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

Victor Aliende da Matta

Jogando RTS com Aprendizado de Reforço Profundo

São Paulo
Dezembro de 2018

Jogando RTS com Aprendizado de Reforço Profundo

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Denis Deratani Mauá

São Paulo
Dezembro de 2018

Resumo

Neste trabalho, introduzimos a área de Aprendizado de Reforço Profundo. Tomamos um jogo de estratégia em tempo real como ambiente de exemplo, chamado MiniRTS (desenvolvido para pesquisa), que consiste de dois jogadores controlando várias unidades para ganhar recursos, construir estruturas e destruir as estruturas do inimigo. Estudamos algoritmos baseados em gradiente de política, como o REINFORCE, Actor-Critic e A3C junto com Redes Neurais Convolucionais. Implementamos o algoritmo A3C, modelamos nosso agente como uma Rede Neural Convolucional de duas cabeças, uma determinando a política enquanto a outra fazia o trabalho de crítico e testamos nosso agente contra duas estratégias fixas, obtendo 53% e 67% de taxa de vitória.

Palavras-chave: Aprendizado de Reforço, Aprendizado Profundo, Jogos de estratégia em tempo real.

Abstract

In this work, we introduce the field of Deep Reinforcement Learning. We took a real time strategy game as our showcase environment, called MiniRTS (developed for research), that consists of two players constructing an arrangement of units to harvest resources, build structures and destroy their opponent's structures. We study policy gradient algorithms, like REINFORCE, Actor-Critic and A3C together with Convolutional Neural Networks. We implement the A3C algorithm, modelling our agent as a Convolutional Neural Network with two heads, one defining the policy while the other works as the critic and we test our agente against two fixed strategies, obtaining 53% and 67% winning rate.

Keywords: Reinforcement Learning, Deep Learning, Real Time Strategy Games.

Sumário

Capítulo 1

Introdução

Esse trabalho se propõe a ser uma introdução a Aprendizado de Reforço, enviesado para a compreensão de alguns resultados mais recentes na área [Arulkumaran *et al.* (2017)] que, seguindo a tendência de outras áreas dentro de Aprendizado de Máquina [LeCun *et al.* (2015)], utilizam redes neurais para aproximação de funções.

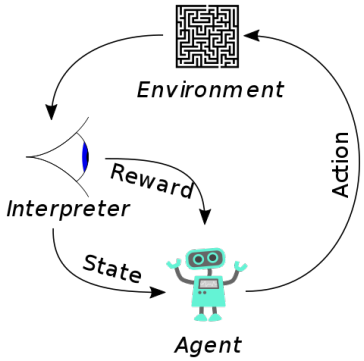

Figura 1.1: *Ilustração de Aprendizado de Reforço*

Aprendizado de Reforço (AR) é uma das grandes áreas que compõe Aprendizado de Máquina [Sutton e Barto (2011)], onde são estudados algoritmos que descrevem o comportamento de *agentes* em um *ambiente*, buscando o maior *retorno* por suas ações (ilustrado na figura 1.2).

Essa definição geral admite aplicação em uma grande quantidade de problemas [Kaelbling *et al.* (1996)], necessitando somente de um ambiente onde o algoritmo possa eficientemente agir e observar a consequência de suas ações. A princípio, não é necessário que o algoritmo seja treinado no mesmo ambiente que ele é ultimamente aplicado, por exemplo, em Robótica, onde o treinamento do algoritmo em um cenário real seria lento e apresentaria riscos para o equipamento, é comum utilizar simulações computacionais do ambiente (que naturalmente introduz outra série de desafios). O maior obstáculo portanto da aplicação de algoritmos de Aprendizado de Reforço tende a ser a quantidade de recursos computacionais necessários para o treinamento.

