UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA E ELETRÔNICA

Gerador de Testes para Verificação Funcional de Memória Compartilhada Baseado em *Deep Q-learning Networks*

Trabalho de Conclusão de Curso apresentado ao curso de Engenharia Eletrônica, Departamento de Engenharia Elétrica e Eletrônica da Universidade Federal de Santa Catarina como requisito parcial para obtenção do grau de bacharel no Curso de Engenharia Eletrônica.

Bruno de Vargas Zimpel

Orientador: Luiz Cláudio Villar dos Santos

Florianópolis, 26 de julho de 2019.

BRUNO DE VARGAS ZIMPEL

GERADOR DE TESTES PARA VERIFICAÇÃO FUNCIONAL DE MEMÓRIA COMPARTILHADA BASEADO EM *DEEP Q-LEARNING NETWORKS*

Trabalho de Conclusão de Curso apresentado ao curso de Engenharia Eletrônica, Departamento de Engenharia Elétrica e Eletrônica da Universidade Federal de Santa Catarina como requisito parcial para obtenção do grau de bacharel no Curso de Engenharia Eletrônica.
Orientador: Luiz Cláudio Villar dos Santos.

FLORIANÓPOLIS
2019

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Zimpel, Bruno de Vargas
Gerador de Testes para Verificação Funcional de Memória Compartilhada Baseado em Deep Q-learning Networks / Bruno de Vargas Zimpel ; orientador, Luiz Cláudio Villar dos Santos, 2019.
100 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia Eletrônica, Florianópolis, 2019.

Inclui referências.

1. Engenharia Eletrônica. 2. Memória compartilhada coerente. 3. Aprendizado por Reforço. 4. Geração de testes. I. Santos, Luiz Cláudio Villar dos. II. Universidade Federal de Santa Catarina. Graduação em Engenharia Eletrônica. III. Título.

Bruno de Vargas Zimpel

GERADOR DE TESTES PARA VERIFICAÇÃO FUNCIONAL DE MEMÓRIA COMPARTILHADA BASEADO EM *DEEP Q-LEARNING NETWORKS*

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Engenharia Eletrônica e aprovado em sua forma final pelo Curso de Graduação em Engenharia Eletrônica.

Florianópolis, 25 de julho de 2019.

Prof. Jefferson Luiz Brum Marques, PhD.
Coordenador do Curso

Banca examinadora:

Prof. Luiz Cláudio Villar dos Santos, PhD.
Universidade Federal de Santa Catarina

Prof. Walter Pereira Carpes Júnior, DSc.
Universidade Federal de Santa Catarina

Prof. Eduardo Augusto Bezerra, PhD.
Universidade Federal de Santa Catarina

Dedico este trabalho a todos aqueles que acreditaram nele e o apoiaram de alguma forma. Em especial, a minha mãe Roseli, por sua confiança e suporte inabalável

Agradecimentos

Primeiramente, gostaria de agradecer a minha mãe Roseli, meu pai Vilmar e minhas irmãs Julia e Camila, que desde sempre me fizeram valorizar o estudo e o aprendizado. Sempre me deram espaço e incentivo para que me tornasse a pessoa curiosa que sou e que me ensinaram as primeiras e mais valiosas lições que já tive. Agradeço também por sempre estarem ao meu lado e apoiarem minhas decisões, além de me ajudarem nesta última fase da graduação, seja apenas ouvindo sem entender muito do que eu estava falando sobre o TCC, seja trazendo carinho nos momentos difíceis.

Aos bons professores e professoras que tive durante toda a minha educação, mas especialmente durante a época da graduação. Agradeço àqueles que souberam ver o ser humano por trás do aluno, entender suas dificuldades e transformar o difícil no palpável. Com respeito, souberam passar um pouco dos seus conhecimentos adiante e tiveram papel ímpar na minha formação profissional e pessoal.

Aos amigos que fiz durante o meu período na faculdade, tiro de nossas experiências os mais diversos aprendizados. Vale mencionar alguns dos que estiveram comigo desde o primeiro semestre: Bruno, Roberto, Ion, Cláudio e Guilherme F e também alguns daqueles que conheci durante os demais períodos: Matheus, Ruan, Felipe e Guilherme M. Com todos, compartilhei momentos de derrotas e de vitórias, de tristezas e de alegrias. Com certeza, foram um dos maiores responsáveis por tornar épocas de muita pressão, suportáveis.

Por fim, agradeço a minha namorada Larissa pelo amor, carinho e parceria de todas as horas. Agradeço por me ouvir e encorajar desde a decisão do tema do TCC até a sua entrega: o seu apoio foi fundamental para a conclusão deste trabalho. Obrigado também por todos os bons momentos destes últimos anos, pelas risadas e pelo cuidado. Sorte é ter você ao meu lado.

I can live with doubt, and uncertainty, and not knowing. I think it's much more interesting to live not knowing than to have answers which might be wrong. I have approximate answers, and possible beliefs, and different degrees of certainty about different things, but I'm not absolutely sure of anything. (Richard P. Feyman, 1981)

Sometimes science is more art than science, Morty. Lot of people don't get that. (Rick Sanchez, 2013)

RESUMO

Este trabalho propõem um módulo diretor para geração de testes pseudo-aleatórios utilizados para verificação funcional de memória compartilhada coerente baseado em técnicas de Aprendizado por Reforço, mais especificamente, no algoritmo de *Deep Q-Learning Networks* (DQN). Através de uma proposição de ações baseada no comportamento de outro gerador reportado na literatura que representa o estado da arte no tema e otimizações ao algoritmo DQN original também reportadas internacionalmente, foi atingido um resultado satisfatório nos experimentos realizados. Para projetos de 32 *cores*, a cobertura acumulada ao final dos testes realizados foi maior do que a atingida pelos seus pares. Além disso, a partir de uma alteração no processo de treinamento da Rede Neural, foi possível fazer com que o agente aprendesse uma política de tomada de decisão muito similar ao comportamento que outro gerador reportado tem por construção.

Palavras-chave: Memória compartilhada coerente, Aprendizado por Reforço, geração de testes.

ABSTRACT

This work proposes a director module for pseudo random test generation used for functional verification of coherent shared memory based on Reinforcement Learning techniques, more specifically, on the Deep Q-Learning Networks (DQN) algorithm. Through the proposition of actions based on the behavior of another generator reported on the literature that represents the state of art on the subject and the use of optimizations to the original DQN algorithm also reported internationally, it was attained a satisfactory result on the realized experiments. For designs of 32 cores, the final cumulative coverage of the generated tests was bigger that the ones obtained by its peers. Besides that, through an alteration proposed on the training procedure for the Neural Network, it was made possible to the agent to learn a decision making policy that resembles the behavior that other reported generator has by construction.

Keywords: Coherent shared memory, Reinforcement Learning, test generation.

Lista de Figuras

1.1 *Framework* de verificação de memória compartilhada em chips *multicore* 5

3.1 Estrutura genérica de uma Rede Neural classificatória simples 20
3.2 Representações euclidianas de algumas das funções de ativação mais comuns na literatura 23
3.3 Diagrama de blocos de um MDP genérico 24
3.4 Arquitetura da rede implementada para o algoritmo *Rainbow* 33
3.5 Exemplo do conceito de treinamento em épocas reversas aplicado ao ambiente do jogo de Atari *Montezuma's Revenge* 35

4.1 Máquina de estados finitos do comportamento do protocolo MESI 38
4.2 *Framework* de verificação de memória compartilhada em chips *multicore* hospedeiro 42
4.3 Arquitetura final da Rede Neural implementada 49
4.4 *Framework* completo de verificação de memória compartilhada em *multicores* proposto 51

5.1 Evolução da cobertura no tempo DQNTG vs MTG e SCATG 58
5.2 Distribuição das ações tomadas pelo agente em uma execução típica do DQNTG 60

Lista de Algoritmos

1 Ação $a_2(\mathbb{S}, \mathbb{K})$ 47
2 Ação $a_3(\mathbb{S}, \mathbb{K})$ 47

Lista de Tabelas

5.1 Organização da memória cache considerada nos experimentos 56
5.2 Limites inferior e superior (absoluto) dos parâmetros de geração $\mathbb{n}$, $\mathbb{s}$ e $\mathbb{k}$ 57
5.3 Resultados finais - 1 hora de testes 58

Sumário

1 Introdução **1**
1.1 Motivação 1
1.1.1 Verificação de chips *multicore* 3
1.2 Técnicas de Aprendizado de Máquina 3
1.2.1 Geração de testes usando técnicas de aprendizado . 4
1.3 Escopo desse trabalho 4
1.3.1 Estrutura experimental 5
1.3.2 Contribuições técnicas 6
1.4 Organização da monografia 6

2 Trabalhos correlatos **9**
2.1 Principais técnicas para verificação funcional auxiliadas por Aprendizado de Máquina 10
2.2 Uso de Aprendizado por Reforço para geração de testes . . 12
2.3 Adequação do Aprendizado por Reforço para a verificação de chips *multicore* 14

3 Aprendizado por Reforço **17**
3.1 Aprendizado de Máquina 17
3.1.1 Redes Neurais e Aprendizado Profundo 19
3.2 Aprendizado por Reforço 23

3.2.1 Algoritmo escolhido: *Rainbow* 27
3.2.2 Detalhes de implementação - algoritmo *Rainbow* . 31
3.2.3 Técnica utilizada para o treinamento 33

4 Proposta de gerador baseado em DQN 37
4.1 Conceitos de geração dirigida explorados na proposta . . . 38
4.1.1 Classificação de transições entre estados 39
4.1.2 Modelo de cobertura 40
4.2 O *framework* de verificação hospedeiro 42
4.3 Novo módulo diretor proposto 43
4.3.1 Detalhes de implementação - novo módulo diretor . 44
4.3.2 Modificações no mecanismo de treinamento 51

5 Validação experimental 55
5.1 Estrutura experimental 55
5.2 Resultados experimentais 57
5.3 Considerações finais e melhorias futuras 61

6 Conclusão 65

Referências bibliográficas 67

Notação

Símbolo	Descrição
L1	Primeiro nível de memória cache
L2	Segundo nível de memória cache
L3	Terceiro nível de memória cache
y_j	Valor disponível na saída de um neurônio da Rede Neural
W	Vetor de pesos das transições em uma Rede Neural
w_j	Fator de peso de uma transição em uma Rede Neural
b_j	Fator de *bias* de uma transição em uma Rede Neural
S	Conjunto de estados de um Processo de Decisão de Markov
s_j	Estado qualquer j do conjunto S
A	Conjunto de ações de um Processo de Decisão de Markov
a_j	Ação qualquer j do conjunto A
R	Recompensa imediata esperada de um Processo de Decisão de Markov
$\mathbb{R}$	Recompensa imediata obtida de um Processo de Decisão de Markov

Símbolo	Descrição
γ	Fator de desconto
ϵ	Probabilidade do agente escolher uma ação aleatória dentre o conjunto de ações possíveis seguindo uma política ϵ-*greedy*.
$\pi(A\|s)$	Política de tomada de decisão das ações do conjunto A dado um estado s
$\mathbb{Q}$	Tabela que representa a qualidade de cada uma das ações do conjunto A em cada um dos estados do conjunto S
$\mathcal{Q}$	Vetor de qualidades de cada uma das ações do conjunto A dado o estado s
q_j	Qualidade da ação a_j no estado s
$\mathcal{R}$	Vetor de recompensas esperadas por cada uma das ações do conjunto A dado o estado s
θ	Representação da *Online Network*
$\overline{\theta}$	Representação da *Target Network*
$\mathcal{L}$	Representação da função perda
$\mathcal{A}$	Representação da função Vantagem
$\mathcal{V}$	Representação da função Valor
$z_j^{\mathcal{K}}$	Átomo j da função $\mathcal{K}$
ϵ^b	Variável aleatória
ϵ^w	Variável aleatória
$\odot$	Operação de produto matricial
Conv_j	Representação da camada j da Rede Neural, feita com uma Rede Convolucional
Noisy_j	Representação da camada j da Rede Neural, feita com uma Rede Ruidosa
$\mathfrak{n}$	Número de operações de memória (leituras e escritas)
$\mathfrak{s}$	Número de variáveis compartilhadas
$\mathbb{k}$	Número de conjuntos distintos da cache (*i.e.* posições de memória com o mesmo *index*) para os quais aquelas variáveis podem ser mapeadas

Símbolo	Descrição
$\mathbb{S}$	Conjunto de valores que $\mathtt{s}$ pode assumir
$\mathbb{N}$	Conjunto dos números naturais
$\mathbb{K}(\mathtt{s})$	Conjunto de valores que $\mathbb{K}$ pode assumir dado um $\mathtt{s}$
$\mathbb{N}$	Conjunto de valores que $\mathtt{n}$ pode assumir
cv_n	Cobertura de cada teste n
$\varnothing$	Conjunto vazio
Δ	Variação
CV	Cobertura total acumulada
ψ	*Objective Attained Threshold*
ϕ	*Zero Improvement Threshold*
n	Número de transições selecionadas do *Replay Buffer* durante o treinamento
τ	*Target Update*
η	*Learning Rate*

CAPÍTULO 1

Introdução

Este capítulo faz uma breve introdução ao trabalho desenvolvido. Inicialmente, apresenta-se o cenário atual do desenvolvimento de novos processadores e a importância da verificação funcional neste contexto. É feita uma breve apresentação do conceito de Aprendizado de Máquina e como ele é utilizado para a verificação funcional hoje. Seguindo isso, passa-se à descrição do escopo deste trabalho, seguido por uma seção dedicada a orientações quanto a organização desta monografia.

1.1 Motivação

Por muito tempo, ao precisar-se de mais desempenho em processadores, o desenvolvimento de novas arquiteturas podia sempre contar com o aumento da frequência de operação e do número de transistores dentro de um chip, já que estes seguiam a Lei de Moore [1], que previa que o número de transistores dentro de uma mesma área do chip dobraria em aproximadamente 18 meses.

Nas últimas duas décadas [2], o limite de dissipação de calor dentro de um mesmo chip estagnou o aumento de frequência de operação de

um processador, apesar de as dimensões da célula base destes sistemas continuarem a diminuir, mesmo que já não no mesmo ritmo previsto por Moore em 1965.

Em razão disso, para tentar manter um crescimento aceitavel do desempenho, o foco dos fabricantes de processadores passou a ser a exploração de paralelismo entre *threads*[1] [3]. Um processador passou a conter múltiplos núcleos de processamento (*cores*), cada um executando concorrentemente uma *thread* distinta de um programa paralelo. Essa mudança de foco deu origem aos assim chamados processadores *multicore*.

Tanto em processadores *singlecore* como em *multicores*, é comum o uso de hierarquias de memória, a fim de melhorar o desempenho do processador. Usualmente, são usados dois ou três níveis de memória cache (L1, L2 e L3) com tamanho progressivamente maior (e por consequência mais lentos) dentro do chip, seguidos pela memória principal (mais lenta ainda) fora do chip. Em *multicores*, é comum termos uma L1 privativa (ou privada) para cada núcleo, e os próximos níveis compartilhados entre todos os *cores*.

A relativa popularização destes sistemas - por exemplo, mesmo a família i3 de processadores da Intel tem dois ou mais núcleos [4] - e também a sua escala - o processador Intel Xeon E7-8890, por exemplo, tem 20 núcleos [5] - trouxe outros problemas, como manter esta memória compartilhada coerente, isto é, garantir que uma leitura de uma mesma variável compartilhada por núcleos diferentes retorne o mesmo valor, independentemente de quais instruções vieram antes ou depois desta leitura em cada um deles.

Há indícios de que a coerência de cache *on-chip* é um requisito que continuará presente por algum tempo, mesmo com o aumento de núcleos por chip, já que os principais *overheads* criados pelos protocolos de coerência independem do número de *cores* [6]. Todavia, ao aumentarmos o número de núcleos de um sistema, o projeto do processador em si fica mais complexo, sendo que um de seus principais obstáculos é manter a coerência da memória cache compartilhada.

[1] Uma *thread* é uma sequência de instruções que compartilham um mesmo espaço de endereços.

1.1.1 Verificação de chips *multicore*

Para o desenvolvimento de um novo sistema, podemos elencar seis principais etapas: especificação, projeto do nível arquitetural, projeto do chip, manufatura, encapsulamento e testes. Na segunda e na última etapa, é onde o sistema é testado. São testes, respectivamente, *pre-silicon* e *post-silicon* (antes e depois da integração física), o primeiro sendo realizado em simulação e o outro em um protótipo do chip.

Os testes *pre-silicon*, no contexto de processadores *multicore*, são chamados de **verificação funcional** do processador e são compostos de baterias de programas de testes executados em simulador, a fim de encontrar erros no projeto. Quando estes são encontrados nesta etapa, obviamente economiza-se tempo de reprojeto do sistema e, consequentemente, o custo final do projeto, o que torna tais testes especialmente atraentes. O principal desafio é que testes executados em simulador são ordens de magnitude mais lentos que quando executados diretamente no sistema físico. Portanto, os testes realizados nesta etapa devem ser eficientes e eficazes em encontrar os erros.

Há duas abordagens para a geração de testes voltados à verificação funcional: *Biased Random Test Generation* (RTG) e *Directed Test Generation* (DTG). A abordagem RTG adiciona restrições para influenciar a geração de testes aleatórios. A abordagem DTG tende a aumentar a cobertura através de testes menores, onde a geração de testes é adaptativa, através do uso de inferência estatística, meta-heurística ou inteligência artificial. Alguns exemplos de verificação funcional com abordagem DTG são mostrados em [7], [8] e [9].

