	E1	738.19	87	1589
SAQN	**Stateless SUMServer**	E2	578.51	122	1360
Adaptativo	**Stateless SUMServer**	E2	750.96	68	1981
SAQN	**SUMServer**	E2	534.12	91	1236
Adaptativo	**SUMServer**	E2	790.52	141	1576
SAQN	**Global SUMServer**	E2	719.48	786	1759
Adaptativo	**Global SUMServer**	E2	734.43	383	1881

Fonte: Os Autores

principalmente nos casos onde o *cluster* utilizado é o E2 com maior capacidade de memória e número de núcleos, que pode aguentar uma maior pressão de dados se comparado com o E1.

Tabela 5.6: Resultados de Aplicações para Configuração C2

Algoritmo	*Aplicação*	*Cluster*	*Throughput (MBps)*	*Delay (ms)*	*T. Proc. (ms)*
SAQN	**Stateless SUMServer**	E1	686.14	10142	1990
Adaptativo	**Stateless SUMServer**	E1	573.34	131	1923
SAQN	**SUMServer**	E1	848.51	5804	2004
Adaptativo	**SUMServer**	E1	846.08	885	1903
SAQN	**Global SUMServer**	E1	675.60	45470	2041
Adaptativo	**Global SUMServer**	E1	600.9	476	1896
SAQN	**Stateless SUMServer**	E2	798.94	25034	2037
Adaptativo	**Stateless SUMServer**	E2	755.39	79	1937
SAQN	**SUMServer**	E2	719.89	8513	1999
Adaptativo	**SUMServer**	E2	566.98	475	1887
SAQN	**Global SUMServer**	E2	742.02	10787	2002
Adaptativo	**Global SUMServer**	E2	417.11	509	1885

Fonte: Os Autores

Para a configuração de ambiente C3, com 8 nós de MQ, os resultados disponíveis na Tabela 5.7 mostram que o SAQN conseguiu obter um *throughput* maior e um tempo de processamento mais próximo de 2000ms nas aplicações executadas no cluster E1, se comparado com as performances do método adaptativo, principalmente para a *Global SUMServer*, onde a diferença nessas duas métricas é maior, indicando que o SAQN fornece mais pressão de dados ao Spark enquanto mantém o *throughput* alto, mesmo para uma aplicação mais pesada. Já para as aplicações executadas no cluster E2, o algoritmo SAQN perde mais *throughput* que o adaptativo, porém consegue aproximar mais o tempo

de processamento do tempo de janela. Outro ponto importante de analisar é o *delay* de escalonamento do SAQN, que diminuiu com relação aos resultados na Tabela 5.6, porém continuam elevados.

Tabela 5.7: Resultados de Aplicações para Configuração C3

Algoritmo	*Aplicação*	*Cluster*	*Throughput (MBps)*	*Delay (ms)*	*T. Proc. (ms)*
SAQN	**Stateless SUMServer**	E1	659.05	7848	1876
Adaptativo	**Stateless SUMServer**	E1	555.18	270	1834
SAQN	**SUMServer**	E1	966.45	4993	1829
Adaptativo	**SUMServer**	E1	934.41	1222	1898
SAQN	**Global SUMServer**	E1	865.91	15100	2043
Adaptativo	**Global SUMServer**	E1	691.85	669	1873
SAQN	**Stateless SUMServer**	E2	842.86	12276	1970
Adaptativo	**Stateless SUMServer**	E2	758.95	74	1955
SAQN	**SUMServer**	E2	926.28	11918	2002
Adaptativo	**SUMServer**	E2	1021.12	603	1890
SAQN	**Global SUMServer**	E2	742.02	10787	2002
Adaptativo	**Global SUMServer**	E2	584.37	447	1885

Fonte: Os Autores

5.2.1 Execuções com Múltiplas Threads

Os últimos testes foram executados com o objetivo de avaliar a performance do algoritmo SAQN para 4 *threads* Spark por nó, e como tal caso se compara com os resultados das avaliações anteriores. Porém, como o uso de *threads* ainda não foi explorado no trabalho base sobre MQs, uma análise mais extensa não foi possível. Por consequência, a execução foi feita apenas com configuração **C3** e no *cluster* **E1**, para todas as aplicações.

Conforme os resultados demostrados na Tabela 5.8, pode-se verificar que o ganho de *throughput* foi quase o dobro do observado na Tabela 5.7 para todas as aplicações, além de reduzir consideravelmente o atraso de escalonamento, resolvendo os principais pontos negativos em testes anteriores. Outro ponto a se observar é o aumento do tempo de processamento, que pode sobrecarregar a memória de execução e causar falhas dependendo do tamanho da janela de processamento.

