Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Gabriel Baptista
Hélio Hideki Assakura Moreira

Reinforcement Learning using Parallel Computing

São Paulo
Junho de 2017

Reinforcement Learning using Parallel Computing

Monografia final da disciplina
MAC5742/0219 – Introdução à Programação
Concorrente, Paralela e Distribuída.

Supervisor: Prof. Dr. Alfredo Goldman vel Lejbman

São Paulo
Junho de 2017

Sumário

1 Introdução

O objetivo deste documento é formalizar os conteúdos de aprendizado por reforço usando computação paralela, os quais já foram apresentados na aula de Computação Paralela e Concorrente para toda a turma.

Para tal, é necessário preparar todo um ambiente previamente, por isso, passa-se primeiramente pela definição de Processo de Decisão Markoviano e Aprendizado por Reforço até chegar na parte paralela ligada em si, mostrando alguns exemplos de sua utilização.

2 Processo de Decisão Markoviano(Markov Decision Process - MDP)

Quando o ambiente é probabilístico(estocástico), uma ação a leva a um estado s com probabilidade P(s | a) , de forma sequencial.

Então, um MDP é definido por um conjunto de estados $s \in S$, um conjunto de ações $a \in A$ e uma função de transição P(s,a,s'). Para cada tupla (s,a,s'), existe uma função de recompensa associada a ela, R(s,a,s').

Um MDP leva o nome de Markov associado a ele, pois o estado presente depende apenas do anterior, não importando os estados mais antigos e nem os possíveis estados futuros, desta forma, podemos escrever:

$$P(\mathrm{S}_{t+1} = \mathrm{s'} \mid \mathrm{S}_t = \mathrm{s}_t, \mathrm{A}_t = \mathrm{a}_t, \mathrm{S}_{t-1} = \mathrm{s}_{t-1}, \mathrm{A}_{t-1} = \mathrm{a}_{t-1}, ..., \mathrm{S}_0 = \mathrm{s}_0) = P(\mathrm{S}_{t+1} = \mathrm{s'} \mid \mathrm{S}_t = \mathrm{s}_t, \mathrm{A}_t = \mathrm{a}_t) \quad (1)$$

2.1 Política

Num problema envolvendo MDPs, deseja-se transitar nos estados em busca de um objetivo, para tal, utiliza-se um sequência de ações que leva a um plano ótimo, chamado de política.

Uma política ótima $\pi^* : S \longrightarrow A$ é uma política que maximiza o valor de recompensa recebido a cada estado.

2.2 Valor de Desconto

Para o cálculo da política ótima é necessário considerar o tempo, é preferível ganhar a recompensa no momento atual do que no futuro, então, define-se uma constante para isso, $\gamma \in [0, 1)$, que representa um valor de desconto para as recompensas.

2.3 Valor de Iteração

Uma vez definidos os conceitos acima, avança-se para o conceito do cálculo das recompensas e valores de cada estado. Para tal, o valor de iteração é dado por:

$$V^{\pi}(S) = R(S) + \gamma \sum_{S'} P(S' \mid S, \pi(S)) V^{\pi}(S') \quad (2)$$

2.4 Equação de Bellman

A partir do valor de iteração, constrói-se a equação de Bellman que serve para atualizar os valores em questão até haver uma convergência, a ideia é tomar 2 passos, o primeiro passo é escolher a primeira ação corretamente e o segundo é manter as escolhas ótimas, ficando a fórmula da seguinte forma:

$$V^*(S) = \max_A \sum_{S'} P(S' \mid S, \pi(S))[R(S, A, S') + \gamma V^*(S')] \quad (3)$$

3 Aprendizado por Reforço (Reinforcement Learning - RL)

Estando em um ambiente desconhecido, a cada estado, como nos MDPs, o agente recebe uma recompensa a cada ação tomada, que servirá como *feedback* e com isso ele deve aprender de forma que maximize o recebimento dessas recompensas.

Modela-se o ambiente para que ele se comporte como um MDP, porém, não é conhecido a priori, a função de transição P e nem a função de recompensa R, então, o agente, como sugere o nome, deve aprender as mesmas de acordo com as experiências anteriores.

Existem dois tipos de RL, o *online* e o *offline*, porém, como o foco deste documento não é o estudo do RL em si, será estudado apenas características gerais e comuns a ambos.