1.2 Técnicas de Aprendizado de Máquina

Aprendizado de Máquina é um campo da Inteligência Artificial que estuda algoritmos e técnicas que permitem a uma máquina aprender uma função sem ser explicitamente programada para isso. Principalmente nos últimos anos, com o aumento da capacidade de processamento - especialmente processamento paralelo - houve um aumento do interesse da comunidade científica e corporativa no seu uso aplicado aos mais diversos problemas, como visão computacional e modelos preditivos financeiros, por exemplo.

A principal divisão destas técnicas é em relação a sua interação com o ambiente. Por esta característica, pode-se dividir o Aprendizado de Máquina em três vertentes principais: Aprendizado Supervisionado [10], Aprendizado Não-Supervisionado [11] e Aprendizado por Reforço [12]. Este último é explorado nesta monografia.

Uma prática da comunidade científica neste meio é utilizar um mesmo ambiente para otimização dos algoritmos em si, antes de partir para a sua aplicabilidade prática. No caso dos algoritmos de Aprendizado por Reforço, em geral o ambiente utilizado é o de jogos eletrônicos de Atari, onde o objetivo é fazer com que a pontuação final do agente implementado pelo algoritmo seja superior aos de outros algoritmos reportados, como feito em [13], [14] e [15], por exemplo.

1.2.1 Geração de testes usando técnicas de aprendizado

Um dos desafios onde estas técnicas podem ser utilizadas na prática é para a geração de testes dirigidos para verificação funcional, tanto em Software [16] como em Hardware [17].

Especificamente sobre verificação de memória compartilhada, pode-se citar, por exemplo, o trabalho de Marco Elver e Vijay Nagarajan, de 2016 [7], que propõe um *framework* de verificação baseado em programação genética, uma das linhas de pesquisa do Aprendizado de Máquina.

Se limitarmos apenas à vertente do Aprendizado por Reforço, contudo, não foi encontrado nenhum trabalho reportado na comunidade científica do seu uso para verificação de memória compartilhada coerente, o que este trabalho se propõem a fazer.

1.3 Escopo desse trabalho

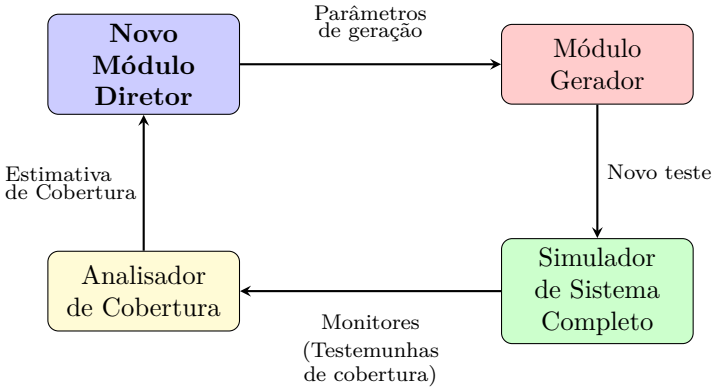
Este trabalho pretende explorar uma técnica de Aprendizado de Máquina, mais especificamente, Aprendizado por Reforço, para a geração dirigida de testes para verificação funcional de memória compartilhada coerente em chips *multicore*.

A partir desta técnica, foi criado um módulo diretor a ser inserido em um *framework* de verificação já existente desenvolvido anteri-

ormente no laboratório *Embedded Computing Laboratory* - ECL, localizado no INE/CTC/UFSC.

A organização geral do *framework* de verificação de memória compartilhada existente e a posição onde será inserido o novo módulo diretor proposto pode ser representada pela Figura 1.1.

Figura 1.1: *Framework* de verificação de memória compartilhada em chips *multicore*



Fonte: Autor

1.3.1 Estrutura experimental

A principal base para o desenvolvimento do novo módulo diretor foi a implementação do algoritmo *Rainbow* [13][2], apresentado por Hessel *et al.* em 2017, disponível publicamente no GitHub [18] e alterado para a aplicação aqui proposta.

Para a simulação comportamental do processador, foi utilizado o simulador proposto por Nathan Binkert *et al.* em 2011 [19], chamado Gem5, também disponível publicamente no GitHub [20]. O Gem5 é representado pelo módulo nomeado como Simulador de Sistema Completo na Figura 1.1.

[2] O algoritmo *Rainbow* foi criado sobre o algoritmo original de *Deep Q-Learning Networks* (DQN)

1.3.2 Contribuições técnicas

Este trabalho faz parte de uma das linhas de pesquisa atuais do ECL e foi um trabalho desenvolvido em conjunto com o mestrando Nícolas Pfeifer.

A definição do algoritmo utilizado, assim como dos parâmetros iniciais do agente (ações, estados e recompensas)[3] foi uma decisão conjunta.

A ligação do algoritmo *Rainbow* com o Gem5, assim como a codificação inicial dos parâmetros ficou a cargo do mestrando.

Alterações pontuais no código desenvolvido ficaram a cargo dos dois envolvidos no projeto, assim como as baterias de testes realizados e a análise dos resultados destas, para então serem propostas novas alterações no projeto.

As principais contribuições do trabalho desenvolvido são:

- A aplicação de um algoritmo de Aprendizado por Reforço que representa o estado da arte para um problema prático de verificação funcional de memória compartilhada coerente;
- Uso do conhecimento de outro gerador que representa o estado da arte na literatura para definição dos parâmetros do agente;
- Proposta de uma nova técnica para o treinamento da Rede Neural.

1.4 Organização da monografia

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 trata de uma breve revisão dos trabalhos correlatos utilizados como referência para o desenvolvimento deste trabalho, tanto no que se refere ao uso de Aprendizado de Máquina para verificação funcional como também especificamente do uso de Aprendizado por Reforço na geração de testes.

O Capítulo 3 revisa diversos conceitos de Aprendizado de Máquina, Redes Neurais, Aprendizado Profundo e Aprendizado por Reforço, para então entrar em detalhes de implementação do algoritmo utilizado como

[3] A serem explicados em mais detalhes no Capítulo 4

base para este trabalho [13] e a técnica utilizada para melhorar o treinamento do agente [15].

O Capítulo 4 apresenta os detalhes da implementação do novo módulo diretor proposto. Uma breve revisão dos conceitos de geração dirigida para memória compartilhada coerente é feita e são apresentados formalmente quais são os parâmetros de geração de testes e como as ações do agente influenciam cada um deles. A Seção 4.3.1 trata das alterações feitas na Rede Neural original e qual sua arquitetura final. Neste capítulo também são apresentados os valores finais dos parâmetros da rede.

O Capítulo 5 apresenta a estrutura experimental utilizada, os testes realizados e os resultados obtidos. É feita uma breve análise destes, assim como os pontos forte e melhorias possíveis ao projeto desenvolvido.

Por fim, o Capítulo 6 faz um breve apanhado do trabalho desenvolvido, os resultados obtidos e os próximos passos para trabalhos futuros.

CAPÍTULO 2

Trabalhos correlatos

Este capítulo apresenta a revisão bibliográfica dos principais aspectos deste trabalho: técnicas utilizadas em **verificação funcional** e aplicações do uso de **Aprendizado por Reforço** em geração de testes.

A Seção 2.2 analisa trabalhos sobre geração de testes baseada especificamente em Aprendizado por Reforço, embora nenhum deles aborde a verificação funcional de memória compartilhada. Finalmente, a Seção 2.3 faz uma breve síntese dos principais resultados úteis ao contexto desta monografia e à avaliação da relevância do trabalho proposto.

Embora a verificação funcional de chips *multicore* tenha sido abordada por uma variedade de técnicas distintas, é relativamente raro encontrar-se técnicas baseadas em Aprendizado de Máquina (ao contrário do que ocorre em outras áreas de aplicação), especialmente quando se considera a verificação baseada em cobertura, que é o foco deste trabalho. A Seção 2.1 resume as principais técnicas de verificação funcional baseadas em Aprendizado de Máquina.

2.1 Principais técnicas para verificação funcional auxiliadas por Aprendizado de Máquina

Em 2003, Shai Fine e Avi Ziv [9] usaram Redes Bayesianas para relacionar estatisticamente o resultado dos testes em relação à cobertura total do sistema com as diretivas para a criação de novos planos de testes a serem simulados. O trabalho tinha dois objetivos principais: 1) realimentar os resultados dos testes realizados e a geração de novos testes (muitas vezes feito de forma manual), e 2) otimizar os testes gerados levando em conta o resultado em relação à cobertura de cada um deles. Para isso, ao invés de modelar o comportamento de um processador como uma FSM (*Finite State Machine*) - tipicamente complexa - o processo de geração é que foi modelado, isto é, o processo de tentativa-e-erro liderado pelo time de verificação, que correlata os testes gerados e os resultados obtidos. Mapeou-se o *pipeline* completo de um processador em tarefas cuja conclusão representava uma porcentagem da cobertura da verificação do sistema e usou-se o histórico de cobertura para treinar a rede. Depois de treinada, a rede alcançava pontos de cobertura raros em aproximadamente 1/3 do tempo da técnica à qual foi comparada (geração de testes puramente aleatórios). Uma das principais desvantagens desta abordagem é que a técnica é dependente da arquitetura, já que a definição de cobertura usada é uma abstração de uma microarquitetura que implementa uma ISA (*Instruction Set Architecture*) específica.

Já em 2017, Raviv Gal *et al.* [21] apresentaram uma técnica utilizada na IBM para otimizar o trabalho do especialista em verificação. Através de uma compilação estatística da probabilidade de cada teste cobrir eventos do sistema, o armazenamento dos resultados dos testes ocupa menos espaço do que todos os registros de cobertura. Ao capturar de forma esquemática essa relação estatística, o trabalho do especialista é facilitado, pois este saberá quais são as alterações necessárias nos planos de teste a fim de alcançar maior cobertura. Além disso, a técnica, chamada *Template Aware Coverage* (TAC), também apresenta a relação direta entre um estímulo e um evento coberto, facilitando o levantamento de hipóteses - e também a sua confirmação/refutação - em testes subsequentes. A partir de técnicas de *Data Analytics*, é possível realizar otimizações baseadas nos resultados obtidos. As otimizações realizadas visam minimizar a quantidade de estímulos necessários para

cobrir um evento específico, com base na probabilidade histórica de cobri-lo utilizando cada um dos estímulos que anteriormente fizeram parte de algum plano de teste executado.

Também em 2017, Kuo-Kai Hsieh *et al.* [22] mostraram como utilizaram duas técnicas de Aprendizado de Máquina - *Proccess Discovery* e *Grammatical Inference* - para gerar novos testes a partir de antigos já validados. Entretanto, este trabalho aborda a verificação de segurança e não a verificação funcional de um sistema. Diferente desta, um teste de verificação de segurança procura encontrar ações de propósito prejudicial, como ataques, por exemplo. Cada teste constitui um ataque e, assim, a abordagem foi de decomposição de cada teste em primitivas (pedaços de código independentes) e o mapeamento das relações estatísticas entre elas, para enfim gerar novas combinações, criando-se assim novos testes. Além disso, adicionaram-se restrições no processo: a primitiva X só pode ser executada depois da primitiva Y, por exemplo. O resultado destes novos ataques gerados foram avaliados a partir da métrica de cobertura do DUT (*Device Under Test*). A partir de 30 testes originais, foram gerados 500 novos, alguns cobrindo os pontos já cobertos, mas outros cobrindo pontos nunca cobertos pelos testes originais.

Por outro lado, a literatura reporta um trabalho com foco exatamente em verificação de memória compartilhada (coerência de cache e consistência de memória). Em 2016, Marco Elver e Vijay Nagarajan [7] apresentam McVerSi, um *framework* que propõem uma abordagem baseada em Programação Genética para a geração dirigida de testes, onde cada teste é associado a um cromossomo e cada operação a um gene. Neste caso, um cromossomo é representado por um grafo orientado acíclico[1] onde os vértices representam operações (leitura ou escrita em memória) e onde os arcos representam uma ordem parcial dessas operações. Para gerar novos testes, os autores usam a técnica de *crossover*, através de um cruzamento seletivo que prioriza operações de memória que tendem a gerar condições de corrida [23], a fim de estimular a cobertura de transições pouco frequentes entre estados

[1] Grafos orientados são estruturas matemáticas que representam as relações (arcos) entre um ou mais elementos (vértices). Grafos orientados acíclicos são grafos em que, para todo vértice v, não existe um caminho que comece e termine em v. Neste caso, os vértices são as operações e os arcos são as possíveis ordens das operações em um teste.

do protocolo de coerência. A principal desvantagem dessa técnica é a dependência da escolha pelo usuário dos parâmetros do gerador, tais como a quantidade de instruções por teste e a quantidade de memória utilizada em um teste, o que pode levar o usuário a limitar inadvertidamente o potencial de detecção de erros ao selecionar valores indevidos. Comparado a abordagens alternativas baseadas em geração de testes pseudo-randômicos e testes decisivos (*litmus tests*), o gerador McVersi detectou 11 erros de projeto não encontrados pelas outras técnicas. A técnica aplicada no gerador McVersi representa o estado da arte da geração de testes dirigidos avaliada por Aprendizado de Máquina.

Entretanto, em 2018, uma técnica não baseada em Aprendizado de Máquina mostrou resultados melhores dos que os obtidos com o gerador McVersi em vários casos. Por isso, ela será abordada a seguir, já que servirá de referência para avaliar o desempenho da técnica baseada em Aprendizado de Máquina aqui proposta.

Andrade *et al.* [8] propuseram a geração dirigida de testes baseada em modelo de cobertura e, ao invés de recorrer a técnicas de Aprendizado de Máquina, propuseram uma técnica que explora restrições não convencionais (*chaining and biasing constraints*), definidas a partir da exploração de propriedades gerais não somente de caches individuais, mas também do protocolo que garante a sua coerência, para um melhor controle da cobertura de transições entre estados. Foram definidas três classes diferentes de transições cuja alternância aumenta a qualidade de testes não-determinísticos. Comparada com McVerSi, a técnica foi mais rápida (1.9 a 5.7 vezes) em obter maior cobertura (68.9% contra 67.5%) em projetos com 32 núcleos. Uma das principais vantagens deste trabalho é sua independência da métrica de cobertura escolhida, protocolo de coerência e parâmetros da cache.

2.2 Uso de Aprendizado por Reforço para geração de testes

O Aprendizado por Reforço (a ser abordado em detalhes no Capítulo 3) é uma técnica de Aprendizado Máquina que difere do Aprendizado Supervisionado e do Não-Supervisionado por não haver conhecimento

sobre os pares de entrada e saída, deixando o **agente**[2] completamente livre para aprender por meio de recompensas.

Seu uso é mais recente do que técnicas mais tradicionais de Aprendizado de Máquina (como Redes Bayesianas, por exemplo). Portanto, há poucos relatos na literatura que descrevem sua aplicação em um ambiente de verificação. Ainda assim, foram encontrados alguns trabalhos que se mostraram relevantes para o escopo desta monografia, dos quais alguns são analisados abaixo.

Em validação funcional de software, é comum a utilização da técnica de SBST (*Search-Based Software Testing*, Phil McMinn [24] 2011) para geração automática de dados de testes, baseados em diversas técnicas meta-heurísticas, em substituição a um trabalho geralmente manual e laborioso. A técnica é baseada na aplicação de técnicas de otimização por busca, *e.g.* Programação Genética, para resolver problemas relacionados a testes de softwares. Por se tratar de um ambiente de tomada de decisão sequencial, em 2018, Junhwi Kim *et al.* [16] apresentaram um *framework* de SBST baseado em Aprendizado por Reforço, onde o SUT (*Software Under Test*) é transformado no ambiente em que o agente será treinado. No artigo, são apresentadas diversas evidências de que o agente aprendeu o comportamento meta-heurístico que funciona bem para encontrar soluções para problemas arbitrários, uma vez que, mesmo em situações em que não foi treinado, o agente obteve resultados positivos.

Já em 2011, Alex Groce [25] tratou da geração de testes para validação funcional de *software* sob outra óptica: ao invés de abordar o problema para o software completo, o foco foi a validação de trechos menores em um projeto de software, como bibliotecas Container[3], pacotes de banco de dados, entre outras. Para validação destes blocos, há diversas ferramentas (*checkers*) disponíveis, contudo, estes são limitados pela linguagem na qual foram escritas. Caso o programador não tenha familiaridade com outras linguagens, a opção aqui é a geração de valores aleatórios de entrada, a qual já se provou tão efetiva quanto *model checkers* em alguns casos. Em outros casos, entretanto, tem-se um *gap* relevante no contexto de projeto de software. Assim, o autor

[2] O sistema que interage com o ambiente a partir de ações.

[3] Um Container é uma unidade padrão de software que empacota o código e todas as suas dependências para que o aplicativo seja executado de maneira rápida e confiável de um ambiente de computação para outro.

apresenta um modelo baseado em Aprendizado por Reforço para substituir a geração de testes aleatórios, de forma que esta técnica atenda todos os casos possíveis. Para isso, o algoritmo sugere um valor para o próximo teste baseado no contexto atual em que se encontra o SUT e o agente é recompensado caso este valor gere um comportamento ainda não observado. Os resultados variaram bastante dependendo do SUT: alguns apresentaram um resultado muito melhor comparado a testes aleatórios, enquanto em outros, este resultado ficou abaixo.