Tabela 5.8: Resultados para SAQN com 4 Threads e configuração C3 e E1

Aplicação	*Throughput (MBps)*	*Delay (ms)*	*T. Proc. (ms)*
Stateless SUMServer	1681.86	3	1353
SUMServer	1542.9	133	3709
Global SUMServer	1777.26	1332	3440

Fonte: Os Autores

Por fim, para aproveitar o ganho considerável de performance fornecido pelo uso de múltiplas *threads* Spark, é preciso um estudo mais aprofundado sobre as configurações necessárias no ambiente para lidar com os riscos de falhas que surgem.

5.3 Discussão

Neste capítulo foi apresentada uma solução baseada em Deep Reinforcement Learning com o objetivo de orquestrar a ingestão de dados em memória do framework Spark juntamente com um sistema de Message Queues. Para a avaliação dessa solução foram feitos testes para verificar o aprendizado da rede através dos valores de recompensa obtidos, para comparar com *baselines* e para analisar a possibilidade da melhorar a performance com o uso de threads.

Os resultado na Tabela 5.9 refletem o ganho de *throughput* obtido pelo método proposto em comparação com uma solução heurística com o sistema de *backpressure* do Spark ativado para diferentes aplicações, de modo que o SAQN apresentou um ganho positivo em todos os casos, chegando até 25%. Além disso, também foi obtido um uso estável e controlado da memória global na maioria dos casos, evitando *crashes*.

Tabela 5.9: Comparação de*throughput* entre Adaptativo BP e SAQN em C3 e E1

Aplicação	*Adaptativo BP*	*SAQN*	*Ganho em Throughput*
Stateless SUMServer	555 MBps	659 MBps	18.7%
SUMServer	934 MBps	966 MBps	6.6%
Global SUMServer	692 MBps	866 MBps	25.1%

Fonte: Os Autores

Já na Tabela 5.10, estão os ganhos de *throughput* para o SAQN ao se usar quatro *threads* Spark com relação a uma execução normal de uma só *thread*, chegando até 155.2% de ganho, além de reduzir drasticamente o atraso de escalonamento, como demonstrado nas seções anteriores.

Tabela 5.10: Comparação de throughput do SAQN com 1 e 4 Threads em C3 e E1

Aplicação	*1 Thread*	*4 Threads*	*Ganho*
Stateless SUMServer	659 MBps	1682 MBps	155.2%
SUMServer	966 MBps	1543 MBps	59.7%
Global SUMServer	866 MBps	1777 MBps	105.2%

Fonte: Os Autores

Concluindo, a solução proposta apresentou melhorias consideráveis no controle de ingestões de dados em memória para aplicações Spark e de maneira mais flexível se

comparada a um método heurístico com *backpressure*, sem sacrificar o ganho de *throughput* nos casos testados, de forma a atingir os objetivos propostos por este trabalho.

6 CONCLUSÃO

Neste trabalho foi realizado um estudo abrangente da literatura sobre algoritmos de ML para gerenciamento de recursos com o objetivo de encontrar a solução mais relevante no contexto de controle de ingestão de dados em memória para aplicações Stream Processing, para então realizar uma análise usando o modelo DQN selecionado em um ambiente real de Spark Streaming, mostrando que o uso de DRL para gerenciamento dinâmico de memória em tal contexto proporciona uma otimização de performance considerável, porém não salva de falhas, visto que houveram contextos de processamento mais lento onde o tempo necessário para a rede aprender um modelo ótimo foi muito grande e impactou demais a execução, ou em outros casos, muito curto e acabou gerando um *overfitting* a um intervalo de tempo específico da aplicação. Ou seja, não sendo flexível o suficiente para *jobs* de longa e curta duração, e *modelfree*, necessitando alterações dos hiper-parâmetros para cada um deles, entretanto a solução com SAQN se mostrou mais flexível que o sistema de *backpressure* nativo do Spark junto de um agente gerenciador dinâmico nativo da MQ, não sofrendo nenhum erro crítico durante as execuções e gerenciando melhor a pressão de dados para o Spark processar, sem sacrificar o *throughput*, enquanto mantém um uso mínimo de memória global. Além disso, a solução implementada também permitiu o uso de múltiplas *threads*, aumentando a performance consideravelmente.

Enfim, a utilização de ML para gerenciamento de recursos apresenta um potencial enorme de avanço que começou a ser mais explorado nos últimos anos, e para uma avaliação mais completa da sua eficiência em ambientes intensivos de Stream Processing serão necessários mais testes com outras implementações de ML mais recentes e poderosas, abrangendo os pontos onde o SAQN não foi o suficiente.