Pesquisa na área de Aprendizado de Reforço comumente adota jogos eletrônicos como o

ambiente para testar seus algoritmos [Tian *et al.* (2017)], já que estes proporcionam desafios complexos e bem definidos para a análise e comparação de algoritmos, em um ambiente totalmente controlável. Dentro destes, os jogos de estratégia em tempo real (RTS) se destacam por poder envolver um espaço de ação grande e mutável, grande conjunto de estados, temas de planejamento, informação imperfeita e recompensas distantes para as ações dos agentes. Conforme essa tendência, nesse trabalho utilizaremos como estudo de caso um jogo RTS bem simples, criado para a pesquisa de algoritmos de AR, isto é, desprovido de gráficos complexos, animações, e preocupações com interface de usuário, estabilidade de rede, entre outros elementos comuns em jogos comerciais.

Figura 1.2: *Captura de tela de Starcraft 2, um dos jogos RTS mais populares*

Um jogo RTS tem a estrutura básica de dois jogadores controlando um conjunto de unidades, se movendo, atacando, juntando recursos e construindo estruturas, com o objetivo final de destruir a(s) estrutura(s) do inimigo. Descrevemos mais profundamente a dinâmica do nosso ambiente no primeiro capítulo.

Como resultado principal, seguimos a abordagem de [Tian *et al.* (2017)]. Assim, modelamos nosso agente com uma Rede Neural Convolucional e utilizamos subida de gradiente com o algoritmo A3C (*Asynchronous Advantage Actor-Critic*) para otimizá-la, que por sua vez é uma variação do algoritmo REINFORCE, calculando uma estimativa estatística do gradiente a cada iteração do processo de otimização.

Testamos nossos agentes contra duas estratégias estabelecidas em [Tian *et al.* (2017)], onde foram reportadas 68% e 63% de taxa de vitória. Em nossos experimentos, obtemos 53% e 67% de taxa de vitória, nos limitando a métodos canônicos de treinamento e modelos mais simples que os utilizados no artigo.

Capítulo 2

O Problema

2.1 Aprendizado de Reforço

Geralmente, os problemas da área são formulados como Processos de Decisão de Markov (MDP) finitos [Arulkumaran *et al.* (2017)]. Um Processo de Decisão de Markov é uma quíntupla (S, A, T, R, γ) [Sutton e Barto (2011)], onde:

- S é o conjunto de estados (que representa o ambiente)
- A é o conjunto de ações que o agente pode tomar
- $T : S \times S \times A \to [0, 1]$ representa a probabilidade de um novo estado s' dados o estado atual s e ação a
- $R : S \times A \to \mathbb{R}$ a função de retorno
- $\gamma \in [0, 1]$ é um fator que determina a importância de retornos futuros comparados com retornos imediatos.

Teremos também um conjunto F de *estados terminais*, onde a probabilidade de transição para qualquer outro estado é 0.

O agente define uma *política* $\pi : A \times S \to [0, 1]$ que representa a distribuição de probabilidade da escolha de uma ação em um dado estado $\pi(a|s)$.

Assim temos um mecanismo para gerar amostras, partindo de um estado S_0, selecionamos uma ação $A_0 \sim \pi(a|S_0)$, deixamos o agente observar $R_0 = R(S_0, A_0)$ e selecionamos um estado $S_1 \sim T(s|S_0, A_0)$, a partir do qual o processo se repete, até alcançar um estado S_T que seja terminal. Ao conjunto $S_0, A_0, R_0, S_1, A_1, R_1, S_2, A_2, R_2, ..., S_T$, damos o nome de *episódio*.

O objetivo do agente é, através de suas observações dos episódios, aprender uma política que maximiza o *retorno descontado* G_0 onde: $G_t := R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ... = R_t + \gamma G_{t+1}$.

Será útil para nossa discussão definir a função *valor* $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$ de um estado s, para a política π em um tempo t qualquer, e a função *ação-valor* $q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$.