Em relação à testabilidade e testes no projeto de hardware digital, vale citar o trabalho de Shakeri *et al.* [17], de 2010. A abordagem dos autores é usar Aprendizado por Reforço para influenciar testes pseudo-aleatórios com o objetivo de encontrar erros no projeto. O agente é treinado e executado em um simulador associado à representação do projeto sob verificação, a qual é descrita em HDL (*Hardware Description Language*). O agente é recompensado em relação ao número de erros de projeto cobertos pela primeira vez. Ou seja, o agente foi treinado para aprender a gastar cada vez menos tempo para encontrar novos erros. Em média, houve uma melhora de 15.3% em relação a técnicas tradicionais e, em alguns casos, a nova técnica foi 42% mais rápida em encontrar o mesmo número de erros.

2.3 Adequação do Aprendizado por Reforço para a verificação de chips *multicore*

Embora o uso de Aprendizado por Reforço tenha trazido impacto positivo na geração de testes para validação de software e para verificação de sistemas digitais, a literatura não reporta seu uso para a verificação de memória compartilhada coerente em chips *multicore*. Neste sentido, a abordagem deste trabalho é original.

Em contraste com outros métodos de Aprendizado de Máquina (muitas vezes usados para aplicações em Regressão ou Classificação), o Aprendizado por Reforço foi desenvolvido especificamente para tomada de decisões sequenciais, como apresentado por Bai Chen *et al.* [26], em 2011. Em seu núcleo está a Cadeia de Markov, uma sequência de processos estocásticos, onde o próximo estado depende apenas do estado atual e não de todos os estados que o precederam. Além disso, téc-

nicas como Programação Genética ou Redes Bayesianas, por exemplo, não levam em conta o aspecto sequencial da geração de testes[4] e isso é geralmente gerenciado por um controlador externo, que alimenta a rede com eventos ainda não alcançados. Essas características mostram que vale a pena explorar o Aprendizado por Reforço num contexto de verificação baseado em geração dirigida de testes.

Quando aplicada à geração de testes dirigidos, a tomada sequencial de decisões, que é uma característica aprendida pelo agente na técnica de Aprendizado por Reforço, tem o potencial de se traduzir em maior cobertura em menor tempo[5]. Além disso, a característica exploratória[6] do Aprendizado por Reforço tem o potencial de estimular transições que dificilmente seriam cobertas por outras técnicas, tal como foi observado no trabalho de Shaketi *et al.* [17].

Ao adotar o Aprendizado por Reforço para a geração dirigida de testes, tomou-se o cuidado de torná-lo independente da arquitetura sob verificação, através de ações inspiradas no modelo de cobertura proposto em [8], o qual resulta da aplicação de restrições ditadas por propriedades gerais de cache individuais e de propriedades genéricas de protocolos de coerência [27].

Para facilitar a comparação com as técnica propostas em [8] e [7], adotou-se o Gem5 [19] como ambiente para treinamento e execução dos experimentos.

[4]Isto é, qual a melhor sequência possível de testes a serem realizados em um dado ambiente para otimizar o resultado.

[5]Já que esta característica ainda não foi explorada no domínio de verificação de memória compartilhada, mas já apresentou resultados interessantes em outras aplicações.

[6]A característica exploratória do Aprendizado por Reforço será explicada no Capítulo 3.

CAPÍTULO 3

Aprendizado por Reforço

Este capítulo apresenta uma breve introdução aos conceitos de Aprendizado de Máquina, Redes Neurais, Aprendizado Profundo e Aprendizado por Reforço. Em relação a este último, são apresentadas algumas técnicas responsáveis pelo bom desempenho de algoritmos que representam o estado da arte, os quais são também utilizados no desenvolvimento deste projeto.

3.1 Aprendizado de Máquina

Aprendizado de Máquina é um campo da Inteligência Artificial que estuda os algoritmos e técnicas que permitem a uma máquina aprender uma função sem ser explicitamente programada para isso. O termo - e sua definição - nasceram em 1959, em um trabalho publicado por Arthur Lee Samuel [28], um dos grandes pioneiros na área. Em 1997, Tom M. Mitchell [29] formalizou a definição de aprendizado da seguinte forma:

> "Um programa de computador é dito capaz de aprender a partir de uma experiência E a respeito de uma série de tarefas T com sua performance sendo medida por P se a sua performance nas tarefas T, medida por P, **melhora** com a experiência E".

Como as tarefas não são explicitamente programadas, o programa aprende através de reconhecimento de padrões e inferências, obtidas a partir de um modelo matemático construído sobre um conjunto de dados, em uma fase de treinamento. Tem-se três vertentes principais de Aprendizado de Máquina: Aprendizado Supervisionado [10], Aprendizado Não-Supervisionado [11] e Aprendizado por Reforço [12]. As duas primeiras são descritas brevemente abaixo, já o Aprendizado por Reforço é tratado em uma seção específica subsequente, por se tratar de um conceito mais relevante para este trabalho.

Aprendizado Supervisionado: Nesta vertente, o conjunto de dados de treinamento é composto por uma ou mais entradas e a saída desejada para aquelas entradas. Sendo assim, o objetivo do algoritmo é encontrar uma função que represente a saída em relação às entradas, com base em exemplos, para poder predizer uma saída, dadas novas entradas futuras, *i.e.* inferir uma distribuição probabilística condicional. Matematicamente, podemos representar o Aprendizado Supervisionado pela seguinte equação:

$$p_X(y|x) \tag{3.1}$$

Onde p_X é a probabilidade de uma saída y, dada uma entrada x. Ela é usada principalmente para **Classificação** e **Regressão**. No primeiro caso, as saídas são limitadas a um conjunto fixo de saídas válidas e o objetivo é classificar cada novo conjunto de dados de entrada em uma das saídas possíveis. Reconhecimento de imagens e visão computacional são exemplos de aplicações. Já em regressão, a saída é um conjunto de valores contínuos e o objetivo é encontrar uma função matemática que descreva o comportamento para novas entradas. Uma aplicação muito comum nesta vertente sendo análises de pesquisas estatísticas. Alguns algoritmos de Aprendizado Supervisionado são: SVM (*Support Vector Machine*) [30], Regressão Linear [31], Regressão Logística [32] e K-NN (*K-Nearest Neighbor*) [33].

Aprendizado Não-Supervisionado: Aqui, o conjunto de dados contém apenas entradas e não saídas relacionadas a estas entradas. O objetivo dos algoritmos é encontrar padrões, estruturas ocultas ou relações estatísticas entre os dados de entrada, classificando-os em grupos não definidos ainda, *i.e.* inferir uma distribuição probabilística *a priori*. Em comparação com o Aprendizado Supervisionado, podemos representar o Aprendizado Não-Supervisionado através da seguinte equação:

$$p_X(x) \tag{3.2}$$

Onde p_X é a probabilidade de um valor x ocorrer, *antes* de se ter qualquer evidência dele. Essa técnica é utilizada principalmente em estimação de densidade estatística, aplicada a criar agrupamentos com base em dados crus. Um exemplo de aplicação é a segmentação de perfis de clientes em um mercado qualquer, o que torna mais assertiva a abordagem de equipes de vendas. Alguns algoritmos que utilizam a abordagem de Aprendizado Não-Supervisionado são EM (*Expectation-Maximization*) [34], HCA (*Hierarchical Cluster Analysis*) [35] e OPTICS (*Ordering Points To Identify the Clustering Structure*) [36].

3.1.1 Redes Neurais e Aprendizado Profundo

Uma classe de algoritmos que pode ser usada para qualquer vertente de Aprendizado de Máquina (inclusive Aprendizado por Reforço) são as Redes Neurais. Estruturas vagamente inspiradas em um cérebro humano, as Redes Neurais são utilizadas para as mais diversas aplicações, como classificação e regressão. Nos algoritmos anteriormente citados, os atributos relevantes e sua relação são definidos antes do treinamento do agente[1], já que estes são dados de entrada bem definidos. Uma vantagem das Redes Neurais é de que estes atributos são descobertos **durante** o treinamento, ou ainda, pode haver a criação de novos atributos a partir da combinação de dois ou mais dados de entrada.

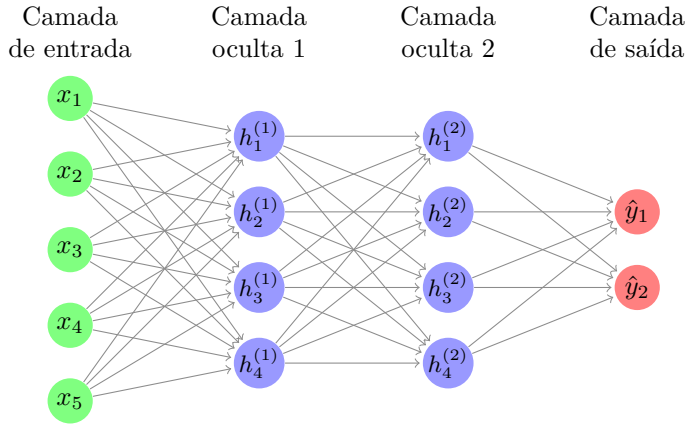
Há diversos tipos de Redes Neurais, como Redes Convolucionais (muito usadas para reconhecimento de imagens) e Redes LSTM (*Long Short-Term Memory*) (muito usadas para reconhecimento de fala). Porém, por simplicidade, o conceito aqui tratado será o de uma Rede

[1] O sistema que interage com o ambiente a partir de ações.

Neural classificatória simples (sem realimentação).

A estrutura básica de uma Rede Neural é formada por diversas camadas (*layers*) de neurônios, que nada mais são do que estruturas que guardam um número em sua saída (geralmente entre 0 e 1), interconectados entre si, como mostra a Figura 3.1:

Figura 3.1: Estrutura genérica de uma Rede Neural classificatória simples



Fonte: Autor

O número de neurônios por camada - assim como o número de camadas - varia significativamente entre aplicações. Por exemplo, na camada de entrada, ele pode ser definido pelo número de *pixels* em uma imagem ou pelo tamanho de um vetor de valores. Por outro lado, na camada de saída este valor pode variar entre um (em aplicações de regressão) e um número genérico n, que pode ser definido pelo número de classes possíveis dentre as quais uma imagem pode ser classificada, por exemplo.

A principal vantagem das Redes Neurais em relação a outros algoritmos, contudo, é dado pela sua capacidade de **generalização** e as principais responsáveis por isso são as chamadas **Camadas Ocultas**. Estas, que podem variar em número[2] e tamanho[3], mapeiam relações entre cada um dos neurônios de entrada até a saída, através de opera-

[2]Número de Camadas Ocultas em uma mesma Rede Neural.

[3]O tamanho de uma Camada se refere ao número de neurônios que a compõem.

ções lineares (somas e multiplicações ou convoluções no caso de Redes Convolucionais), são estas operações que definem o valor na saída de cada neurônio. Cada uma dessas relações são transições entre neurônios, representadas pelas linhas em cinza na Figura 3.1, que podem ser descritas pela Equação 3.3:

$$y = wx + b \tag{3.3}$$

Onde w representa o peso daquela transição, x é o valor do neurônio origem da transição, y é o valor do neurônio destino e b é um termo de *bias* adicionado durante o treinamento.

O peso w e o valor de *bias* b são otimizados durante o treinamento, a partir de um algoritmo denominado *Backpropagation* [37] que utiliza o método matemático de Gradiente Descendente [38] de forma a minimizar uma função de **perda global**. As Camadas Ocultas são chamadas assim justamente porque não é definido como a relação entre atributos e como eles serão representados nas camadas será feito: isso é definido pela própria rede durante o aprendizado.

Além disso, a Rede Neural nos permite definir uma função f^* que aproxime uma função f quando não temos a possibilidade de separar os atributos por hiperplanos bem definidos[4]. Com as Camadas Ocultas, podemos criar novos atributos derivados dos originais e obter regiões de decisão mais complexas[5], obtendo resultados melhores mesmo com atributos menos "organizados". A principal vantagem é que estes novos atributos não precisam ser propostos pelo usuário, eles são definidos pela própria rede.

Para exemplificar esse conceito de hiperplanos não definidos e regiões de decisão mais complexas, podemos utilizar um exemplo clássico e simples: uma rede classificatória que diz se uma imagem contém um cachorro ou um gato. Não sabemos quais atributos das imagens a rede considera como sendo mais importantes nesta classificação. Uma camada oculta pode estar analisando o formato das orelhas, por exemplo,

[4]Um "hiperplano bem definido"se refere a relação entre os atributos (dados de entrada). Em uma rede classificatória com o objetivo de classificar imagens em espécies de plantas, por exemplo, poderíamos tem um hiperplano definido por número de pétalas vs presença de espinhos.

[5]Hiperplanos definidos a partir de atributos não definidos pelos dados de entrada. Na mesma aplicação anterior, por exemplo, uma das Camadas Ocultas pode avaliar o quadrante superior direito de cada uma das imagens a procura de algo específico.

enquanto outra camada analisa o focinho. A ordem ou os atributos são definidos pela própria rede durante o treinamento, a fim de minimizar a função de perda.

É comum a utilização de uma **Função de Ativação**[6] para adicionar às redes um elemento não-linear, já que só combinações lineares não representam completamente uma função genérica qualquer, o que pode fazer com que não consiga-se aproximar corretamente a função desejada. Entre as funções de ativação mais comuns, podemos citar a Função Sigmoide (σ), a Função Tangente Hiperbólico (tanh) e a Função ReLU (*Rectified Linear Unit*), definidas abaixo, respectivamente:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

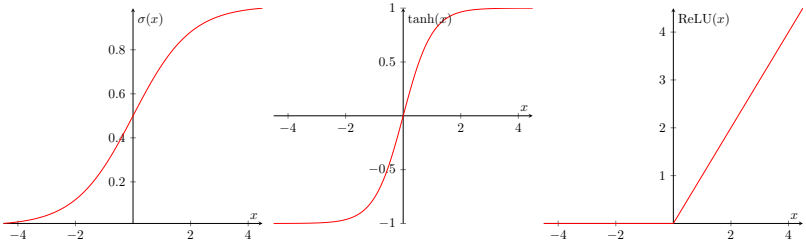
$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3.5}$$

$$\text{ReLU}(x) = \begin{cases} 0, & \text{se } x < 0 \\ x, & \text{se } x \geq 0 \end{cases} \tag{3.6}$$

Na Figura 3.2 estão representadas cada uma das funções acima descritas e fica claro que cada uma delas mitiga o problema de uma forma diferente. Não se entrará aqui no mérito das vantagens de cada uma, mas este ponto será explorado na descrição da escolha da Função de Ativação utilizada neste trabalho, na Seção 3.2.2.

[6] Função matemática utilizada como transição entre duas Camadas de neurônios ou na saída de uma Rede Neural.

Figura 3.2: Representações euclidianas de algumas das funções de ativação mais comuns na literatura



(a) Função $\sigma(x)$ (b) Função $\tanh(x)$ (c) Função $\mathrm{ReLU}(x)$

Fonte: Autor

Embora as Redes Neurais tenham sido introduzidas em 1943 por Warren S. McCulloch e Walter Pitts [39], sua popularização é relativamente recente, pois sua alta demanda de processamento só de tornou viável e prática com a disponibilidade de processadores paralelos (*e.g. multicores* e GPUs).

Essa disponibilidade tecnológica também deu origem a outro conceito, o de Aprendizado Profundo (*Deep Learning*), que nada mais é do que a utilização de Redes Neurais com múltiplas Camadas Ocultas. Não há um consenso entre qual o limiar entre uma Rede Neural normal (rasa) e uma Rede Neural Profunda. Jürgen Schmidhuber, por exemplo, em 2014 [40], defendeu que quando uma rede tem mais de duas Camadas Ocultas, ela pode ser considerada Profunda, conforme aumentamos esse número, se faz necessária também a criação de novos conceitos. Segundo ele, quando temos um número maior do que 10 Camadas Ocultas, já estamos em um terreno de Redes Neurais **Muito Profundas**.

3.2 Aprendizado por Reforço

Diferentemente dos Aprendizados Supervisionado e Não-Supervisionado, o Aprendizado por Reforço não é construído tendo como base a ideia de achar uma função que defina a relação entre dois conjuntos de dados. Ao invés disso, nesta vertente o foco é otimizar a forma com que um **Agente** faz escolhas em um determinado **Ambiente**, sem nenhuma supervisão, apenas a partir de uma função objetivo chamada **Recom-**

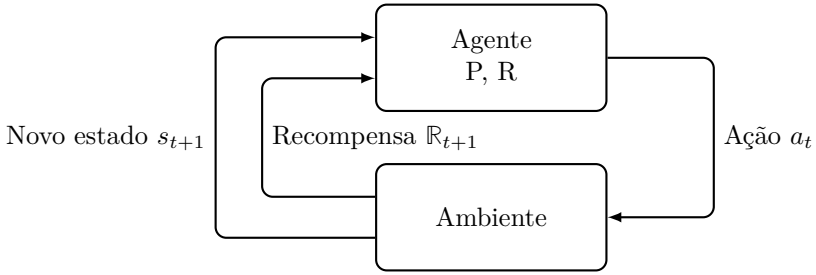
pensa.