6.1 Considerações Pessoais

Ao longo deste trabalho, tive a oportunidade de executar todas as etapas de desenvolvimento de um trabalho acadêmico com total controle sobre a escolha do tema de pesquisa, sua implementação e sua avaliação, junto das orientações do Cláudio Geyer, do Kassiano Matteussi e do Júlio dos Anjos, que foram cruciais para a execução deste trabalho. Com a liberdade fornecida, houveram dificuldades, erros e decisões a serem tomadas que impactaram no andamento do cronograma proposto inicialmente. Uma delas

começa pelo próprio cronograma, que foi planejado com a ideia de usar como base os códigos utilizados pelos autores dos artigos escolhidos na Seção 3, porém tais códigos não estavam disponíveis e foi necessário implementar soluções mais genéricas dos algoritmos já conhecidas, e adaptá-las para as soluções propostas nos artigos além de adaptar para o contexto deste trabalho, gerando um esforço que deveria ter sido previsto. Outra dificuldade encontrada foi a integração entre os agentes de RL com os Messages Queues, principalmente pelo fato dos MQs serem desenvolvidos em C, e os agentes na biblioteca Keras do Tensorflow, em Python, sendo necessário decidir e implementar uma maneira de realizar a comunicação entre os dois módulos, o que também deveria ter sido previsto no cronograma. E como explicado na Seção 4, foi feita através da criação de uma API usando Cython, sendo uma tarefa tão importante quanto a implementação dos algoritmos.

Finalmente, as dificuldade encontradas neste projeto me proporcionaram um aprendizado muito importante em gerência de tempo e tarefas, além do conhecimento de várias ferramentas, bibliotecas e frameworks, que certamente serão úteis em futuros trabalhos de pesquisa e projetos profissionais.

6.2 Direções Futuras

A continuação mais clara ao trabalho deverá ser a conclusão da implementação dos outros algoritmos em (CHEN et al., 2020) e (CHEN; SAAD; YIN, 2019) para se ter um conhecimento mais completo da performance no gerenciamento da cache dos MQs com um método de aprendizado por reforço mais avançado que o DQN e com um modelo supervisionado por LSTM, respectivamente.

Outros caminhos possíveis incluem avaliações das soluções em tipos diferentes de ambientes com outros *frameworks* de *Stream Processing*, Message Queues, aplicações e redes como Edge, IoT e Fog, validando tanto o método de Message Queues quanto a utilização de *Machine Learning* para o gerenciamento de recursos em contextos mais heterogêneos.

REFERÊNCIAS

AMODEI, D. et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In: PMLR. **International conference on machine learning**. [S.l.], 2016. p. 173–182.

BAEZA-YATES, R.; RIBEIRO-NETO, B. **Recuperação de Informação-: Conceitos e Tecnologia das Máquinas de Busca**. [S.l.]: Bookman Editora, 2013.

BENIFA, J. B.; DEJEY, D. Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment. **Mobile Networks and Applications**, Springer, v. 24, n. 4, p. 1348–1363, 2019.

CHALLITA, U.; DONG, L.; SAAD, W. Proactive resource management for lte in unlicensed spectrum: A deep learning perspective. **IEEE transactions on wireless communications**, IEEE, v. 17, n. 7, p. 4674–4689, 2018.

CHEN, M.; SAAD, W.; YIN, C. Liquid state machine learning for resource and cache management in lte-u unmanned aerial vehicle (uav) networks. **IEEE Transactions on Wireless Communications**, IEEE, v. 18, n. 3, p. 1504–1517, 2019.

CHEN, M. et al. Data correlation-aware resource management in wireless virtual reality (vr): An echo state transfer learning approach. **IEEE Transactions on Communications**, IEEE, v. 67, n. 6, p. 4267–4280, 2019.

CHEN, M. et al. Intelligent resource allocation management for vehicles network: An a3c learning approach. **Computer Communications**, Elsevier, v. 151, p. 485–494, 2020.

CHEN, X. et al. Multi-tenant cross-slice resource orchestration: A deep reinforcement learning approach. **IEEE Journal on Selected Areas in Communications**, IEEE, v. 37, n. 10, p. 2377–2392, 2019.

DUC, T. L. et al. Machine learning methods for reliable resource provisioning in edge-cloud computing: A survey. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 52, n. 5, p. 1–39, 2019.

HOCHREITER, S.; SCHMIDHUBER, J. Long Short-Term Memory. **Neural Computation**, v. 9, n. 8, p. 1735–1780, 11 1997. ISSN 0899-7667.