2.2 Ambiente

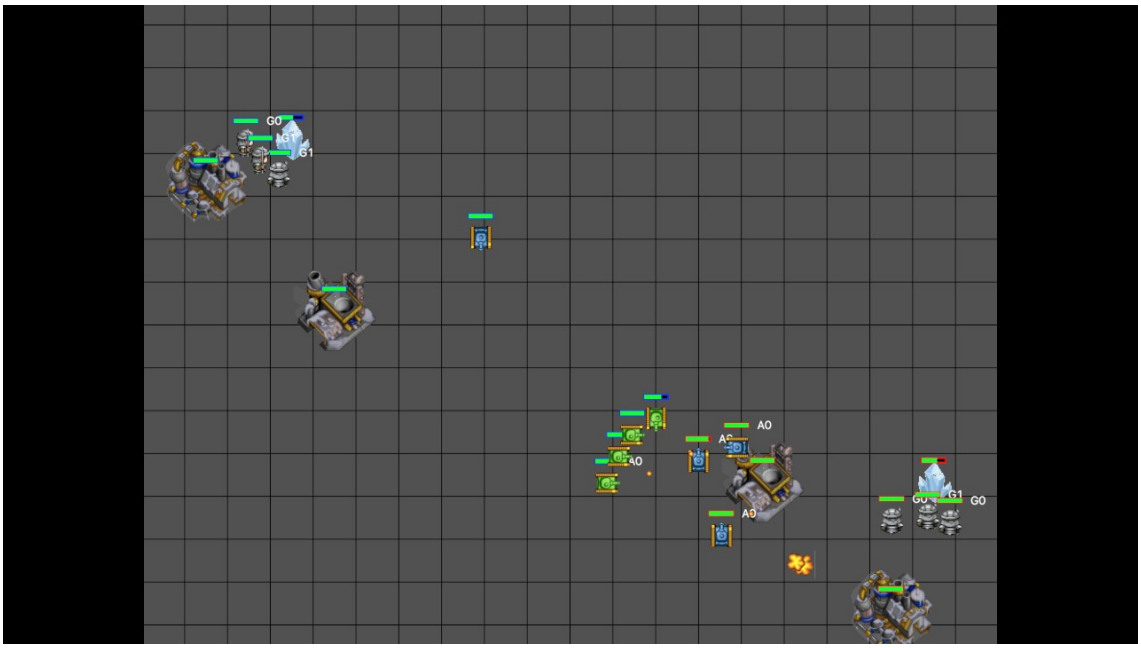

Figura 2.1: *Cenário completo do jogo*

O ambiente utilizado no trabalho foi o ELF [Tian *et al.* (2017)]. Criado para a pesquisa em IA, sua escolha propõe múltiplos benefícios: proporciona todos os desafios característicos de jogos RTS, dispensa das complexidades irrelevantes para pesquisa que estão presentes em jogos destinados para o mercado, é integrado com uma framework de Aprendizado de Reforço (baseada em PyTorch [Paszke *et al.* (2017)]), faz um uso altamente eficiente dos recursos computacionais e, por último, proporciona alto grau de controle para o usuário.

O jogo consiste de dois jogadores disputando por recursos, construindo unidades e as controlando com o objetivo final de destruir a base do adversário. Mais precisamente, cada jogador começa com sua base em cantos opostos de um mapa quadricular, que contém fontes de recursos. É importante ressaltar que os jogadores só tem visão do que está suficiente próximo de suas unidades (e conhecimento da localização da base inimiga), isso significa que se trata de um problema de *informação parcial*, ilustrado na figura 2.2. Assim, a formulação do problema como um MDP é uma aproximação.

A observação do estado é uma discretização do mapa em uma grid 20 x 20, e a informação é distribuída em múltiplos canais (nome herdado da área de processamento de imagens), o que significa que o estado é um cubo H x 20 x 20, onde cada matriz i x 20 x 20 contém um tipo de informação do mapa. Essas informações são: primeiro a posição de unidades de cada tipo, depois o HP (quantidade de pontos de vida de cada unidade, quando o HP atinge 0, a unidade é destruída) e por fim um canal que representa as fontes de recursos, respeitando a visão parcial.