A relação Agente-Ambiente é tipicamente formulada a partir de um Processo de Decisão de Markov (MDP - *Markov Decision Process*) [44], que é definido a partir da tupla (S, A P, R) onde:

- S é o conjunto de estados $\{s_1, s_2, ..., s_n\}$;
- A é o conjunto de ações possíveis $\{a_1, a_2, ..., a_n\}$
- P é a probabilidade de que a ação a_t no estado s_t no tempo t leve o agente ao estado s_{t+1}, no tempo $t + 1$;
- R é a recompensa imediata **esperada**[7], como resultado da ação a_t, depois de transicionar do estado s_t para o estado s_{t+1}.

A Figura 3.3 ilustra a relação entre esses parâmetros. A partir da escolha da ação com maior probabilidade P de retornar ao agente a recompensa esperada R, ele executa uma ação a_t que leva o Ambiente a um novo estado s_{t+1} e dá ao agente uma recompensa $\mathbb{R}_{t+1}$. Estes dois parâmetros realimentam o agente, para que ele defina quais são as próximas possíveis ações em $t + 2$.

Figura 3.3: Diagrama de blocos de um MDP genérico



Fonte: Autor

[7] A recompensa esperada **R** não é necessariamente igual a recompensa obtida, representada aqui por $\mathbb{R}$

Outro parâmetro que pode entrar na descrição genérica de um MDP é o fator de desconto γ, que relaciona o valor da sua recompensa no tempo. Limitado entre 0 e 1, o γ definirá o desconto de recompensas futuras no presente: quanto menor, menos o agente valoriza recompensas futuras.

A partir do treinamento, o agente aprende uma política de escolha de ações baseada no estado no qual ele se encontra a fim de maximizar a recompensa obtida, otimizando assim a tomada de decisões dado o ambiente.

Um tópico sempre em voga no meio de Aprendizado por Reforço é a relação entre *Exploitation* x *Exploration* (sem tradução). *Exploitation* se refere a aproveitar o conhecimento já obtido nas iterações prévias para tomar uma decisão melhor no presente. Já *Exploration* se refere a explorar o ambiente à procura de novas informações que potencialmente poderiam levar o agente a tomar melhores decisões no futuro. Há diversos meios para balancear estas duas linhas de ação, uma delas, utilizada neste trabalho, é a política ϵ-*greedy* [45] onde, a partir de um parâmetro ϵ, define-se a probabilidade de o agente escolher entre a ação com a probabilidade de ganhar a recompensa máxima naquele estado $(1 - \epsilon)$ ou uma ação aleatória dentre o espaço de todas as ações possíveis (ϵ). Assim, adiciona-se um grau de *Exploration* ao aprendizado do agente.

Devido a essas características, o Aprendizado por Reforço é amplamente utilizado em ambientes de tomada sequencial de decisões, como por exemplo, para controlar um carro autônomo ou ensinar a um robô como movimentar um membro.

O algoritmo que utiliza a descrição básica de conceito de Aprendizado por Reforço, é chamado de *Q-learning* [46]. Ele calcula a **qualidade** (*Q-Value*) de cada uma das ações do conjunto A tomada pelo agente em um determinado estado s do conjunto S. Essa qualidade é a recompensa esperada (R) caso aquela ação seja tomada pelo agente.

Inicialmente, temos uma tabela com um número de células igual ao espaço de Estados x Ações. Essa tabela contém o valor da qualidade de cada ação em cada estado, e as células são inicializadas com um valor padrão, geralmente zero. Conforme ações vão sendo executadas pelo agente, essa tabela é atualizada com os valores de recompensas obtidas $(\mathbb{R})$. Caso o agente passe pela segunda vez pelo mesmo estado e esco-

lha uma ação que já realizou, *i.e.* a célula referente a esta ação neste estado não contém o valor zero, a **recompensa esperada** retornada pelo algoritmo é o valor salvo na tabela, ou seja, a recompensa obtida anteriormente. A nova recompensa obtida agora pode ou não ser igual à recompensa esperada, e a partir dessa diferença, são feitas correções na tabela de *Q-values*.

Através da atualização da tabela e dos estados percorridos pelo agente, é criada uma política de tomada de decisão $\pi(A|s)$, que nada mais é do que a distribuição probabilística de cada umas das ações do conjunto A serem tomadas em um dado estado s.

Este algoritmo pode ser representado pela equação geral 3.7, onde $\mathbb{Q}$ representa esta tabela SxA.

$$\mathbb{Q}(S, A) = \mathrm{R} \tag{3.7}$$

Contudo, num domínio com um espaço de estados ou de ações muito grandes, a representação dos *Q-values* através de uma tabela é inviável e algoritmos mais simples não conseguem aprender de forma eficaz qual a melhor política de escolha de ações. Em 2013, Mnih *et al.* [14] apresentaram uma forma de generalizar aquele algoritmo através do uso de Redes Neurais Profundas, propondo o algoritmo *Deep Q-Learning Networks* (DQN), que resolve essa ineficácia.

Neste algoritmo, há algumas mudanças em relação a uma Rede Neural comum:

- *Experience Replay*: Adiciona-se um *Replay Buffer*, que salva as últimas experiências do agente e, para o treinamento, busca neste *buffer* um pacote de transições de forma aleatória. Essas experiências são salvas como tuplas da forma $(s_t, a_t, \mathbb{R}_{t+1}, s_{t+1})$. O número de experiências é fixo e varia conforme a aplicação.

- Duas Redes Neurais: Neste algoritmo, tem-se duas Redes Neurais: a *Online Network* (representada por θ), que é a rede otimizada e utilizada para a seleção das ações e a *Target Network* (representada por $\overline{\theta}$) que é uma cópia temporária da rede, que não é otimizada diretamente (mas apenas de tempos em tempos). A razão de se utilizarem duas redes é a seguinte: em cada passo

t do treinamento, o valor dos pesos da rede muda. Como são realizadas muitas alterações nestes valores com frequência, eles podem divergir. Desta forma, os pesos da *Target Network* são fixos e apenas atualizados periodicamente, tornando o treinamento mais estável.

As Redes Neurais são utilizadas para calcular a qualidade de cada uma das ações a_j do conjunto A dado um estado qualquer s, e elas são implementadas de forma que as suas camadas de saída tenham um número de neurônios igual ao número de ações possíveis j. Assim, calcula-se simultaneamente a qualidade de cada uma das ações, ao invés de executar o algoritmo por j vezes para fazer esse cálculo separadamente. Como a saída das redes são a qualidade de cada uma das ações (recompensa esperada por cada uma delas), estas são representadas pelas Equações 3.8 e 3.9, onde $\mathcal{R}$ representa o vetor de recompensas esperadas (*Q-values*) por cada uma das ações possíveis.

$$\mathcal{Q}_{\theta}(s, A) = \mathcal{R}_{\theta} \tag{3.8}$$

$$\mathcal{Q}_{\bar{\theta}}(s, A) = \mathcal{R}_{\bar{\theta}} \tag{3.9}$$

Seguindo aquela publicação [14], diversos outros autores apresentaram otimizações ao algoritmo original. Em 2017, Hessel *et al.* [13] criaram um novo algoritmo, denominado *Rainbow*, através de uma compilação destas otimizações e melhorias puntuais ao algoritmo original de 2013. Este foi a base para o desenvolvimento deste trabalho.

3.2.1 Algoritmo escolhido: *Rainbow*

A partir de diversas melhorias propostas para o algoritmo DQN e também a partir de conceitos mais antigos aplicáveis neste contexto, surge o algoritmo *Rainbow* [13], através da integração do algoritmo DQN com outras 6 otimizações, como definidas a seguir:

1) **DDQN** (*Deep Double Q-learning Networks*): No algoritmo original, o treinamento se dava a partir de uma variação do método de Gradiente Descendente, que visava minimizar a função perda ($\mathcal{L}$)

dada pela Equação 3.10. Onde $\mathbb{R}_{t+1}$ representa a **recompensa imediata obtida** após a ação tomada no tempo t,

$$\mathcal{L}(\mathbb{R}, \gamma, S, A) = [\mathbb{R}_{t+1} + \gamma_{t+1}\max(\mathcal{Q}_{\overline{\theta}}(s_{t+1},A)) - \mathcal{Q}_{\theta}(s_t,a_t)]^2 \tag{3.10}$$

Contudo, ao utilizarmos o valor máximo da função $\mathcal{Q}_{\overline{\theta}}$ na Equação 3.10, podemos superestimar o valor da recompensa de uma ação potencial na rede, o que prejudica o aprendizado. Por isso, a otimização DDQN, proposta por van Hasselt *et al.* em 2016 [48], visa contornar este problema através do uso da *Online Network*: calcula-se o valor da ação que leva a uma maximização da recompensa na Rede θ para só então usar-se esse valor na função $\mathcal{Q}_{\overline{\theta}}$. Isso resulta em uma nova função perda, definida pela Equação 3.11, onde $\text{argmax}(f(X))$ denota a função que retorna o argumento x do conjunto X que maximiza uma função $f(X)$, neste caso, a função $\mathcal{Q}_{\theta}(s_{t+1},A)$[8]:

$$\mathcal{L}(\mathbb{R}, \gamma, S, A) =$$

$$[\mathbb{R}_{t+1} + \gamma_{t+1}\mathcal{Q}_{\overline{\theta}}(s_{t+1},\underset{a'}{\text{argmax}}(\mathcal{Q}_{\theta}(s_{t+1},A))) - \mathcal{Q}_{\theta}(s_t,a_t)]^2 \tag{3.11}$$

2) ***Replay*** **Priorizado**: Durante o treinamento, o agente seleciona aleatoriamente um conjunto de transições possíveis do *Replay Buffer*. Contudo, idealmente, gostaria-se que fosse possível priorizar as transições com as quais tem-se mais a aprender. Esta otimização, proposta por Tom Schaul *et al.* em 2016 [49], adiciona pesos a todas as transições do *buffer*, sendo que cada **nova transição** é nele adicionada com peso máximo, criando-se assim um efeito de priorização. Essa solução não é perfeita, já que transições novas que nada ensinam à rede também acabam sendo priorizadas. De qualquer forma, o uso dessa otimização mostrou resultados positivos para o treinamento da rede.

3) ***Dueling Networks***: Para explicar este conceito, primeiro temos que decompor a Equação 3.7, cujo objetivo é calcular a qualidade

[8] Onde a' representa que o argumento que a função argmax(f(x)) retornará é uma ação a e não o estado s_{t+1}

de cada umas das ações a do conjunto A, num estado s, conforme mostra a Equação 3.12, onde a função $\mathcal{V}(s)$, chamada de Função Valor representa a qualidade associada a um estado s e a função $\mathcal{A}(A)$, chamada Função Vantagem (*advantage*) representa o incremento de qualidade obtido com cada ação a do conjunto A.

$$\mathcal{Q}(s, A) = \mathcal{V}(s) + \mathcal{A}(A) \tag{3.12}$$

O objetivo desta otimização, proposta por Wang *et al.* em 2016 [50], é que haja uma rede para calcular cada uma das partes da Equação 3.12. Assim, os cálculos de cada uma das funções são feitos separadamente e apenas são combinados para calcular a qualidade de cada ação na última camada comum as duas redes. Assim, é possível estimar o valor de cada estado s de forma mais robusta, desassociando-o de uma ação específica.

4) **Aprendizado Multi-etapas**: No algoritmo DQN original, dentro de um conjunto de transições selecionado uniformemente do *Replay Buffer*, escolhe-se apenas uma por vez para o treinamento da rede. Esta otimização, proposta por Andrew Barto *et al.* [45] na primeira versão do seu livro, publicado em 1998, procuram selecionar n transições consecutivas por vez e fazer uma estimativa da recompensa atual considerando mais de uma recompensa futura. Desse modo, pode-se representar a recompensa esperada no tempo t para qualquer ação a escolhida conforme a Equação 3.13:

$$R_t^{(n)} \equiv \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1} \tag{3.13}$$

Como resultado, pode-se redefinir a função perda descrita na Equação 3.11, através da variante Multi-etapas, apresentada na Equação 3.14, onde S'_{t+n} representa o conjunto formado pelos estados selecionados do *Replay Buffer* e varia de tamanho conforme o parâmetro n.

$$\mathcal{L}(\mathbb{R}, \gamma, S, A) =$$

$$\left\{\mathbb{R}_{t+1} + \left[\sum_{k=0}^{n-1} \gamma_t^{(k)} \mathcal{Q}_{\bar{\theta}}(S'_{t+n}, \underset{a'}{\text{argmax}}(\mathcal{Q}_\theta(S'_{t+n}, A)))\right] - \mathcal{Q}_\theta(s_t, a_t)\right\}^2 \tag{3.14}$$

Um valor de n escolhido apropriadamente para a aplicação pode acelerar o aprendizado. Essa definição no geral é empírica e feita na fase de otimização da estrutura da Rede.

5) Aprendizado por Reforço Distributivo: O algoritmo original trabalha com o conceito de **recompensa esperada** para otimizar a Rede Neural e a função perda basicamente define a distância entre o valor esperado e o valor-alvo, conforme apresentado na Equação 3.10. Nesta otimização, proposta em 2017 por Bellemare *et al.* [51], trabalha-se com a distribuição estatística da recompensa e não só de um valor esperado. Assim, a função procura minimizar não a distância entre dois valores, mas o erro distributivo[9]. Assim, a camada de saída da Rede Neural não tem o número de neurônios definido como o número de ações possíveis j, mas sim, como o tamanho discretizado das distribuições estatísticas da qualidade de cada uma das ações. Aqui é introduzido o conceito de **átomos (z)**, que representam o tamanho destas distribuições.

Outra adição que esta otimização traz é o uso da Função de Ativação *Softmax*, utilizada na saída da Rede Neural. Esta função tem o objetivo de transformar um vetor de valores X = $\{x_1, x_2, ..., x_n\}$ em um vetor de probabilidades P = $\{p_1, p_2, ..., p_n\}$. Antes de passar por essa função, alguns valores podem ser negativos, ou maiores do que um. Ao passar pela função, tem-se um vetor onde a soma de seus valores é igual a 1 (100%). Ela é representada matematicamente pela Equação 3.15 e tem o objetivo de transformar a magnitude de cada um dos átomos que representam um valor de qualidade para uma ação em um item de uma soma cujo valor se limita 1. A partir destes novos valores dos átomos, tem-se

[9] Erro distributivo é a distância entre duas distribuições [52]

a distribuição estatística discreta da qualidade daquela ação.

$$Softmax(X) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \tag{3.15}$$

6) **Redes Ruidosas**: Em ambientes onde a recompensa depende não de uma ação, mas de uma sequência correta de ações para ocorrer, a exploração do ambiente proveniente da política *ϵ-greedy* não é o suficiente. Sendo assim, uma abordagem que pode ser utilizada é a adição de Ruído Gaussiano às operações feitas entre neurônios[10].
Esta otimização, proposta em 2018 por Fortunato *et al.* [53], complementa a Equação 3.3, adicionando-se Ruído Gaussiano como mostra a Equação 3.16, onde ϵ^b e ϵ^w representam variáveis aleatórias e $\odot$ representa a operação de produto matricial.

$$y = (wx + b) + ((w \odot \epsilon^w)x + b \odot \epsilon^b) \tag{3.16}$$

Esta otimização tem o potencial de aumentar o grau exploratório (*exploration*) da rede, já que qualquer mudança no vetor de pesos W pode induzir uma mudança na política de escolha das ações a serem tomadas em cada uma das iterações.

3.2.2 Detalhes de implementação - algoritmo *Rainbow*

Estados, Ações e Recompensas

A definição de estados, ações e recompensas é um passo fundamental na utilização de técnicas baseadas em Aprendizado por Reforço, independente da aplicação. Na implementação original, onde a aplicação eram jogos eletrônicos de Atari, essa definição era a seguinte:

– **Estados:** Por se basear em processamento de imagem, aqui os estados eram definidos por cada um dos *frames* possíveis do jogo

[10] Conforme explicado na Seção 3.1.1, a relação entre dois neurônios é mapeada como uma série de operações lineares como a representada na Equação 3.3.

sendo avaliado, marcando assim a evolução do agente no ambiente.

– **Ações:** As ações possíveis para o agente foram definidas como todas as possíveis ações de um jogador humano, ou seja, eram limitas pelo próprio controle do jogo.

– **Recompensas:** Além da conclusão da fase/jogo no qual o agente estava, jogos onde não havia uma conclusão clara, mas sim uma pontuação cumulativa, a recompensa do agente era calculada a partir da diferença do campo de pontuação entre dois *frames* consecutivos.

Arquitetura da Rede Neural

A implementação original do algoritmo, por ser ligada a aplicação com imagens, considerava três Redes Convolucionais ligadas em sequência e duas Redes Ruidosas, uma para a Função Vantagem $\mathcal{A}(A)$ e uma para a Função Valor $\mathcal{V}(s)$.

Pela natureza do problema ser referente a processamento de imagens, a camada de entrada da rede tinha de computador todos os pixels que compunham um estado. Por este motivo, o número de neurônios utilizado na rede era exorbitante: a saída da terceira camada convolucional, por exemplo, tinha 3136 neurônios, enquanto as camadas ocultas implementadas como Redes Ruidosas tinham 512 neurônios cada.