HOWARD, R. A. **Dynamic Programming and Markov Processes**. [S.l.]: The MIT Press, 1960.

HUSSAIN, F. et al. Machine learning for resource management in cellular and iot networks: Potentials, current solutions, and open challenges. **IEEE Communications Surveys & Tutorials**, IEEE, v. 22, n. 2, p. 1251–1275, 2020.

HUVAL, B. et al. An empirical evaluation of deep learning on highway driving. **arXiv preprint arXiv:1504.01716**, 2015.

KITCHENHAM, B. Procedures for performing systematic reviews. **Keele, UK, Keele University**, v. 33, n. 2004, p. 1–26, 2004.

KRAMER, M. A. Nonlinear principal component analysis using autoassociative neural networks. **AIChE Journal**, v. 37, n. 2, p. 233–243, 1991.

LIU, N. et al. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In: IEEE. **2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)**. [S.l.], 2017. p. 372–382.

LU, H. et al. Optimization of lightweight task offloading strategy for mobile edge computing based on deep reinforcement learning. **Future Generation Computer Systems**, Elsevier, v. 102, p. 847–861, 2020.

LUONG, N. C. et al. Optimal auction for edge computing resource management in mobile blockchain networks: A deep learning approach. In: IEEE. **2018 IEEE International Conference on Communications (ICC)**. [S.l.], 2018. p. 1–6.

MANDAL, S. K. et al. Dynamic resource management of heterogeneous mobile platforms via imitation learning. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 27, n. 12, p. 2842–2854, 2019.

MAO, H. et al. Resource management with deep reinforcement learning. In: **Proceedings of the 15th ACM Workshop on Hot Topics in Networks**. [S.l.: s.n.], 2016. p. 50–56.

MELO, F. S. Convergence of q-learning: A simple proof. **Institute Of Systems and Robotics, Tech. Rep**, p. 1–4, 2001.

MNIH, V. et al. Playing atari with deep reinforcement learning. **arXiv preprint arXiv:1312.5602**, 2013.

MNIH, V. et al. Human-level control through deep reinforcement learning. **nature**, Nature Publishing Group, v. 518, n. 7540, p. 529–533, 2015.

MURPHY, K. P. **Machine Learning: A Probabilistic Perspective**. [S.l.]: The MIT Press, 2012. (Adaptive Computation and Machine Learning series). ISBN 0262018020,9780262018029.

NEWQUIST, H. The brain makers, second edition. In: ____. [S.l.]: New York, NY: The Relayer Group, 2018. p. 491.

NORVIG, S. R. P. **Artificial Intelligence: A Modern Approach**. 4. ed. [S.l.]: Pearson, 2020. ISBN 2019047498,9780134610993,0134610997.

SCHMIDHUBER, J. Deep learning in neural networks: An overview. **Neural networks**, Elsevier, v. 61, p. 85–117, 2015.

SHAHIDINEJAD, A.; GHOBAEI-ARANI, M. Joint computation offloading and resource provisioning for e dge-cloud computing environment: A machine learning-based approach. **Software: Practice and Experience**, Wiley Online Library, 2020.

SILVER, D. et al. Mastering the game of go with deep neural networks and tree search. **nature**, Nature Publishing Group, v. 529, n. 7587, p. 484–489, 2016.

SUN, H. et al. Learning to optimize: Training deep neural networks for wireless resource management. In: IEEE. **2017 IEEE 18th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)**. [S.l.], 2017. p. 1–6.

TAIGMAN, Y. et al. Deepface: Closing the gap to human-level performance in face verification. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2014. p. 1701–1708.

WATKINS, C. J. C. H. Learning from delayed rewards. King's College, Cambridge United Kingdom, 1989.

WEI, Y. et al. User scheduling and resource allocation in hetnets with hybrid energy supply: An actor-critic reinforcement learning approach. **IEEE Transactions on Wireless Communications**, IEEE, v. 17, n. 1, p. 680–692, 2017.

YANG, H. et al. Learning-based energy-efficient resource management by heterogeneous rf/vlc for ultra-reliable low-latency industrial iot networks. **IEEE Transactions on Industrial Informatics**, IEEE, v. 16, n. 8, p. 5565–5576, 2019.

ZAHARIA, M. et al. Spark: Cluster computing with working sets. **HotCloud**, v. 10, n. 10-10, p. 95, 2010.

ZHANG, Y.; YAO, J.; GUAN, H. Intelligent cloud resource management with deep reinforcement learning. **IEEE Cloud Computing**, IEEE, v. 4, n. 6, p. 60–69, 2017.