A ideia de estimar as funções anteriores de acordo com cada experiência requer um custo computacional muito alto, então a melhor forma de resolver esse problema é estimar a função Valor V(*) diretamente, para isso, existem algoritmos e o mais utilizado e que irá ser abordado aqui é o *algoritmo Q-Learning*.

3.1 Algoritmo Q-Learning

Antes de qualquer coisa, é necessário definir o conceito de *Q-Valor*, que é o valor que soma a recompensa e o valor da transição para todos estados anteriores com determinada ação:

$$Q(S,A) = \sum_{S'} P(S' \mid S, \pi(S))[R(S,A,S') + \gamma V(S')] \tag{4}$$

Substituindo (4) em (3), temos a Equação de Bellman escrita como:

$$V^*(S) = \max_A Q^*(S,A) \tag{5}$$

E pode-se ainda definir a operação *Q-value update* para cada par (S,A):

$$Q^{k+1}(S,A) = \sum_{S'} P(S,A,S')[R(S,A) + \gamma \max_A Q^k(S,A)] \tag{6}$$

O algoritmo *Q-Learning* consiste em fazer *Q-value updates* para cada par (S,A), porém, como não é conhecido P e R, computa-se uma aproximação dos valores de Q durante a fase de aprendizagem. Para uma experiência de transição (s,a,s',r), temos:

$$Q(S,A) \approx r + \gamma \max_{a1} Q(s',a') \tag{7}$$

Mas, como deseja-se uma medição sobre todas experiências, computa-se o valor Q da seguinte forma:

$$Q(S,A) = (1-\alpha)Q(S,A) + \alpha[r + \gamma \max_{a1} Q(s',a')] \tag{8}$$

Onde α é a taxa de aprendizagem e $\alpha \in [0,1]$.

Com isso, conclui-se que o algoritmo consiste em aprender a função Q*(S,A) ao invés de V*(S) e portanto, a política ótima é dada por:

$$\pi^*(S) = \arg\max_A Q^*(S,A) \tag{9}$$

Como visto em [8], o pseudocódigo a seguir retrata a abordagem que acabou de ser descrita, e este será utilizado para os testes e exemplos que virão nas próximas sessões.

Initializa com Q(s,a) arbitrário
Observa o estado atual **s**
Repita
    seleciona uma ação **a** e executa **a em s**
    recebe o reforço **r**
    observa o novo estado **s'**
    atualiza Q(s,a) :
        $Q(s,a) \leftarrow (1-\alpha)\, Q(s,a) + \alpha\, [r + \gamma \max_{a'} Q(s',a')]$
    $s \leftarrow s'$

Figura 1: Algoritmo Q-Learning

4 Aprendizado por Reforço Paralelo (Parallel Reinforcement Learning - PRL)

Segundo [5], aprendizado paralelo (ou Parallel Learning, PL) é um paradigma em RL em que diversos agentes agrupam suas experiências enquanto, de forma concorrente, aprendem o problema, melhorando a performance e diminuindo os tempos de convergência. É tipicamente usado para acelerar a convergência de problemas de RL com agente único. [5] também introduz o conceito de Multi Agent Reinforcement Learning (MARL), que segundo os autores, diferentemente do PL, é usado para examinar estratégias competitivas ou cooperativas e novos comportamentos. Em PL, agentes influenciam seus comportamentos por meio da partilha de informação, e aprendem em diferentes instâncias do mesmo problema, enquanto no MARL, os agentes interagem diretamente entre si, aprendendo na mesma instância (tradução nossa).

Para exemplificar uma aplicação do PRL, escolhemos um artigo publicado em 2015 em um Workshop de mobilidade e controle de tráfego baseados em agentes, de título *Parallel Reinforcement Learning for Traffic Signal Control*. Nele, é apresentado um novo método de PRL para controle de sinal de tráfego, e a partir de experimentos, os autores mostram que seu trabalho, que usa múltiplos agentes, supera a performance da abordagem com agente único. O artigo sugere usar até 4 agentes, divididos em 2 tipos: mestre e escravos. Eles transmitem suas experiências para uma matriz Q, e o mestre pode usar as informações coletadas para decidir sua ação no próximo intervalo de tempo. Os resultados examinados são retirados do agente mestre. Para cada agente, os estados são definidos pela equação:

$$s = [P_c, PTE, QL_1, ..., QL_n] \tag{10}$$

Em que (P_c) é a fase atual, (PTE) é o tempo decorrido na fase e (QL_i) é o tamanho da fila de carros na fase `i`.