A função de ativação escolhida foi a função ReLU, por trazer algumas vantagens:

1) Ao adicionarmos mais camadas em uma Rede Neural com funções de ativação menos esparsas (como a Função Tangente Hiperbólica) e aplicarmos o algoritmo de *Backpropagation* para que a rede aprenda, pode surgir o problema de *vanishing gradient*, onde o valor desse gradiente se aproxima de zero. A função ReLU mitiga este problema sendo menos densa, isto é, não limita os valores possíveis entre um intervalo pequeno (como é o caso da Função Tangente Hiperbólica, por exemplo).

2) A função ReLU é mais eficiente computacionalmente falando, já que a operação realizada em si é bem simples.

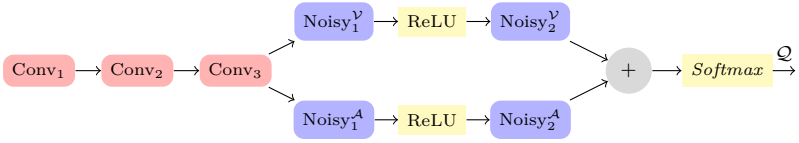
3) Na prática, Redes Neurais com função de ativação ReLU mostraram uma performance de convergência melhor que outras funções, como a Sigmoide, por exemplo [56].

A camada de saída da Rede Neural que computa a função $\mathcal{V}(s)$ tem um número de neurônios igual ao número de átomos que representarão a distribuição da qualidade de um estado. O algoritmo usa 51 átomos. Já na camada de saída da Rede Neural que computa a função $\mathcal{A}(A)$, este número é igual ao número de átomos multiplicado pelo número de ações possíveis naquele ambiente. Como em cada jogo utilizado como ambiente para validação do algoritmo há um número diferente de ações, o número de neurônios desta camada varia.

Ao juntar-se novamente as funções $\mathcal{A}(A)$ e $\mathcal{V}(s)$ para o cálculo da qualidade de cada ação na última camada geral, passa-se esta última por uma Função *Softmax*, para a criação da distribuição estatística discreta das qualidades, conforme descrito na otimização de Aprendizado por Reforço Distributivo, na Seção 3.2.1.

Assim, a organização de cada camada da Rede Neural pode ser representada pela Figura 3.4, onde Conv_n representa cada uma das camadas de Redes Convolucionais, $\text{Noisy}_n^{\mathcal{V}}$ representa uma camada de Rede Ruidosa para o cálculo da função $\mathcal{V}(s)$ e $\text{Noisy}_n^{\mathcal{A}}$ representa uma camada de Rede Ruidosa para o cálculo da função $\mathcal{A}(A)$:

Figura 3.4: Arquitetura da rede implementada para o algoritmo *Rainbow*



Fonte: Autor

3.2.3 Técnica utilizada para o treinamento

Pelas características do problema de verificação, que é o objetivo deste trabalho, não obteve-se um bom resultado ao utilizar-se técnicas tradicionais de treinamento (como será melhor explicado no próximo

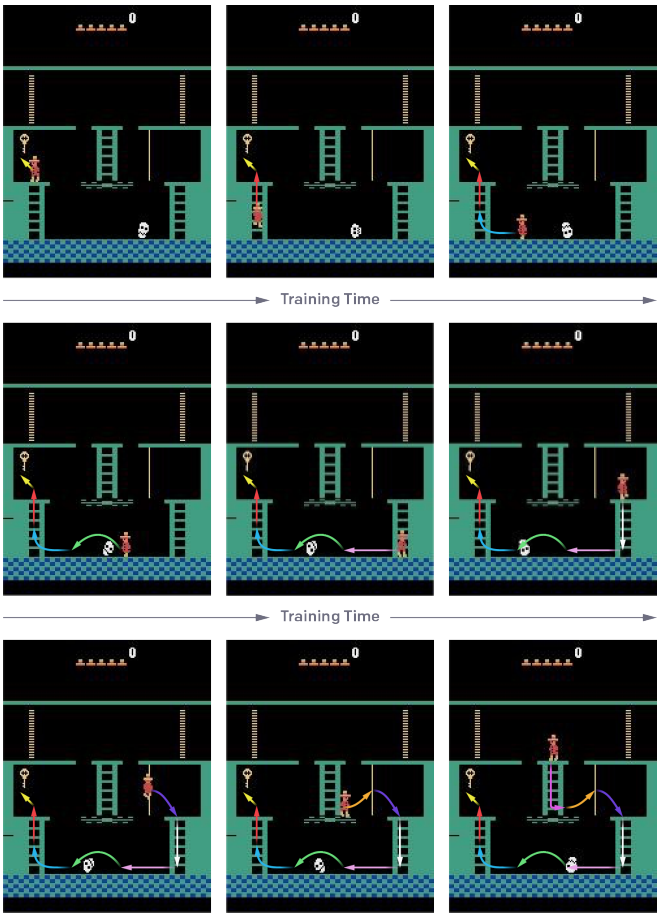
capítulo), por isso, foi necessário adaptar o processo de treinamento. A alternativa encontrada foi uma adaptação do trabalho de Tim Salimans *et al.* [15], tornando necessária sua revisão nesta seção.

Geralmente, o treinamento é quebrado em episódios (ou épocas), mas cada episódio tem o objetivo de treinar o agente no ambiente **completo**. Neste artigo, os autores apresentam o conceito de dividir o **próprio ambiente** em episódios, isto é, fazer com que o agente aprenda qual a melhor sequência de ações necessárias em cada época do treinamento, que representam apenas **parte** do ambiente.

Como exemplo, eles apresentam como quebraram um ambiente no qual se faziam necessárias diversas ações em uma certa ordem para que o agente obtivesse uma recompensa, em etapas do ambiente na **ordem reversa**, ou seja, o agente começaria o treinamento da última etapa (a qual daria a recompensa) e ele só teria de aprender qual ação correta a ser tomada naquele estado. Ao aprender o que fazer aí, o ambiente adicionava uma etapa anterior. Nesta nova época do treinamento, o agente teria que aprender duas ações para chegar à recompensa, sendo que já tinha o conhecimento da última. Este processo se repete até que o agente chegue no primeiro estado possível do ambiente, que seria o mais difícil caso tivéssemos começado por ele. Como o agente já aprendeu de outras experiências, ele torna-se menos difícil.

A Figura 3.5 ilustra o conceito criado, aplicado ao ambiente do jogo de Atari *Montezuma's Revenge*:

Figura 3.5: Exemplo do conceito de treinamento em épocas reversas aplicado ao ambiente do jogo de Atari *Montezuma's Revenge*



Fonte: Tim Salimans *et al.* 2018

Como podemos ver, o treinamento inicia do último estado possível desta fase em específico e vai voltando até o início da fase.

No próximo capítulo, serão abordados os detalhes de implementação do trabalho desenvolvido, tendo em mente os conceitos apresentados até agora e entrando mais a fundo na aplicação proposta: verificação de memória compartilhada coerente em chips *multicore*.

CAPÍTULO 4

Proposta de gerador baseado em DQN

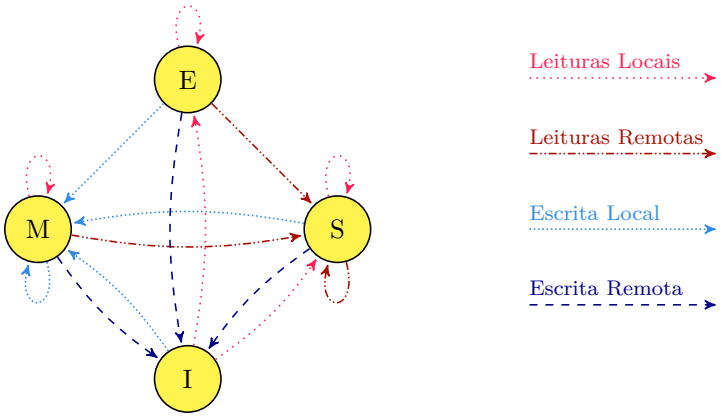
Este capítulo propõe a aplicação de conceitos de Aprendizado por Reforço (já apresentados na Seção 3.2) ao contexto de geração dirigida de testes para a verificação funcional de memória compartilhada coerente, durante a etapa de projeto de chips *multicore*. Para definir a recompensa, o conjunto de estados e o conjunto de ações a serem usados no aprendizado, a proposta explora conceitos publicados em recente trabalho sobre geração dirigida [8].

A Seção 4.1 revisa os principais conceitos daquele trabalho. A Seção 4.2 descreve o *framework* de verificação onde será inserido o módulo diretor. A Seção 4.3 descreve o gerador proposto. A Seção 4.3.1 define recompensa, estados e ações apropriadas ao contexto de geração dirigida e também as alterações propostas na arquitetura original da Rede Neural. A Seção 4.3.2 mostra as adaptações propostas para a etapa de treinamento do agente.

4.1 Conceitos de geração dirigida explorados na proposta

Um protocolo de coerência pode ser representado por uma máquina de estados finitos (FSM - *Finite State Machine*). Por exemplo, a Figura 4.1 ilustra a máquina de estados do protocolo MESI [57]. Os estados denominados M (*modified*), E (*exclusive*), S (*shared*) e I (*invalid*) são atributos de cada bloco em cache e indicam as permissões de um processador para acesso àquele bloco. O estado I indica que o processador não tem permissão de acesso ao bloco em cache, os estados E e S indicam que a permissão é de apenas leitura e o estado M indica que a permissão é de escrita e leitura. Transições entre estados são induzidas por requisições de permissão de leitura e escrita. Por exemplo, se um bloco está no estado S e o processador faz uma requisição de escrita, ela não é diretamente concedida pelo controlador daquela cache, mas este envia uma requisição de permissão para o mecanismo de coerência. Somente depois de ele receber essa permissão, o controlador de cache efetua uma transição para o estado M, provendo assim ao processador o acesso ao bloco em cache.

Figura 4.1: Máquina de estados finitos do comportamento do protocolo MESI



Fonte: Autor

A técnica de geração denominada de SCA (*steep coverage-ascent*)

[8] classifica as transições entre estados e propõe um modelo analítico de cobertura para dirigir a geração de testes, como explicado a seguir.

4.1.1 Classificação de transições entre estados

A técnica SCA distingue três classes de transições:

- **Classe 1:** Transições induzidas por eventos locais ativados pelo processador ou por outro controlador de cache privado localizado no domínio do mesmo processador no nível hierárquico imediatamente superior. Estas transições são resultados de colisões *intra-processor*, *i.e.* de um mesmo núcleo.
- **Classe 2:** Transições induzidas por eventos locais ativados por solicitações de processadores remotos. Estas transições são resultados de colisões *inter-processor*, *i.e.* de dois diferentes núcleos.
- **Classe 3:** Transições induzidas por eventos de substituição de um bloco na cache (*replacement*), ativadas pelo próprio controlador da cache.

Aquele trabalho [8] mostra que, ao induzir uma alternância entre essas classes, obtém-se maior cobertura em menor tempo. A alternância entre transições das Classes 1 e 2 resulta da exploração de restrições não convencionais, que são aplicadas a um gerador aleatório de testes, através de uma técnica denominada de *chaining* [27]. Por outro lado, a alternância entre uma transição da Classe 3 e uma transição das Classes 1 ou 2 pode ser induzida através de três parâmetros que restringem a geração aleatória de testes:

- $\mathbb{n}$**:** Número de operações de memória (leituras e escritas)
- $\mathbb{s}$**:** Número de variáveis compartilhadas
- $\mathbb{k}$**:** Número de conjuntos distintos da cache (*i.e.* posições de memória com o mesmo *index*) para os quais aquelas variáveis podem ser mapeadas

Para habilitar ou não transições da Classe 3, é feita uma limitação no espaço de valores dos parâmetros $\mathbb{s}$ e $\mathbb{k}$. Para cada valor de $\mathbb{s}$, restringe-se quais são os valores possíveis de $\mathbb{k}$ mantidos no espaço possível de geração: apenas aqueles para os quais $\mathbb{s}$ é um múltiplo de $\mathbb{k}$. Assim sendo, podemos definir $\mathbb{S}$ como o conjunto de valores que $\mathbb{s}$ pode assumir, conforme na Equação 4.1[1] e $\mathbb{K}$ como o conjunto de valores que $\mathbb{k}$ pode assumir dado um determinado $\mathbb{s}$, conforme a Equação 4.2. A Equação 4.3[2] define $\mathbb{N}$ como o conjunto de valores que $\mathbb{n}$ pode assumir.

$$\mathbb{S} = \{\mathbb{s}_{min} \leq \mathbb{s} \leq \mathbb{s}_{max} \wedge \mathbb{s} = 2^i \ \forall \ i \ \in \mathbb{N}\} \tag{4.1}$$

$$\mathbb{K}(\mathbb{s}) = \{\mathbb{k} \in [1, \mathbb{s}] \wedge \mathbb{s} \ \textbf{mod} \ \mathbb{k} = 0\} \tag{4.2}$$

$$\mathbb{N} = \{\mathbb{n}_{min} \leq \mathbb{n} \leq \mathbb{n}_{max} \wedge \mathbb{n} = 2^i \ \forall \ i \ \in \mathbb{N}\} \tag{4.3}$$

4.1.2 Modelo de cobertura

A associatividade da memória cache define como cada bloco da memória principal é mapeado para cada bloco da memória cache. Pode ser classificado como **mapeamento direto** ($\alpha = 1$), onde cada bloco da memória principal é mapeado para apenas um bloco da memória cache, **totalmente associativo** ($\alpha = M$, onde M representa o número de blocos na cache) onde cada bloco da memória principal é mapeado para qualquer bloco da memória cache ou **associativo *n-way*** ($\alpha = n$) onde há um número n de conjuntos com um número de blocos na cache, e cada bloco da memória principal é mapeado um conjunto na memória cache e pode ser alocado em qualquer bloco dado aquele conjunto associativo.

Com estes conceitos, a técnica SCA define analiticamente um modelo de cobertura de transições em função dos valores dos parâmetros

[1] $\mathbb{s}_{min}$ e $\mathbb{s}_{max}$ são definidos pelo usuário.
[2] $\mathbb{n}_{min}$ e $\mathbb{n}_{max}$ são definidos pelo usuário.

$\mathbb{n}$, $\mathbb{s}$ e $\mathbb{k}$. A cobertura de transições das Classes 1 e 2 é proporcional ao número de operações e inversamente proporcional ao número de variáveis compartilhadas como mostra a Equação 4.4.

$$TC_{1/2} \propto \frac{\mathbb{n}}{\mathbb{s}} \tag{4.4}$$

A cobertura de transições da Classe 3 é capturada em duas equações, dependendo do padrão de acesso à memória. No melhor caso, a cobertura é diretamente proporcional ao número de operações; no pior caso, a cobertura é diretamente proporcional ao número de variáveis compartilhadas; em ambos os casos a cobertura é inversamente proporcional ao número de conjuntos, conforme mostram as Equações 4.5 e 4.6, respectivamente.

$$TC_{3_1} \propto \frac{\mathbb{n}}{\mathbb{k}} \tag{4.5}$$

$$TC_{3_2} \propto \frac{\mathbb{s}}{\mathbb{k}} \tag{4.6}$$

Se denominarmos α como a associatividade da cache, para induzir a substituição de um bloco na memória cache, são necessárias pelo menos $\alpha + 1$ referências a variáveis compartilhadas distintas que são mapeadas a um mesmo conjunto associativo. Sendo assim, uma condição necessária para habilitar a substituição de blocos na cache é que:

$$\frac{\mathbb{s}}{\mathbb{k}} \geq \alpha + 1 \tag{4.7}$$

Da mesma forma, uma condição necessária para desabilitar a substituição de blocos na cache é que:

$$\frac{\mathbb{s}}{\mathbb{k}} < \alpha + 1 \Leftrightarrow \frac{\mathbb{s}}{\mathbb{k}} \leq \alpha \tag{4.8}$$

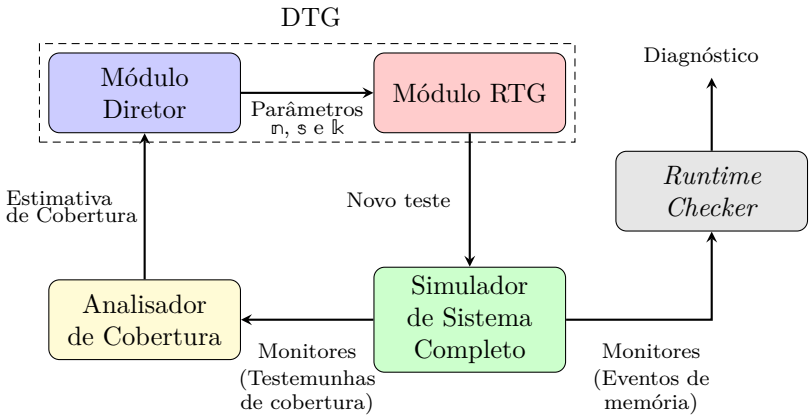
Sendo assim, pode-se habilitar ou desabilitar a ocorrência de *replacement* a partir da escolha apropriada dos parâmetros $\mathbb{s}$ e $\mathbb{k}$ dados os conjuntos $\mathbb{S}$ e $\mathbb{K}$.

4.2 O *framework* de verificação hospedeiro

A Figura 4.2 apresenta um diagrama de blocos que ilustra o *framework* de verificação proposto em [8]. O *framework* consiste de um gerador dirigido de testes denominado de DTG (*Directed Test Generator*), que sintetiza programas paralelos a serem executados em um simulador de uma representação do projeto do chip *multicore* sob verificação.