A recompensa dada ao agente é muito simples: +1 no caso de vitória, -1 no caso de derrota. Essa simplicidade é um grande obstáculo, levando ao problema de *recompensas*

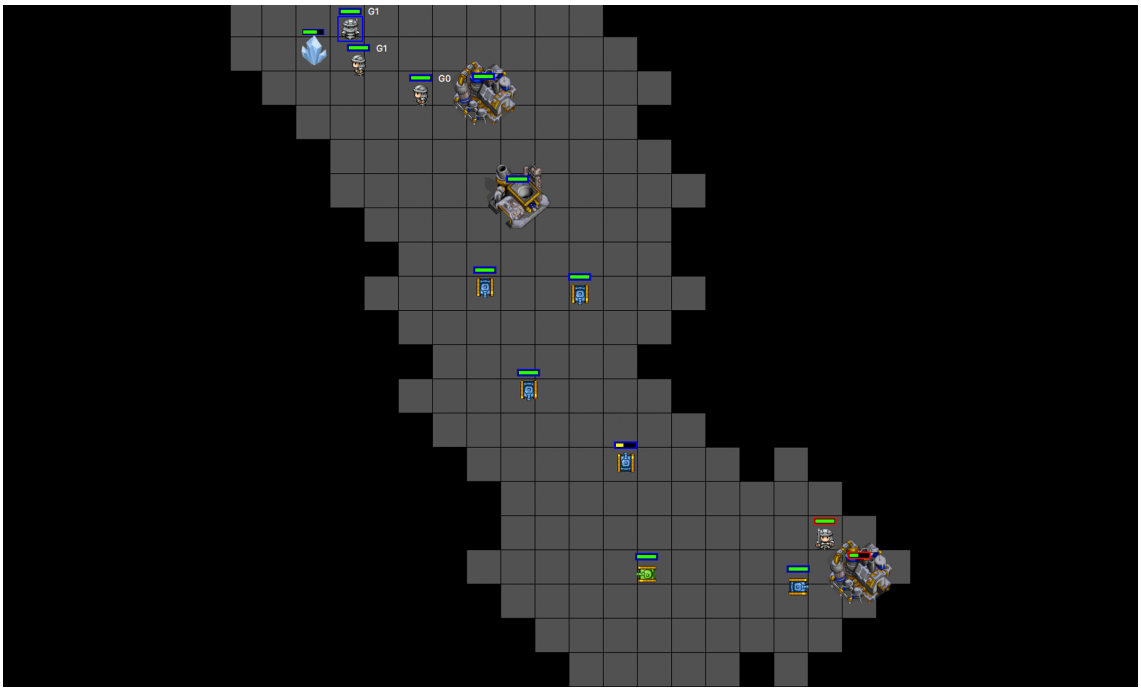

Figura 2.2: *Cenário parcialmente oculto*

esparsas. É comum na área manualmente criar uma função heurística de recompensa para ajudar o treinamento dos agente. Um exemplo seria contar o valor das peças no tabuleiro para um agente aprendendo a jogar xadrez. Não faremos isso neste trabalho.

A noção de tempo no jogo precisa também ser discretizada. Nos nossos experimentos, permitimos uma ação de cada lado para cada 50 frames do jogo.

As ações disponíveis para os agentes são as mesmas em todos os estados, e estão descritas na tabela 2.1.