A função de recompensa é dada por:

$$R(s, a, s') = CWT_s - CWT_{s'} \tag{11}$$

Com CWT_s e $CWT_{s'}$ sendo o tempo de espera cumulativo da fila de veículos nos estados `s` e `s'`, respectivamente.

A demanda `D` de carros usada na simulação é dada pela função

$$D(t) = \begin{cases} b + \frac{h \times t}{i} & t < \frac{e}{2} \\ b + h \times (\frac{e}{2 \times i} - 1 - \frac{t - 0.5 \times e}{i} & otherwise \end{cases} \tag{12}$$

Sendo `b` o fluxo base, `e` o tamanho do episódio e `h` o tamanho do aumento da demanda a cada intervalo `i`.

Como podemos ver na figura 2, ao usar múltiplos agentes, houve uma diminuição de até 7.79 % no tempo de espera médio (AWT), o tamanho médio das filas (AQL) ficaram até 5.64 % menores e o número médio de estados visitados (ANSV) cresceu até 27.53 %. Apesar de ser um trabalho inicial, os autores se

mostraram otimistas quanto aos resultados, dizendo que podem futuramente melhorar ainda mais a performance do algoritmo.

Experiment	AWT (s)	% Reduction AWT	AQL	% Reduction AQL	ANSV	% Increase ANSV
2 Phase, 1 agent	41.45	-	12.74	-	6,170.02	-
2 Phase, 2 agents	39.84	3.88 %	12.37	2.90 %	6,874.23	11.41 %
2 Phase, 3 agents	38.66	6.73 %	12.23	4.00 %	7,425.38	20.35 %
2 Phase, 4 agents	38.29	7.62 %	12.12	4.87 %	7,868.37	27.53 %
3 Phase, 1 agent	85.52	-	15.06	-	13,001.19	-
3 Phase, 2 agents	82.45	3.59 %	14.77	1.93 %	13,326.58	2.50 %
3 Phase, 3 agents	80.40	5.99 %	14.42	4.25 %	15,035.03	15.64 %
3 Phase, 4 agents	78.86	7.79 %	14.21	5.64 %	16,322.52	25.55 %

Figura 2: Resultado dos experimentos de [5]

Muitos algoritmos usados em RL realizam cálculos de matrizes e vetores, e vários deles são altamente paralelizáveis. Temos como exemplo o algoritmo *Least-Squares Policy Iteration - LSPI*[6], que recebe como entrada um conjunto de amostras do ambiente e realiza uma aproximação de iteração de uma política, usando a função `LTSDQ` para calcular a função dos valores de ação-estado durante a etapa de valoração de política. As imagens 3, 4 e 5 mostram a implementação dos algoritmos. Em especial, a implementação de `LSTDQ BLAS` usa 5 funções, das quais:

- *scal* - Multiplica um vetor por um escalar
- *axpy* - Soma um vetor com outro multiplicado por um escalar
- *gemv* - Realiza a multiplicação vetor-matriz
- *ger* - Multiplica 2 vetores
- *dot* - Faz o produto escalar de 2 vetores

Todas essas funções são facilmente paralelizáveis, e são realizadas diversas vezes, melhorando o desempenho de algoritmos de RL. Assim, podemos ver que várias partes de um programa de RL podem ser paralelizadas, seja usando múltiplos agente ou mesmo realizando operações com matrizes e vetores.

Algorithm 1 LSPI

Given:
- D - Samples of the form (s, a, r, s')
- k - Number of basis functions
- ϕ - Basis functions
- γ - Discount factor
- ϵ - Stopping criterion
- w_0 - Initial policy

1: $w' \leftarrow w_0$
2: **repeat**
3: $w \leftarrow w'$
4: $w' \leftarrow \text{LSTD}Q(D, k, \phi, \gamma, w)$
5: **until** $(\| w - w' \| < \epsilon)$
6: **return** w

Figura 3: Algoritmo LSPI

Algorithm 2 LSTDQ

Given:
- D - Samples of the form (s, a, r, s')
- k - Number of basis functions
- ϕ - Basis functions
- γ - Discount factor
- w - Current policy

1: $B \leftarrow \frac{1}{\delta} I$ // $(k \times k)$ matrix
2: $b \leftarrow 0$ // $(k \times 1)$ vector
3: **for all** $(s, a, r, s') \in D$ **do**
4: $B \leftarrow B - \frac{B\phi(s,a)\left(\phi(s,a) - \gamma\phi(s', \pi(s'))\right)^T B}{1 + \left(\phi(s,a) - \gamma\phi(s', \pi(s'))\right)^T B\phi(s,a)}$
5: $b \leftarrow b + \phi(s, a)r$
6: **end for**
7: $\tilde{w} \leftarrow Bb$
8: **return** $\tilde{w}$