Durante a execução, um *runtime checker* monitora eventos de memória e verifica sua adequação ao comportamento esperado para o subsistema de memória compartilhada coerente. Além disso, um analisador de cobertura monitora as transições entre estados e envia uma estimativa de cobertura para o DTG. O DTG, por sua vez, consiste de um módulo diretor e um módulo de geração pseudo-aleatória de testes denominado RTG (*Random Test Generator*), sendo que o primeiro define os parâmetros usados pelo segundo para restringir a geração aleatória de testes.

Figura 4.2: *Framework* de verificação de memória compartilhada em chips *multicore* hospedeiro



Fonte: Autor

- **Módulo Diretor:** Este módulo tem o papel de restringir a geração de um novo teste a ser executado no simulador de sistema.

Ele não gera o teste em si, na realidade, ele apenas passa os três parâmetros ($\mathbb{n}$, $\mathbb{s}$ e $\mathbb{k}$) que tem o papel de restringir a aleatoriedade da geração de um novo teste para o Módulo RTG.

- **Módulo RTG:** Este módulo é o responsável por, a partir dos parâmetros recebidos, gerar programas de teste a serem simulados. No caso deste trabalho, o gerador em questão é o descrito em [27], baseado nas técnicas de *Chaining and Biasing*.

- **Simulador de Sistema Completo:** Este módulo representa o simulador comportamental Gem5 [19]. Enquanto os programas estão sendo simulados, monitores observam eventos de memória em cada um dos *cores*.

- **Analisador de Cobertura:** Eventos que servem como testemunhas de cobertura, são computados por este módulo, que retorna este dado ao Módulo Diretor, que levará isto em conta para gerar os novos parâmetros dos próximos testes.

- ***Runtime Checker*** Este módulo é o responsável por checar se os eventos de memória ativados pelos testes sendo simulados respeitam o Modelo de Consistência de Memória (MCM) da arquitetura alvo. Este é usado para indicar se um erro no modelo foi encontrado. Como ele é usado para medição de esforço para a detecção de erros artificias e este não é o objetivo deste trabalho, este módulo não está no projeto final.

O gerador aqui proposto será implementado dentro desse *framework* de verificação, através da substituição do Módulo Diretor por um novo módulo baseado nas otimizações propostas ao algoritmo DQN na implementação *Rainbow* [13].

4.3 Novo módulo diretor proposto

As principais contribuições deste trabalho, a serem descritas nas próximas seções, são: 1) a definição de recompensa, estados e ações apropriada ao contexto de verificação funcional de memória compartilhada coerente; 2) a adaptação da rede neural; e 3) a adaptação do mecanismo de treinamento. Para avaliar experimentalmente a adequação

dessas contribuições, reusou-se ao máximo a implementação do algoritmo *Rainbow* reportada em [13], fazendo-se apenas as alterações decorrentes desta proposta.

Esta parte do trabalho foi completamente desenvolvida na linguagem de programação Python e fez uso extensivo das bibliotecas e funções da plataforma *open source* PyTorch [58], que reúne diversas ferramentas para o desenvolvimento de aplicações utilizando técnicas de *Deep Learning*.

Nesta seção, primeiro serão apresentados os principais detalhes construtivos da Rede Neural e como ela se insere no contexto já apresentado na Figura 4.2. Em seguida, será descrito como conhecimentos dos trabalhos anteriormente realizados no ECL, como [8] e [27], foram utilizados para melhorar o desempenho desta implementação.

4.3.1 Detalhes de implementação - novo módulo diretor

Estados, ações e recompensa

O primeiro ponto a ser tratado antes mesmo de iniciar-se a implementação foi a definição de estados, ações e recompensas para o novo módulo diretor, conceitos fundamentais em uma aplicação baseada em Aprendizado por Reforço.

Em verificação funcional baseada em cobertura, o principal ponto que marca a evolução do agente no ambiente é a **cobertura** em si. Além disso, em uma aplicação de memória compartilhada, como [8], sabe-se que quanto maior o número de operações de memória por teste, maior é a probabilidade de que sejam cobertas transições não cobertas antes. Sendo assim, caso o agente tenha a informação do valor de π, ele pode direcionar qual a próxima ação a ser tomada de forma mais assertiva. Por exemplo: caso o último teste tenha sido executado com o valor de $\pi_{\max}$, não há sentido no agente escolher uma ação que aumente o valor de π atual[3].

Há diversas formas de passar esta informação ao agente. A escolhida nesta implementação foi a normalização do valor de π pelo valor de $\pi_{\max}$, assim o agente tem tanto a informação no π atual como também da sua distância até o $\pi_{\max}$ sem a necessidade da definição *a priori*

[3] O valor de $\pi_{\max}$ é definido pelo usuário.

de quais são esses valores, fazendo com que o aprendizado do agente seja válido para diferentes variações do ambiente. Com estas considerações, podemos definir o **espaço de estados** como apresentado nas Equações 4.9 e 4.10, onde $\mathbf{cv_n}$ representa a cobertura (*coverage*) de cada teste n:

$$S = \{s_1, s_2, s_3, ..., s_n\} \tag{4.9}$$

Onde um estado qualquer s_j é definido como a dupla

$$s_j = \left(\sum_{i=0}^{j} \mathrm{cv}_i, \frac{\mathbb{n}_j}{\mathbb{n}_{\max}} \right) \tag{4.10}$$

Inspirado em [8], as ações também visam reforçar a possibilidade de alternância entre as três classes apresentadas na Seção 4.1, assim, temos três ações possíveis no nosso **espaço de ações**, sendo que cada uma delas afeta o gerador de forma diferente, através da alteração dos parâmetros de geração $\mathbb{n}$, $\mathbb{s}$ e $\mathbb{k}$.

$$A = \{a_1, a_2, a_3\} \tag{4.11}$$

A primeira, a_1 visa aumentar o valor de $\mathbb{n}$ atual. Como, por padrão, trabalha-se com potências de 2 para definição do valor dos parâmetros $\mathbb{n}$, $\mathbb{s}$ e $\mathbb{k}$, a primeira ação dobra o valor de $\mathbb{n}$, como mostra a Equação 4.12.

$$a_1(\mathbb{n}) = 2 * \mathbb{n} \tag{4.12}$$

Já as ações a_2 e a_3 referenciam-se a, respectivamente, executar o próximo teste com *replacement* ativado ou desativado. Isso é feito através da manipulação dos parâmetros de geração $\mathbb{s}$ e $\mathbb{k}$, levando-se em consideração as Equações 4.7 e 4.8. Para a_2 o valor de $\mathbb{s}$ escolhido é valor máximo disponível. O valor de $\mathbb{k}$ é o valor mínimo dado aquele $\mathbb{s}$. O oposto ocorre para a ação a_3, ou seja, é escolhido o valor mínimo possível de $\mathbb{s}$ e o valor máximo de $\mathbb{k}$ dado aquele $\mathbb{s}$.

Além disso, para evitar repetições do mesmo valor de $\mathbb{s}$ **e** $\mathbb{k}$, três restrições são adicionadas as ações a_2 e a_3. A primeira (i) diz respeito a ação a_2. Toda vez que ela é executada, o valor de $\mathbb{k}$ escolhido para

aquele teste é retirado do conjunto $\mathbb{K}(\mathbb{s})$. Quando ela for executada novamente, se o conjunto $\mathbb{K}(\mathbb{s})$ é vazio, o valor de $\mathbb{s}$ é retirado do conjunto $\mathbb{S}$ antes da escolha do valor dos parâmetros.

As restrições (*ii*) e (*iii*) dizem respeito a ação a_3. A (*ii*) diz que, após a escolha de um valor $\mathbb{k}$, este é retirado do conjunto $\mathbb{K}(\mathbb{s})$. Quando a ação a_3 for executada novamente, se o conjunto $\mathbb{K}(\mathbb{s})$ é vazio, o valor de $\mathbb{s}$ é retirado do conjunto $\mathbb{S}$ antes da escolha do valor dos parâmetros.

A restrição (*iii*) diz que se o valor de $\mathbb{k}$ escolhido para um dado $\mathbb{s}_n$ for igual a 1, um novo valor de $\mathbb{s}$ será escolhido: $\mathbb{s}_{n+1}$ e, a partir deste, um novo valor de $\mathbb{k}$. Isso é feito porque $\mathbb{k} = 1$ representa o $\mathbb{k}$ mínimo dado aquele $\mathbb{s}$, o que tornaria a alteração nos valores dos parâmetros $\mathbb{s}$ **e** $\mathbb{k}$ mais próximos de uma execução dos testes com *replacement* ativado do que desativado, objetivo desta ação.

A restrição (*ii*) também é aplicada novamente nestes novos valores, ou seja, toda vez que restrição (*iii*) entrar em ação e o conjunto $\mathbb{K}(\mathbb{s})$ for vazio, um novo valor de $\mathbb{s}$ é escolhido.

Formalmente, podemos definir a ação a_2 pelo Algoritmo 1 e a ação a_3 pelo Algoritmo 2, onde a função $pop(x, X)$, representa a retirada do item x do conjunto X e a função $index(min(X))$ retorna o índice do menor item x do conjunto X.

Algoritmos que representam as ações a_1 e a_2

Algoritmo 1 Ação $a_2(\mathbb{S}, \mathbb{K})$

```
1: 𝕤 = max(𝕊)
2: if 𝕂(𝕤) = ∅ then
3:     pop(𝕤, 𝕊)
4:     𝕤 = max(𝕊)
5: 𝕜 = min(𝕂(𝕤))
6: pop(𝕜, 𝕂(𝕤))
7: return 𝕤, 𝕜
```

Algoritmo 2 Ação $a_3(\mathbb{S}, \mathbb{K})$

```
1: 𝕤 = min(𝕊)
2: if 𝕂(𝕤) = ∅ then
3:     pop(𝕤, 𝕊)
4:     𝕤 = min(𝕊)
5: 𝕜 = max(𝕂(𝕤))
6: j = 1
7: while 𝕜 = 1 do
8:     i = index(min(𝕊))
9:     𝕤 = 𝕊[i + j]
10:    if 𝕂(𝕤) = ∅ then
11:        pop(𝕤, 𝕊)
12:        𝕤 = min(𝕊)
13:    𝕜 = max(𝕂(𝕤))
14:    j = j + 1
15: pop(𝕜, 𝕂(𝕤))
16: return 𝕤, 𝕜
```

Depois que todas possibilidades de valores $\mathbb{s}$ e $\mathbb{k}$ forem percorridas, os conjuntos $\mathbb{S}$ e $\mathbb{K}(\mathbb{s})$ são reinicializados com os valores originais.

A **recompensa** do agente é a variação da cobertura total acumulada (representada por **CV**), assim, o agente aprende a priorizar as ações que lhe trarão uma maior variação na cobertura, otimizando a escolha das ações dado um estado para aumentar ao máximo a sua cobertura atual. Ela é definida conforme a equação 4.13

$$R = \Delta \mathrm{CV} \tag{4.13}$$

Modificações da rede neural

Como a natureza da aplicação desta monografia já trabalha diretamente com números, não há a necessidade das Redes Convolucionais utilizadas originalmente para quebrar a matriz que representava o *frame* de entrada em um vetor de valores de dimensão reduzida. Sendo assim,

a arquitetura aqui apresentada contém apenas as duas Redes Ruidosas.

A entrada da Rede Neural deve referenciar um estado. Sendo assim, só necessitam-se dois neurônios nessa camada, conforme apresentado na definição de estados na seção anterior.

Ainda pela natureza da aplicação, as camadas ocultas tem 10 neurônios cada uma, devido ao menor espaço de ações possíveis em relação a [13]. O número de átomos foi reduzido para 11.

Antes de fazer a soma entre os vetores da Função Vantagem ($\mathcal{A}(A)$) e da Função Valor ($\mathcal{V}(s)$), o vetor da Função Vantagem ($\mathcal{A}(A)$) com 33 itens é separado em três vetores de 11 itens, cada um representando a distribuição da vantagem de uma ação. Então, soma-se o vetor resultado da Função Valor ($\mathcal{V}(s)$) com cada um destes vetores da Função Vantagem ($\mathcal{A}(A)$). Esses três novos vetores são passados pela Função *Softmax*, para finalmente termos na saída da rede a distribuição estatística discreta da qualidade de cada uma das ações.

Matematicamente, estas operações podem ser representadas pelas Equações 4.14, 4.15 e 4.16

$$\mathcal{A}(A) = \{z_1^{\mathcal{A}}, z_2^{\mathcal{A}}, ..., z_{33}^{\mathcal{A}}\} \Rightarrow \begin{cases} \mathcal{A}(a_1) = \{z_1^{\mathcal{A}}, z_2^{\mathcal{A}}, ..., z_{11}^{\mathcal{A}}\} \\ \mathcal{A}(a_2) = \{z_{12}^{\mathcal{A}}, z_{13}^{\mathcal{A}}, ..., z_{22}^{\mathcal{A}}\} \\ \mathcal{A}(a_3) = \{z_{23}^{\mathcal{A}}, z_{24}^{\mathcal{A}}, ..., z_{33}^{\mathcal{A}}\} \end{cases} \tag{4.14}$$

$$\mathcal{V}(s) = \{z_1^{\mathcal{V}}, z_2^{\mathcal{V}}, ..., z_{11}^{\mathcal{V}}\} \tag{4.15}$$

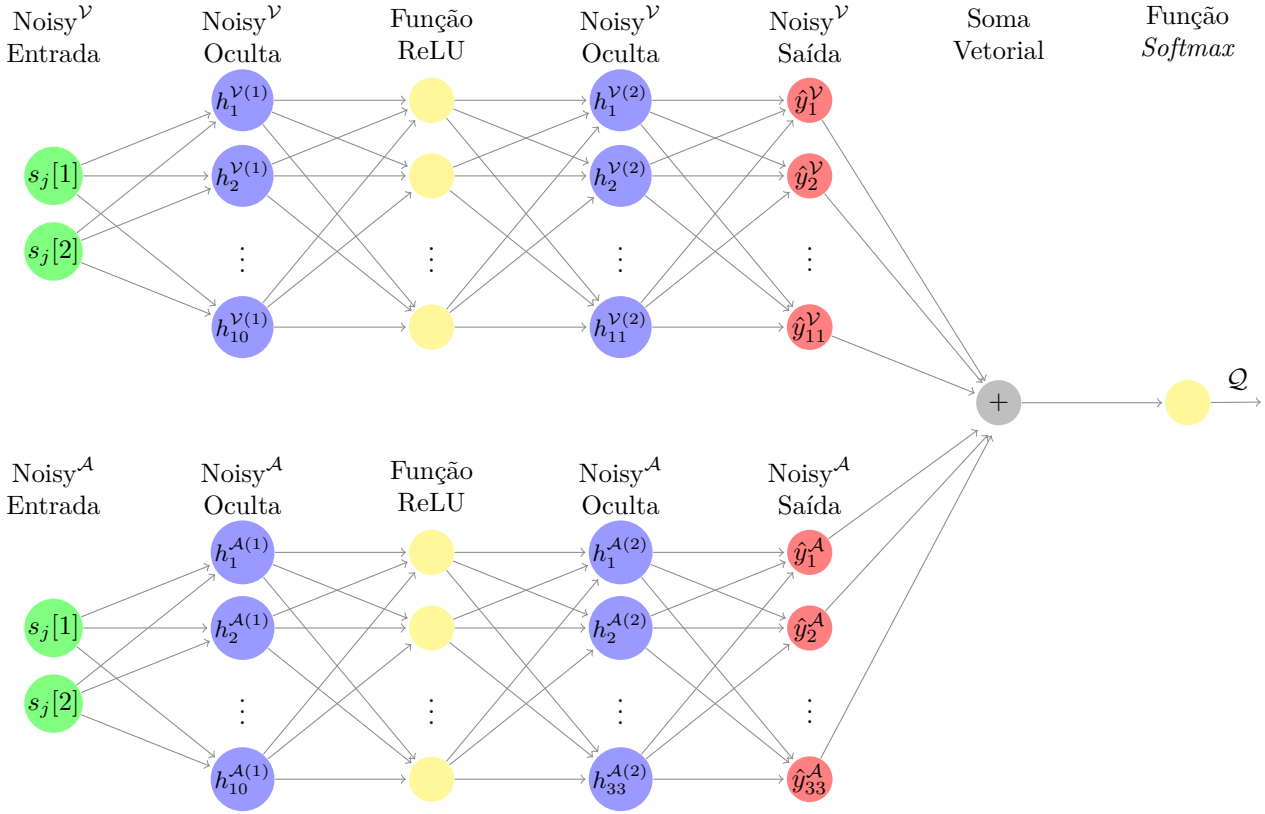
$$\mathcal{Q}(s, A) = \{q_1(s, a_1), q_2(s, a_2), q_3(s, a_3)\}$$

Onde

$$\begin{cases} q_1(s, a_1) = Softmax(\mathcal{V}(s) + \mathcal{A}(a_1)) \\ q_2(s, a_2) = Softmax(\mathcal{V}(s) + \mathcal{A}(a_2)) \\ q_3(s, a_3) = Softmax(\mathcal{V}(s) + \mathcal{A}(a_3)) \end{cases} \tag{4.16}$$

Depois destas considerações, podemos representar a arquitetura final da Rede Neural implementada através da Figura 4.3, onde a camada denominada **Soma Vetorial** representa as operações acima descritas.