Comando	Descrição
INATIVO	Não faz nada
CONSTRÓI-TRABALHADOR	Se a base está inativa, constrói um trabalhador.
CONSTRÓI-QUARTEL	Move um trabalhador (coletando recursos ou inativo) para um lugar vazio e constrói um quartel.
CONSTRÓI-GLADIADOR	Se existe um quartel inativo, constrói um gladiador.
CONSTRÓI-TANQUE	Se existe um quartel inativo, constrói um tanque.
BATER-E-CORRER	Se existem tanques, move eles em direção a base inimiga e ataca, fuja se forem contra atacados.
ATACAR	Gladiadores e tanques atacam a base inimiga.
ATACAR EM ALCANCE	Gladiadores e tanques atacam inimigos a vista.
TODOS A DEFESA	Gladiadores e tanques atacar tropas inimigas perto da base ou da fonte de recursos.

Tabela 2.1: *Ações*

Capítulo 3

Solucionando o problema

Atacaremos o problema com a seguinte abordagem, chamada de *gradiente de política*: vamos definir uma *parametrização* da política, ou seja, uma família de funções da forma $\pi_\theta(a|s) = \mathbb{P}(A_t = a|S_t = s, \theta_t = \theta)$ e uma função objetivo $\mathcal{J}(\theta)$ de tal forma que nosso problema se reduz a encontrar um vetor $\theta \in \mathbb{R}^d$ que maximiza a função $\mathcal{J}$. Para isso, utilizaremos o gradiente da função $\mathcal{J}$, por isso temos que escolher π e $\mathcal{J}$ diferenciáveis em θ.

Vale notar que essa abordagem é até aqui idêntica a abordagem padrão utilizada em problemas de Aprendizado Supervisionado Profundo. A diferença estará na definição da função $\mathcal{J}$ e, por consequência, como iremos calcular seu gradiente. Podemos fazer a distinção entre 2 elementos do algoritmo: o *modelo* (que corresponde a escolha da parametrização da política) e o *método de otimização* (que é o algoritmo que encontrará um vetor de parâmetros satisfatório).

3.1 Modelo

O nosso modelo será um Rede Neural Convolucional [O'Shea e Nash (2015)]. Com um poder de representatividade muito grande e acompanhadas de poderosos métodos de otimização, modelos baseados em Redes Neurais têm batido recordes em um grande número de tarefas em Aprendizado de Máquina [LeCun *et al.* (2015)].

3.1.1 Redes Neurais

Uma Rede Neural é essencialmente uma composição de várias funções simples, chamadas de *neurônios*. Um neurônio é em geral uma função da forma $\sigma(T(x;\theta))$, onde T é uma função linear e σ é uma função não linear (diferenciável).

Redes Neurais possuem um poderoso algoritmo chamado de *retroprogagação* (em inglês *backpropagation*) [LeCun *et al.* (2015)] para calcular a derivada do modelo inteiro de uma forma eficiente.

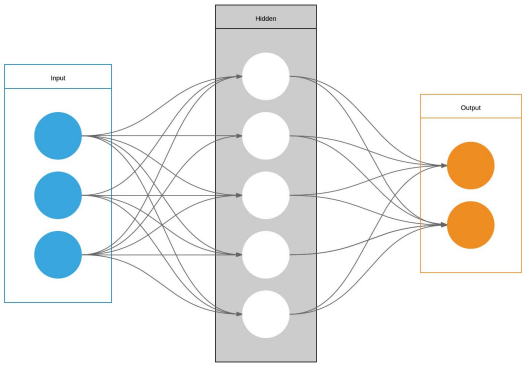

Figura 3.1: *Uma Rede Neural simples*

As Redes Neurais Convolucionais são Redes Neurais que explicitamente colocam no modelo hipóteses sobre a distribuição espacial dos dados, principalmente através de assumir que a informação está codificada em uma matriz quadrada (ou várias delas) de forma invariante a translação. Esse tipo de rede têm sido revolucionária na área de visão computacional [LeCun *et al.* (2015)], e é próprio a aplicação delas aqui, já que o estado do jogo é comunicado através de um mapa quadricular que respeita essas hipóteses. Falaremos mais sobre elas no capítulo sobre nosso modelo.