Figura 4: Algoritmo LSTDQ

Algorithm 3 LSTDQ BLAS Implementation

Given:
- All input should be explicitly passed into GPU memory
- D - Samples of the form (s, a, r, s')
- k - Number of basis functions
- ϕ - Basis functions
- γ - Discount factor
- w - Current policy

$B \leftarrow scal(I, \delta)$ // $(k \times k)$ matrix
$b \leftarrow 0$ // $(k \times 1)$ vector
for all $(s, a, r, s') \in D$ **do**
  $x \leftarrow scal(\phi(s', \pi(s')), \gamma)$
  $x \leftarrow axpy(\phi, x, -1)$
  $x \leftarrow scal(x, -1)$ // Now $x = \phi(s, a) - \gamma\phi(s', \pi(s'))$
  $y \leftarrow gemv(B, \phi(s, a))$
  $z \leftarrow gemv(x, B)$
  $num \leftarrow ger(y, z)$ // Calculate the numerator, see Equation (4.3)
  $denom \leftarrow 1 + dot(\phi(s, a), z)$ // Calculate the denominator, see Equation (4.3)
  $\Delta B \leftarrow scal(num, \frac{1}{denom})$
  $B \leftarrow axpy(\Delta B, B, -1)$
  $\Delta b \leftarrow scal(\phi(s, a), r)$
  $b \leftarrow axpy(\Delta b, b)$
end for
$\tilde{w} \leftarrow gemv(B, b)$
return $\tilde{w}$

Figura 5: Algoritmo LSTDQ BLAS

5 Exemplos

Segundo [7], o uso de reinforcement learning obteve sucesso em diversas aplicações, como controle de robôs, cálculo de rotas de elevadores, telecomunicações e jogos como backgammon, damas e Go[14]. Essa última aplicação tem se tornado muito famosa devido ao desenvolvimento do `AlphaGo`, da Google, que usa principalmente técnicas de aprendizado de máquina e RL para jogar Go.

Começamos essa sessão com dois exemplos simples mostrando como RL pode ser paralelizado, e terminamos com `AlphaGo`, como já explicado no parágrafo anterior.

5.1 Pac-Man

O jogo do Pac-Man não paralelo e que não utiliza RL, consiste num jogador controlando um agente amarelo no formato de pizza(sem um pedaço), com o objetivo de comer todas a bolas que se encontram num labirinto, enquanto evitam quatro fantasmas que escapam de uma prisão. Nos quatro cantos do labirinto, tem bolas maiores que ao comê-las permitem ao pac-man caçar os fantasmas ao invés de ser caçado por um período de tempo.

O jogo utilizando RL, consiste em modelar o labirinto com os fantasmas e o pac-man como um MDP, onde cada estado considera a posição que o pac-man e os fantasmas estão bem como as bolas que permanecem no campo. Para tal, fazendo uso de RL, consegue-se criar um agente que na maior parte dos casos consegue resolver esse problema utilizando o algoritmo *Q-Learning*, de forma que o agente aprende a verificar a posição dos fantasmas de forma quase que instantânea, porém esse processo é feito de forma sequencial.

Pois bem, tendo definido o ambiente como um MDP e a parte de RL pronta, pode-se modelar o jogo de forma sequencial com uma pequena alteração, sincronizando a verificação dos fantasmas para com o agente. Desta forma, há um ganho na eficiência do problema, pois num caso genérico do jogo com n fantasmas, para um n grande, a parte sequencial de verificação a cada estado ficaria pesada.

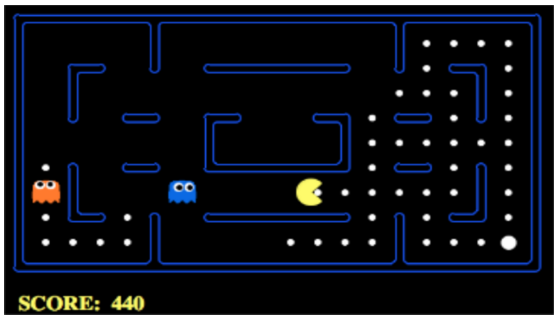

Figura 6: Pacman com n = 2

5.2 Grid

O jogo consiste em uma demonstração do algoritmo *Q-Learning* de forma simplificada, onde o jogador está num tabuleiro com duas salas terminais uma com prêmio positivo e outra com prêmio negativo, o restante do tabuleiro são salas com probabilidades de seguir em uma das direções ou em salas vazias, onde o jogador nunca poderá estar nela.