Figura 4.3: Arquitetura final da Rede Neural implementada



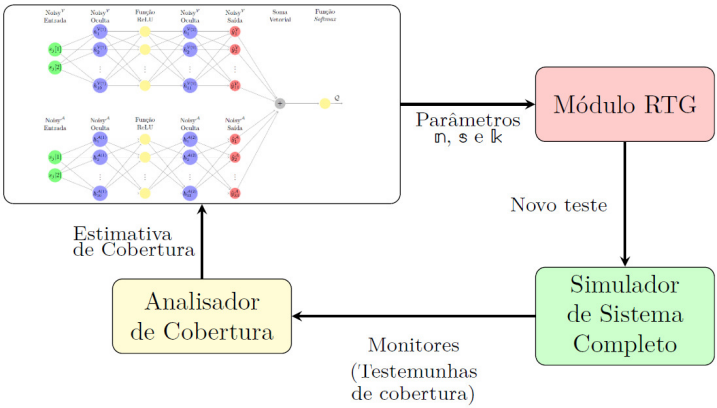
Fonte: Autor

Essa rede, depois de treinada, representa o novo módulo diretor proposto que passa os valores dos parâmetros de geração $\mathbb{n}$, $\mathbb{s}$ e $\mathbb{k}$ para o gerador de testes pseudo-aleatórios, que cria testes a serem executado no simulador comportamental.

Externo ao módulo diretor, há um Analisador de Cobertura, responsável por, através da observação dos testes realizados, manter registro da métrica de cobertura adotada. Alterando-se a métrica, o mesmo módulo diretor pode ser reutilizado, desde que sejam feitas as devidas alterações no Analisador de Cobertura. Neste trabalho, a métrica de cobertura adotada é a de **cobertura estrutural baseada em transições**, ou seja, faz referência direta ao exercício de transições entre estados de uma FSM, neste caso, das FSMs que definem o comportamento de controlador de cada nível cache privado de cada núcleo em um processador *multicore*, e também das FSMs do controladores do nível de caches compartilhadas entre todos os núcleos do processador. Estas são semelhantes a FSM apresentada na Figura 4.1. Diferente de uma métrica de cobertura funcional, por exemplo, que faz referência direta a uma funcionalidade do projeto, ao invés de sua estrutura. Além disso, por se tratar de uma métrica de cobertura da estrutura do protocolo de coerência, ela é independente da arquitetura utilizada.

O registro da cobertura volta à Rede Neural como parte de um novo estado, para que então, uma nova decisão de ação seja tomada. Com o *loop* fechado, o novo *framework* de verificação proposto é apresentado na Figura 4.4.

Figura 4.4: *Framework* completo de verificação de memória compartilhada em *multicores* proposto



Fonte: Autor

Na seção a seguir, será apresentado como foi utilizado o conhecimento dos trabalhos anteriores realizados no laboratório e reportados em [8] e [27] para tornar o treinamento mais eficaz para a Rede Neural implementada.

4.3.2 Modificações no mecanismo de treinamento

Uso dos geradores reportados na literatura

A técnica utilizada para o treinamento foi adaptada a partir do que foi relatado em [15], apresentada na Seção 3.2.3. O conceito adotado neste trabalho é inspirado no trabalho de Tim Salimans, mas foi adaptado para o nosso contexto.

Para separar-se transições quanto a um *proxy* de dificuldade, usou-se os dados experimentais de execução do gerador baseado nas técnicas de *Chaining and Biasing* [27]. O resultado desta execução foi analisado quanto ao número de vezes que cada uma das transições foi percorrida, e criou-se um novo parâmetro: o **número de grupos de treinamento**. Cada grupo contém o mesmo número de transições, mas com dificuldade crescente, ou seja, transições que foram cobertas menos

vezes. Este valor foi otimizado posteriormente nos testes do sistema completo.

Foi observado que ao começar-se o processo de treinamento pelas transições menos cobertas na execução do gerador proposto em [27], que representam a parte "final"do ambiente (como proposto em [15]), no geral o agente ficava preso nelas durante todo o treinamento. Por esta razão, invertemos a lógica proposta no artigo original: o ambiente continua sendo considerado dividido em épocas, mas o treinamento começa pelo "início"do ambiente, pelo grupo de transições que foram cobertas mais vezes pelo gerador. Depois, com o conhecimento destas, avança-se para um novo grupo de transições mais difíceis, *i.e.* que foram cobertas menos vezes. Dessa forma, o agente pode focar em aprender um grupo de transições por vez, com o conhecimento obtido a partir dos grupos anteriores.

Além disso, também foram criados dois novos parâmetros para o treinamento:

- ***Objective Attained Threshold*** **(ψ)**: Dado o conjunto de transições em um mesmo grupo, o agente deve cobrir cada uma delas ψ vezes para que elas sejam consideradas *aprendidas*.

- ***Zero Improvement Threshold*** **(ϕ)**: Em um mesmo grupo, se o agente fica por mais de ϕ vezes sem receber nenhuma recompensa a execução daquele grupo é reiniciada.

Independente do número de grupos, em cada época do treinamento o conjunto de transições a serem cobertas é diferente. Estas transições são atualizadas quando o agente avança para a próxima época, *i.e.* para o próximo grupo, depois de cobrir cada transição do grupo anterior ψ vezes.

Estrutura e parâmetros do treinamento

Para comprovar que o trabalho proposto podia ser generalizado para diferentes *designs*, o treinamento foi realizado considerando uma arquitetura de 8 *cores* e os testes foram realizado numa arquitetura de 32 *cores*. O tempo de treinamento foi de 10 horas consecutivas do agente

no ambiente separado em grupos.

Antes de passar para a apresentação da configuração dos experimentos e os resultados obtidos, vale aqui uma breve explicação dos principais parâmetros de treinamento, seja os adicionados especificamente para este trabalho, seja os já presentes no algoritmo *Rainbow* alterados no desenvolvimento deste projeto. Salvo dito o contrário, a escolha do valor do parâmetro foi feita em testes posteriores com o sistema completo a fim de otimizar os resultados obtidos.

Na lista a seguir, os últimos três itens fazem referência a parâmetros criados durante o desenvolvimento desta monografia, enquanto os outros são parâmetros já presentes na implementação original proposta em [13].

- **Multi-etapas (n)**: No algoritmo *Rainbow*, o parâmetro n era definido como 3. O valor utilizado neste trabalho é 2.
- **Fator de Desconto (γ)**: No algoritmo *Rainbow* γ era definido como 0.99. O valor utilizado neste trabalho é 0.9.
- ***Experience Replay Memory Capacity***: É a capacidade, *i.e.* o tamanho, do *Replay Buffer*. No algoritmo *Rainbow* este era definido como $1^{*}10^{6}$. O valor utilizado neste trabalho é 1200. Este valor foi escolhido a partir de um cálculo aproximado de quanto tempo o *Rainbow* levava para executar cada iteração da rede. Como cada interação do modelo proposto aqui leva mais tempo (por ter de passar pelo gerador e pelo simulador), o tamanho da memória foi menor.
- ***Target Update* (τ)**: Número de testes realizados antes de atualizar a *Target Network*. No algoritmo *Rainbow*, τ era definido como $32^{*}10^{3}$. O valor utilizado neste trabalho é 32. O mesmo cálculo aproximado de tempo feito para o parâmetro acima foi feito aqui também.
- ***Learning Rate* (η)**: Este parâmetro define o quanto os pesos são atualizados durante o treinamento, fazendo um balanço entre a influência de resultados recentes e resultados históricos. Este valor varia entre 0 e 1, sendo que 1 significa que o peso será alterado 100% do valor indicado pelo algoritmo de *Backpropagation*.

No algoritmo *Rainbow*, η era definido como $6.25^{*}10^{-5}$. O valor utilizado neste trabalho é 0.001.

- ***Batch Size***: Este parâmetro define quantos testes serão realizados antes de aplicar-se o algoritmo de *Backpropagation* para a atualização dos pesos da rede. No algoritmo *Rainbow* este era definido como 32. O valor utilizado neste trabalho é 4.

- ***Learn Start***: Este parâmetro define quantos testes serão realizados na rede antes de iniciar-se o treinamento de fato, a fim de preencher parte do *Replay Buffer*. No algoritmo *Rainbow* este era definido como $8^{*}10^{4}$. O valor utilizado neste trabalho é 256.

- ***Objective Attained Threshold*** (ψ): O valor final utilizado é 20.

- ***Zero Improvement Threshold*** (ϕ): O valor final utilizado é 100.

- **Número de Grupos de Treinamento**: O valor final utilizado é 15.

No próximo capítulo, será apresentada a estrutura experimental de validação do trabalho desenvolvido, assim como os resultados obtidos em comparação com os resultados de [7] e [8].

CAPÍTULO 5

Validação experimental

Este capítulo apresenta a estrutura de teste utilizada, assim como os resultados das validações experimentais. Depois da apresentação dos resultados, uma breve análise é feita, assim como um balanço dos pontos positivos e negativos do trabalho desenvolvido quando comparado a outros geradores de testes pseudo-aleatórios para verificação funcional de memória compartilhada coerente que, atualmente, representam o estado da arte.

5.1 Estrutura experimental

O simulador comportamental utilizado para realizar os experimentos foi o Gem5 [19]. A arquitetura considerada foi a SPARC [59] e a organização da memória cache adotada está representada na Tabela 5.1. O tamanho dos blocos nos três níveis de memória cache é o mesmo: 64 bytes e o protocolo de coerência utilizado é o protocolo diretório MESI, descrito na Figura 4.1, de três níveis (três níveis de memória cache).

Tabela 5.1: Organização da memória cache considerada nos experimentos

Nível	Privacidade	Tamanho	Associatividade
L1	Privada	4 KB	Mapeamento direto[1]
L2	Privada	64 KB	2-*way*[2]
L3	Compartilhada	2 MB	8-*way*[3]

Fonte: Autor

Os geradores utilizados como comparação foram o McVersi *Test Generator* [7], que será representado pela sigla MTG e o *Steep Coverage-Ascent Test Generator* [8], que será representado pela sigla SCATG. Já o trabalho desenvolvido nesta monografia será representado pela sigla DQNTG (*Deep Q-Learning Networks Test Generator*).

Em relação ao MTG, todos os parâmetros genéticos foram mantidos exatamente como propostos em [7] e, como a escolha do número de operações de memória por teste neste gerador é estático, ele foi definido como o valor máximo que os seus pares foram limitados para alcançar dinamicamente: 4 KiB (4096) operações.

Tanto o SCATG quanto o DQNTG tem os mesmos limites inferiores e superiores para os valores dos parâmetros $\mathbb{n}$, $\mathbb{s}$ e $\mathbb{k}$, conforme representado na Tabela 5.2, lembrando que o valor máximo de $\mathbb{k}$ depende do valor de $\mathbb{s}$ sendo utilizado, portanto o limite máximo do parâmetro $\mathbb{k}$ apresentado na Tabela 5.2 é o valor máximo absoluto, isto é, o valor máximo quando o parâmetro $\mathbb{s}$ é máximo.

[1] Significa que cada bloco da memória principal é mapeado para um único bloco na memória cache.

[2] Significa que cada bloco da memória principal é mapeado para um conjunto associativo da memória cache composto por dois blocos e pode ser salvo em qualquer um destes blocos.

[3] Significa que cada bloco da memória principal é mapeado para um conjunto associativo da memória cache composto por oito blocos e pode ser salvo em qualquer um destes blocos.

Tabela 5.2: Limites inferior e superior (absoluto) dos parâmetros de geração $\mathbb{n}$, $\mathbb{s}$ e $\mathbb{k}$

Parâmetro	Valor mínimo	Valor máximo
$\mathbb{n}$	1024	4096
$\mathbb{s}$	4	128
$\mathbb{k}$	1	128

Fonte: Autor

Para ambos os geradores MTG e SCATG foram executados 10 sequências de testes com a duração máxima de 1 hora. Com o resultado obtido de cada uma das 10 execuções de cada sequência, foi extraída a mediana, assim descartam-se quaisquer desvios pontuais (tanto execuções extraordinariamente boas como extraordinariamente ruins).

Já para o DQNTG, foi feito um treino prévio de 10 horas de duração, seguido por 10 execuções de sequências de testes limitadas a 1 hora. Com o resultado dessas 10 execuções, foi extraída a mediana, como nos outros dois geradores.

Para cada gerador, cada teste gerado é executado cinco vezes e a cobertura acumulada destas cinco iterações é retornada como resultado daquele teste.

Todos os experimentos de todos os geradores foram executados num computador munido com processador Intel Core i5-8250U (1.6 GHz, 4 GB RAM e 4 *cores*).

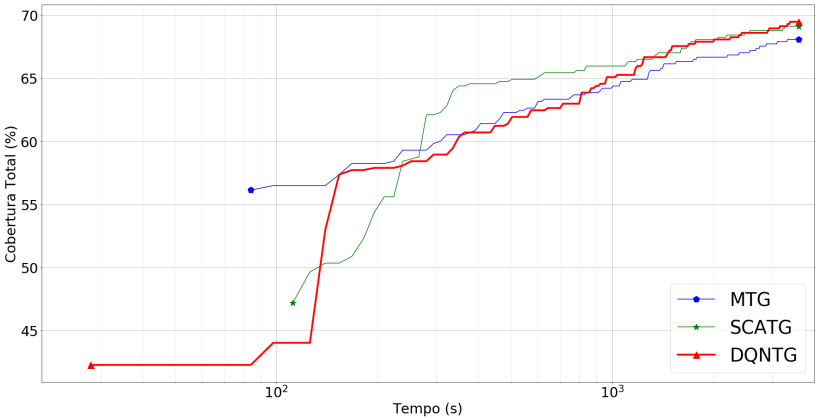
Como apresentado na Seção 4.3.2, o DQNTG foi treinado considerado uma arquitetura de 8 *cores*, já as sequências de testes para comparação com o MTG e o SCATG foram feitas considerando uma arquitetura de 32 *cores*. As execuções do MTG e do SCATG foram feitas apenas para arquiteturas de 32 *cores*.

5.2 Resultados experimentais

Diversos experimentos foram executados a fim de otimizar os parâmetros da rede e de treinamento. Na Figura 5.1, está apresentado o

resultado principal obtido com o trabalho, *i.e.* o resultado alcançado após os ajustes finos feitos a partir das execuções anteriores. O valor final dos parâmetros utilizados está descrito na Seção 4.3.2.

Figura 5.1: Evolução da cobertura no tempo DQNTG vs MTG e SCATG



Fonte: Autor

Após uma hora de sequências de testes, considerando a mediana de 10 execuções para cada um dos geradores, o resultado final da cobertura acumulada foi o seguinte:

Tabela 5.3: Resultados finais - 1 hora de testes

Gerador	Cobertura final
MTG	68.07%
SCATG	69.12%
DQNTG	**69.47%**

Fonte: Autor

Pode-se observar que, até um certo limite de cobertura (59.29%), o MTG necessita de menos tempo do que os outros dois geradores para

atingir este valor. Isso pode ser explicado pela exploração deste gerador de um tamanho variável de *threads*[4] para cada núcleo do processador sendo simulado, enquanto que na implementação atual do Módulo RTG utilizado nos geradores SCATG e DQNTG, todas *threads* tem a mesma quantidade de instruções.

Depois de um certo limite, contudo, esta fator já não é uma vantagem: o SCATG leva menos tempo do que os geradores DQNTG e MTG para atingir um valor maior de cobertura: ele atinge a marca de 64.56% 2.48 vezes mais rápido que o DQNTG e 2.88 vezes mais rápido que o MTG. Isso se deve ao fato do SCATG ser construído para forçar eventos de conflito que favorecem uma maior cobertura, enquanto os outros dois geradores, MTG e DQNTG, tem de aprender qual a melhor forma de fazer isso. Enquanto no primeiro isso se dá na forma de definir qual operações de memória levarão a uma condição de corrida, o segundo isso se dá na forma de aprender quais as ações para um dado momento de cobertura levarão a um recompensa ao agente, *i.e.* a uma variação na cobertura atual.

Como pode-se ver, o resultado final dos geradores DQNTG e SCATG foi semelhante, sendo que o primeiro se saiu ligeiramente melhor. Na mediana, neste período de 1 hora, o SCATG executou 81 testes e o DQNTG executou 67. Essa diferença se justifica pois mesmo testes com o mesmo número de operações ($\mathbb{n}$) podem levar um tempo diferente para serem concluídos, já que os outros parâmetros ($\mathbb{s}$ e $\mathbb{k}$) podem influenciar nesta duração também.

Um ponto positivo que vale ressaltar em relação ao MTG quando comparados aos outros dois geradores é a sua constância: a evolução da cobertura acumulada é constante, como mostra a Figura 5.1, diferente do comportamento dos geradores SCATG e DQNTG.

Apesar do resultado final melhor, é visível que o novo gerador é menos eficiente no início da execução da sequência de testes. Isso é reflexo do método utilizado para treinamento da Rede Neural. Ao dividir-se as épocas do treinamento em grupos de transições por uma *proxy* de dificuldade, os primeiros grupos contém transições que foram cobertas mais vezes. Por essa frequência, são transições que, no geral, não necessitam uma ordem correta de ações em um certo momento para

[4] Uma *thread* é uma sequência de instruções que compartilham um mesmo espaço de endereços.