Utilizando RL, de forma análoga, o ambiente é modelado como um MDP e o jogador a cada mudança de sala, altera a probabilidade do estado e ação anterior, de forma que aumenta-se a probabilidade para um movimento que é considerado promissor e diminui-se a mesma, caso contrário.

Paralelizar o jogo consiste em sincronizar essa mudança de sala e atualização de probabilidades, simula-se o jogador sincronizando todas ações possíveis para cada estado e com isso ele terá uma certa rapidez em convergir para uma boa probabilidade ou não e decidir o caminho para chegar a sala terminal.

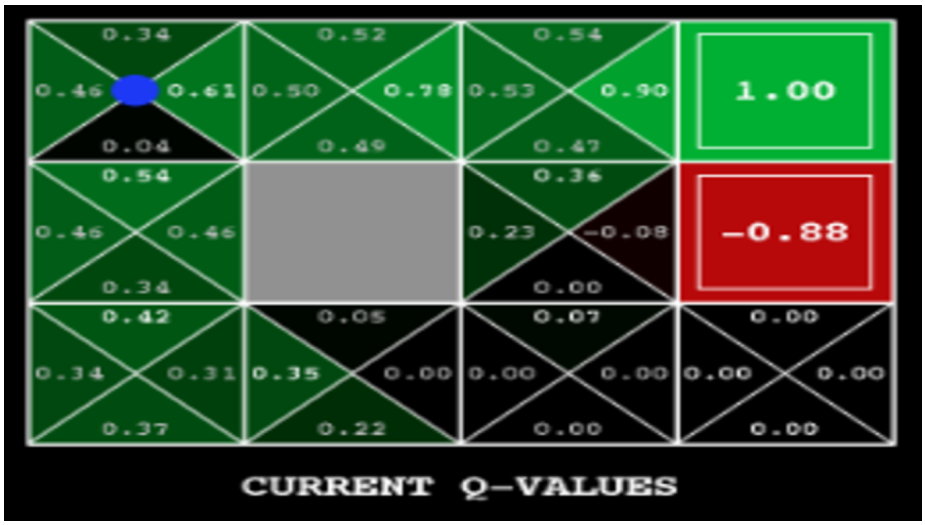

Figura 7: Uma instância do GridWorld, onde as salas terminais são as que possuem um quadrado dentro

5.3 AlphaGo

O AlphaGo é, de acordo com [9], o primeiro programa de computador a derrotar um jogador profissional humano de Go, o primeiro a derrotar o campeão mundial e, talvez, o melhor jogador de todos os tempos (tradução nossa).

Em [9], é posto que o simples uso de uma `Search tree`[11] não conseguiria lidar com o volume de possibilidades que o jogo proporciona. Em [10], diz que para o jogo de xadrez, há cerca de 10^{120} nós na árvore. Para isso, o AlphaGo usa uma `Monte Carlo tree search`, ou árvore de busca de Monte Carlo. Nela, a ideia é simular diversas partidas. Cada simulação começa de um estado de

uma partida, e termina quando algum dos jogadores ganhou. No começo, as ações são tomadas de forma aleatória, mas vários dados são guardados, como o número de visitas de cada nó, e quantas vezes o ganhador passou por dado nó.

Essas informações então são consideradas em futuras simulações. O que torna mais interessante é que esse método não requer a supervisão de jogadores reais. Além da MCTS, o AlphaGo usa `Redes neurais profundas`[12]. Essas redes recebem o estado atual do tabuleiro e uma rede neural chamada *"policy network"* seleciona a jogada, enquanto a rede *"value network"* tenta prever o ganhador do jogo.

O conceito de RL também é utilizado no AlphaGo. Primeiramente, o programa visualizou jogos amadores de alto nível, para desenvolver um estilo de jogo semelhante ao humano. Então, realizou diversas partidas consigo mesmo, aprendendo pelo processo de RL.