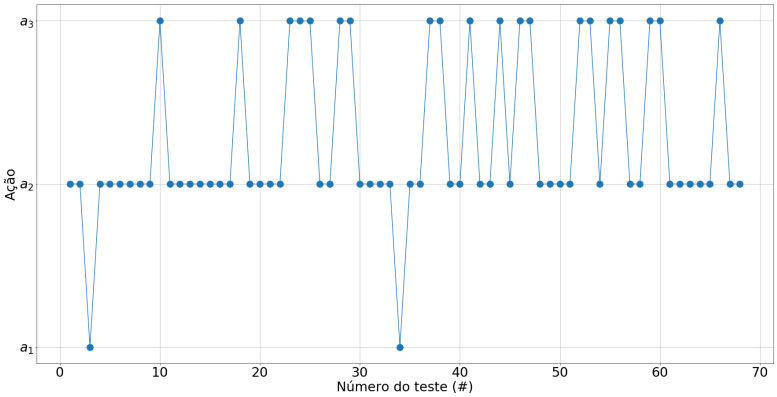
que sejam cobertas. Sendo assim, a rede não aprende necessariamente a forma mais eficaz de iniciar o processo de verificação, considerando que, inicialmente, a tomada de decisões é basicamente aleatória.

O que não se aplica ao final do processo. Com transições mais difíceis de serem cobertas e considerando aqui o papel dos dois parâmetros criados para o treinamento (ψ e ϕ) a rede se mostra assertiva na decisão de qual ação tomar, o que se reflete na constante evolução da cobertura acumulada no tempo, e, consequentemente, num melhor resultado final.

Este *trade-off* foi uma decisão de projeto considerando que, a longo prazo, é mais benéfico que cubram-se mais transições do protocolo de coerência - e, por conseguinte, aumente-se o potencial da descoberta de erros de projeto - do que cubram-se mais eficientemente transições simples no início do processo de verificação.

Ao analisar-se as ações tomadas pelo agente em uma execução típica do DQTNG, é possível gerar o gráfico apresentado na Figura 5.2, onde vemos como as ações foram distribuídas nos (neste caso) 68 testes gerados na 1 hora de execução do gerador.

Figura 5.2: Distribuição das ações tomadas pelo agente em uma execução típica do DQNTG



Fonte: Autor

Observando a figura, fica claro que o comportamento do agente se

assemelha ao algoritmo proposto por trás do funcionamento do módulo diretor de [8] (SCATG), variando testes que são executados com *replacement* ativado (ação a_2) e desativado (ação a_3) e, eventualmente, aumentando o valor de $\mathbb{n}$ (ação a_1), o que mostra como a Rede Neural aprendeu a imitar um comportamento que leva a bons resultados sem ser explicitamente programada para isso.

Isso é visível também quando comparamos a evolução dos dois geradores no gráfico da Figura 5.1. Lembrando que este gráfico está em escala logarítmica, ambos os geradores passaram a maior parte do tempo do experimento (entre 1100 e 3600 segundos) com uma evolução muito similar.

5.3 Considerações finais e melhorias futuras

As principais desvantagens do trabalho desenvolvido estão na dependência dos dados de execução de outro gerador para a categorização das transições que compõem o ambiente quanto a uma *proxy* de dificuldade, o que dificulta a replicação dos resultado obtidos, e no tempo de treino de 10h, ausente nos dois outros geradores que servem como comparação para esta implementação.

Em relação aos dados de execução do gerador baseado na técnica de *chaining and biasing* [27] utilizados no treinamento, apesar de haverem 15 grupos com o mesmo número de transições a serem cobertas por época, o agente nunca passou do grupo número 12. Depois de reiniciar neste grupo diversas vezes, ao final das 10 horas de treinamento, havia um início de progresso. A última interação do agente neste período foi a conclusão do grupo número 12 pela sexta vez.

Contudo, ao completar este grupo, a cobertura total acumulada era de apenas **66.67%**, o que significa que durante a execução dos testes com a rede já treinada, foram cobertas transições de grupos mais avançados, os quais nunca foram objetivo na fase de treinamento, como por exemplo a transição **L3_S_I+L2_GETX**[5], pertencente ao grupo 15,

[5]Esta transição representa um caso de concorrência entre duas ações do controlador da cache em relação a um mesmo bloco. A primeira, a ser priorizada, é a transição deste bloco em L3 do estado S (*Shared*) para I (*Invalid*), já a segunda, é uma requisição de alteração do estado do bloco em L2 para M (*Modified*).

coberta 20 vezes[6] por uma das execuções, por exemplo.

Isso mostra que a divisão de grupos baseadas em outro gerador (na execução do gerador baseado em *chaining and biasing* [27]) é essencial para guiar o processo de aprendizagem, mas, para melhorias a partir de um certo ponto, a *proxy* utilizada para a dificuldade encontrada por ele já não é precisa o suficiente para definir épocas de treinamento.

Portanto, apesar de importante para o início do projeto, a longo prazo a melhoria deixa de ser tão relevante. Alternativamente, poderia ser feito um primeiro treinamento sem considerar separação de grupos, a fim de ter uma execução do DQNTG contínua para reunir dados necessários de execução para fazer uma separação dos grupos, para treinamento de um segundo modelo[7]. Isso abre possibilidades de melhoria da eficiência do algoritmo, mas também resolve o problema de replicação do trabalho.

O segundo ponto, que diz respeito às 10 horas de treinamento é mais complexo. Utilizando técnicas de aprendizado, sempre haverá a necessidade de um período de treinamento anterior a execução dos testes. Na utilização de computadores mais potentes, as 10 horas de treinamento poderiam significar que os 15 grupos fossem de fato completados ou que este tempo fosse menor. Contudo, mesmo em outra máquina, esse tempo de treinamento ainda existiria.

Entretanto, apesar de este tempo não poder ser excluído do processo, ele pode ser mitigado por outras vantagens que esta técnica propõem. Uma delas, já clara, é que o resultados obtidos são melhores do que o dos seus pares e este treinamento só precisa ser feito uma vez. Caso executemos apenas um teste de verificação de uma hora, este tempo de treinamento representa 90.09% do tempo de esforço computacional total. Já se executarmos 100 testes de verificação de uma hora, o tempo de treinamento só representa 9.09% do esforço total, e o resultado dos 100 testes são ligeiramente melhores do que os obtidos utilizando as alternativas hoje reportadas na literatura. Ao aumentarmos o número de testes de verificação, essa porcentagem cai ainda mais, e o tempo de treinamento se torna mais insignificante, principalmente se o ganho em desempenho em relação a outras alternativas puder ser

[6] O número total de transições cobertas por essa execução foi de 8480 transições.

[7] Denominação dada à definição dos parâmetros da Rede Neural estabelecidos através da etapa de treinamento. Este modelo treinado é o que é utilizado para a execução da rede na fase de testes.

melhorado também.

Além disso, a implementação é independente da arquitetura utilizada. Um exemplo disso é que os resultados alcançados e reportados aqui são referentes a um treinamento realizado em uma arquitetura de 8 *cores* e execução de testes em uma arquitetura de 32 *cores*. Este conceito poderia ser levado ainda mais adiante, como a utilização do modelo treinado numa arquitetura SPARC (como é o caso deste trabalho) para a verificação de uma arquitetura x86[8], por exemplo.

Este conceito de independência da arquitetura utilizada é presente também no SCATG e no MTG. Portanto, para tornar a desvantagem das 10 horas necessárias ao treinamento menos custosa, melhorias ao algoritmo também deveriam ser realizadas, considerando que os resultados experimentais finais do SCATG e do DQNTG ainda foram próximos.

No próximo capítulo, de conclusão, será feito uma breve recapitulação do trabalho proposto, impactos do desenvolvimento do mesmo e próximos passos.

[8]Nome dado às arquiteturas da família de processadores Intel

CAPÍTULO 6

Conclusão

Neste trabalho, foi apresentada uma nova proposta de módulo diretor para um *framework* de verificação funcional de memória compartilhada coerente baseado no algoritmo *Rainbow*, criado sobre o algoritmo DQN (*Deep Q-learning Network*).

O trabalho se dividiu numa breve análise dos trabalhos correlatos, apresentação dos detalhes conceituais essenciais para o entendimento da implementação proposta, definições dos parâmetros utilizados, alterações propostas ao algoritmo original e resultados experimentais.

Após um período de 10 horas de treinamento, o novo algoritmo obteve resultados de cobertura ligeiramente melhores do que dois outros geradores (que podem ser considerados como representantes do estado da arte). Após uma hora de verificação, a cobertura acumulado do novo gerador atingiu 69.47% contra 69.12% e 68.07%, obtidas pelos geradores SCATG [8] e MTG [7], respectivamente.

Ao considerar os resultados pela perspectiva de Aprendizado de Máquina, este trabalho comprovou que o agente efetivamente aprendeu uma política de tomada de decisão que imita o comportamento de um dos trabalhos [8] sem ser explicitamente programado para isso.

Sob a perspectiva de verificação funcional, entretanto, para atingir

uma ligeira melhoria de cobertura num intervalo de uma hora de verificação, foi necessário um período preliminar de treinamento dez vezes maior (10 horas). Portanto, a viabilidade prática da técnica proposta só se caracterizaria se o esforço adicional de verificação (resultante da necessidade de treinamento) for desprezível frente a um tempo de verificação muito maior.

Sabendo-se que na prática o tempo de verificação levado para atingir uma cobertura próxima de 100% é consideravelmente maior do que uma hora (podendo levar dias a semanas), novas simulações mais longas deveriam ser executadas comparando-se os três geradores, para, além de se ter uma conclusão mais assertiva sobre qual deles é mais eficiente para o processo de verificação funcional de memória compartilhada coerente, também tenha-se uma maior clareza do peso do tempo de treinamento quando comparado ao esforço computacional total. Infelizmente, o tempo de simulação necessário para essa avaliação de viabilidade prática não é coerente com o tempo disponível para um trabalho de conclusão de curso.

Trabalhos futuros

Apesar deste trabalho ter comprovado a eficácia da técnica proposta, tornando-a uma candidata potencial para verificação funcional, a análise de sua viabilidade prática demanda a avaliação de sua eficiência, através de simulações mais longas, a qual requer experimentação mais extensiva do que a aqui apresentada.

A pesquisa preliminar colaborativa aqui reportada continuará no âmbito da dissertação de mestrado do aluno Nícolas Pfeifer, onde será realizada a análise de viabilidade prática da técnica utilizada e também uma nova proposta de conjunto de ações (considerando mais opções possíveis para o agente), aliada a otimizações ao algoritmo proposto visando a diminuição do tempo necessário para treinamento.

Referências bibliográficas

[1] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[2] Srinivas Devadas. Toward a coherent multicore memory model. *Computer, IEEE Computer Society*, pages 30 – 31, 2013.

[3] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, sixth edition, 2019.

[4] Intel. Processadores intel® core™ i3. `https://www.intel.com.br/content/www/br/pt/products/processors/core/i3-processors.html`, 2018.

[5] Intel. Intel® xeon® processor e7-8890 v4. `https://www.intel.com/content/www/us/en/products/processors/xeon/e7-processors/e7-8890-v4.html`, 2018.

[6] M. Martin, M. Hill, and D. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM, July 2012*, 2012.

[7] M. Elver and V. Nagarajan. Mcversi: A test generation framework for fast memory consistency verification in simulation. *IEEE Int. Symp. on High Performance Computer Architecture (HPCA)*, pages 618–630, 2016.

[8] Gabriel A. G. Andrade, Marlesson Graf, Nícolas Pfeifer, and Luiz C. V. dos Santos. Steep coverage-ascent directed test generation for shared-memory verification of multicore chips. *International Conference On Computer Aided Design, ICCAD '18*, 2018.

[9] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, 2003.

[10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, third edition, 2010.

[11] Geoffrey Hinton and Terrence J. Sejnowski. *Unsupervised Learning: Foundations of Neural Computation*. The MIT Press, 1999.

[12] Wang Qiang and Zhan Zhongli. Reinforcement learning model, algorithms and its application. *International Conference on Mechatronic Science, Electric Engineering and Computer*, 2011.

[13] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *Association for the Advancement of Artificial Intelligence (www.aaai.org)*, 2017.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *Advances in Neural Information Processing Systems - NIPS*, 2013.

[15] Tim Salimans and Richard Chen. Learning montezuma's revengefrom a single demonstration. *32nd Conference on Neural Information Processing Systems - NIPS 2018*, 2018.

[16] Junhwi Kim, Minhyuk Kwon, and Shin Yoo. Generating test input with deep reinforcement learning. *2018 ACM/IEEE 11th International Workshop on Search-Based Software Testing*, 2018.

[17] Niki Shakeri, Nastaran Nemati, Majid Nili Ahmadabadi, and Zainalabedin Navabi. Near optimal machine learning based random test generation. *2010 IEEE East-West Design & Test Symposium (EWDTS)*, 2010.

[18] Kai Arulkumaran. Rainbow: Combining improvements in deep reinforcement learning. `https://github.com/Kaixhin/Rainbow`, 2017.

[19] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.

[20] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. `https://github.com/gem5/gem5`, 2011.

[21] R. Gal, E. Kermany, B. Saleh, A. Ziv, M. Behm, and B. Hickerson. Template aware coverage - taking coverage analysis to the next level. *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017.

[22] K. Hsieh, L. Wang, W. Chen, and J. Bhadra. Learning to produce direct tests for security verification using constrained process discovery. *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017.

[23] D. A. Huffman. *The Synthesis of Sequential Switching Circuits*. PhD thesis, Massachusetts Institute of Technology, 1954.

[24] Phil McMinn. Search-based software testing: Past, present and future. *ICSTW '11 Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, 2011.

[25] Alex Groce. Coverage rewarded: Test input generation via adaptation-based programming. *2011 IEEE International Conference on Automated Software Engineering*, 2011.

[26] Bai Chen and DU Xiu-ting. Efficient reinforcement learning with trials-spanning learning scale for sequential decision-making. *2011 IEEE International Conference on Granular Computing*, 2011.

[27] Gabriel A. G. Andrade, Marleson Graf, and Luiz C. V. dos Santos. Chaining and biasing: Test generation techniques for shared-memory verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[28] Arthur Lee Samuel. Some studies in machine learning using the game of checkers. *IBM Journal*, 3(3), jul 1959.

[29] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[30] Corinna Cortes and Vladimir Vapnik. Support-vector networks. In *Machine Learning*, pages 273–297, 1995.

[31] David A. Freedman. *Statistical Models: Theory and Practice*. Cambridge University Press, 2009.

[32] Jan Salomon Cramer. The origins of logistic regression. Working Paper 2002-119/4, Tinbergen Institute, dec 2002.

[33] Naomi S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46:175–185, 1992.

[34] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society, Series B*, 39:1–38, 1977.

[35] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, second edition, 2008.

[36] Mihael Ankerst, Markus M. Breuniga, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. *ACM SIGMOD'99*, 1999.

[37] Seppo Linnainmaa. The representation of the cumulative rounding error of an algorithm asa taylor expansion of the local rounding errors. Master's thesis, University of Helsinki, 1970.

[38] Jack Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

[39] Warren S. Mcculloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletim of Mathematical Biology*, 5:115–133, 1943.

[40] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[41] Mark K Cowan. Machine learning. `https://github.com/battlesnake/neural/blob/master/examples/neural-networks-ebook.pdf`, 2013.

[42] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[43] Andrew Ng and Kian Katanforoosh. Cs229 lecture notes - deep learning. `http://cs229.stanford.edu/notes/cs229-notes-deep_learning.pdf`, 2018.

[44] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5), 1957.

[45] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning:An Introduction*. The MIT Press, 2017.

[46] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, 1989.

[47] Moustafa Alzantot. Deep reinforcement learning demysitifed (episode 2) — policy iteration, value iteration and q-learning. `https://medium.com/@m.alzantot/deep-reinforcement-learning-demysitifed-episode-2-policy-iteration-value-iteration-and-q-978f9e89ddaa`, 2017.

[48] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence - AAAI-16*, 2016.

[49] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *International Conference on Learning Representations - ICLR 2016*, 2016.

[50] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. *Proceedings of The 33rd International Conference on Machine Learning*, 2016.

[51] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *Proceedings of the 34th International Conference on Machine Learning - ICML 2017*, 2017.

[52] Massimiliano Tomassoli. Distributional rl - simple machine learning. `https://mtomassoli.github.io/2017/12/08/distributional_rl/`, 2017.

[53] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Matteo Hessel, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. *International Conference on Learning Representations - ICLR*, 2018.

[54] Arthur Juliani. Simple reinforcement learning with tensorflow part 4: Deep q-networks and beyond. `https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df`, 2016.

[55] Ziad Salloum. Double q-learning, the easy way. `https://towardsdatascience.com/double-q-learning-the-easy-way-a924c4085ec3`, 2018.

[56] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems Conference - NIPS*, 2012.

[57] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *Proceedings of the 11th annual international symposium on Computer architecture - ISCA '84*, 1984.

[58] Open Source. Pytorch. `https://pytorch.org/`.

[59] Robert B. Garner, Anant Agrawal, Fayé Briggs, Emil W. Brown, David Hough, Bill Joy, Steve Kleiman, Steven Muchnick, Masood Namjoo, Dave Patterson, Joan Pendleton, and Richard Tuck. The scalable prcessor achitecture (sparc). *COMPCON Spring 88 33th IEEE Computer Society International Conference*, 1988.