Em 2016, ano em que o artigo [10] foi publicado, o AlphaGo rodava em 48 CPUs e 8 GPUs, e a versão distribuída usava 1202 CPUs e 176 GPUs. A versão `AlphaGo Lee`, usada para jogar contra Lee Se-dol, considerado o melhor jogar de Go do mundo, usava 50 TPUs[13], distribuídas. O resultado foi 4-1 a favor do programa. A última versão, chamada `AlphaGo Master`, usava uma única máquina com uma nova versão da TPU (v2). Ela foi usada na *Future of Go Summit*, vencendo 60 jogadores profissionais de Go e segundo a DeepMind, desenvolvedora do AlphaGo, com 3 pedras de vantagem.

6 Conclusão

RL utilizando computação paralela ainda é uma área que está no início, porém com um potencial muito grande de crescimento, esse conceito já é encontrado em projetos grandes com o `AlphaGo` e controle de robôs, porém, paralelizar projetos não é uma tarefa fácil por conta das linguagens e lógicas utilizadas nos mesmos.

A tendência é que cada vez mais com um aumento nos estudos e testes na área de aprendizado de máquina, programas que são paralelizáveis se tornem paralelos de fato, conseguindo assim um ganho de performance e que sejam tão utilizáveis quanto os originais.

Referências Bibliográficas

[1] KLEIN, D.; ABBEEL, P., Lecture 8: MDPs I. Disponível em: http://ai.berkeley.edu/lecture_slides.html. Acesso em: 20 jun. 2017.

[2] KLEIN, D.; ABBEEL, P., Lecture 9: MDPs II. Disponível em: http://ai.berkeley.edu/lecture_slides.html. Acesso em: 20 jun. 2017.

[3] MARKOV decision process. In: "Wikipédia: the free encyclopedia". Disponível em: ¡https://en.wikipedia.org/wiki/Markov_decision_process¿ Acesso em: 20 jun 2017.

[4] MAUÁ, D. D., Sequential Decision Making, Disponível em: https://www.ime.usp.br/~ddm/mac425/aulas/mdp.pdf. Acesso em: 20 jun. 2017.

[5] MANNION, P.; DUGGAN, J.; HOWLEY, E. Parallel Reinforcement Learning for Traffic Signal Control. In: INTERNATIONAL WORKSHOP ON AGENT-BASED MOBILITY, TRAFFIC AND TRANSPORTATION MODELS, METHODOLOGIES AND APPLICATIONS (ABMTRANS), 4., 2015, Londres. Artigo, Londres:Elsevier, 2015. p. 956-961

[6] GOERINGER, T. MASSIVELY PARALLEL REINFORCEMENT LEARNING WITH AN APPLICATION TO VIDEO GAMES. Ago. 2013. 70f. Dissertação (Mestrado em Ciências) - Case Western Reserve University, Cleveland, Ohio. 2013.

[7] REINFORCEMENT learning. In: "Wikipédia: the free encyclopedia". Disponível em: ¡https://en.wikipedia.org/wiki/Reinforcement_learning¿ Acesso em: 20 jun 2017.

[8] DE BARROS, N. L., Reinforcement Learning I. In: "MAC0425/5739 Inteligência Artificial - PACA 1sem2017". Disponível em: ¡http://paca.ime.usp.br/pluginfile.php/107189/course/section/17629/Aula-ReinforcementLearning-2017.pdf¿ Acesso em: 20 jun 2017.

[9] The story of AlphaGo so far, Disponível em: https://deepmind.com/research/alphago/. Acesso em: 20 jun. 2017.

[10] BURGER, C. Google DeepMind's AlphaGo: How it works. 16 Mar. 2016. Disponível em: https://www.tastehit.com/blog/google-deepmind-alphago-how-it-works/. Acesso em: 20 jun. 2017.

[11] SEARCH tree. In: "Wikipédia: the free encyclopedia". Disponível em: ¡https://en.wikipedia.org/wiki/Search_tree¿ Acesso em: 20 jun 2017.

[12] DEEP neural networks. In: "Wikipédia: the free encyclopedia". Disponível em: ¡https://en.wikipedia.org/wiki/Deep_learning#Deep_neural_networks¿ Acesso em: 20 jun 2017.

[13] TENSOR processing unit. In: "Wikipédia: the free encyclopedia". Disponível em: ¡https://en.wikipedia.org/wiki/Tensor_processing_unit¿ Acesso em: 20 jun 2017.

[14] GO (game). In: "Wikipédia: the free encyclopedia". Disponível em: ¡https://en.wikipedia.org/wiki/Go_(game)¿ Acesso em: 20 jun 2017.