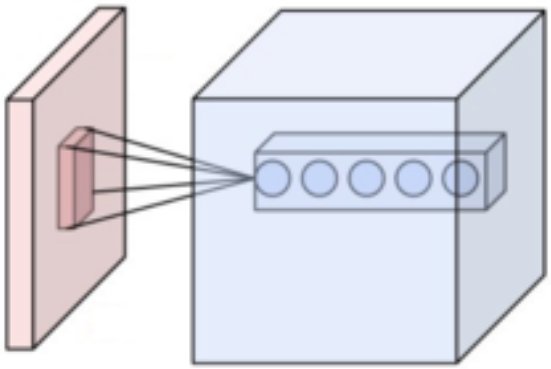

Figura 3.2: *Uma Rede Neural Convolucional*

3.1.2 Hiperparâmetros

É comum em Aprendizado de Máquina em geral e Aprendizagem Profunda em específico a presença de várias variáveis que não são otimizadas pelo algoritmo de otimização utilizado (em Aprendizagem Profunda elas surgem quando a função objetivo não é diferenciável em função da variável). Estas variáveis são chamadas de *hiperparâmetros*. De maneira geral, escolhemos hiperparâmetros de acordo com o melhor consenso da literatura atual e/ou uma busca exaustiva em várias opções.

3.2 Método de Otimização

O nosso algoritmo de otimização será uma variação de *subida de gradiente* LeCun *et al.* (2015). Esse algoritmo, que é a base dos algoritmos de otimização utilizados em Aprendizagem Profunda, consiste em fazer pequenos ajustes no parâmetro θ na direção do gradiente $\nabla\mathcal{J}(\theta)$, resumidamente, $\theta_{t+1} = \theta_t + \gamma\nabla\mathcal{J}(\theta)$. A ideia do algoritmo vem do fato que, para um γ suficientemente pequeno, teremos que $\mathcal{J}(\theta_{t+1}) > \mathcal{J}(\theta_t)$ se o gradiente for diferente de 0.

3.2.1 Função Objetivo

Nossa função objetivo será escolhida como $\mathcal{J}(\theta) \doteq v_{\pi_\theta}(S_0)$, o valor do estado inicial na política definida por θ, ou seja, o valor esperado do retorno descontado durante todo o jogo. Assim, diretamente adaptamos o objetivo dos nossos agentes como uma função de θ.

Essa função é diferenciável e além disso temos uma forma conveniente de expressar seu gradiente através do teorema do gradiente de política [Sutton e Barto (2011)]:

$$\nabla\mathcal{J}(\theta) \propto \mathbb{E}_\pi\Big[q_\pi(S_t, A_t)\frac{\nabla_\theta\pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)}\Big] = \mathbb{E}_\pi\Big[G_t\frac{\nabla_\theta\pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)}\Big] \tag{3.2}$$

Onde os termos são aqueles definidos no capítulo 2.

Ou seja, podemos simular o ambiente, calcular a expressão entre os colchetes, e com ela fazer múltiplas atualizações do nosso parâmetro, obtendo uma aproximação estatística da subida de gradiente. Entraremos em detalhes da otimização em um capítulo posterior.

Capítulo 4

Redes Neurais Convolucionais

O nosso modelo será uma Rede Neural Convolucional com duas cabeças: uma que nos dá a distribuição $\pi(\cdot|S)$ e a outra nos dá a estimativa do valor $\hat{v}$.

4.1 Os componentes da rede

4.1.1 Redes Neurais Convolucionais

A entrada deve ser codificada como um cubo (H, W, L). A camada é codificada como um conjunto de F *filtros*, cada um com dimensão (H', W', L). A saída de cada filtro será uma matriz quadrada, e a saída da camada é a composição das F matrizes como um cubo. A saída do filtro é calculada passando o filtro na imagem original, produzindo o produto escalar do filtro e da seção da entrada.

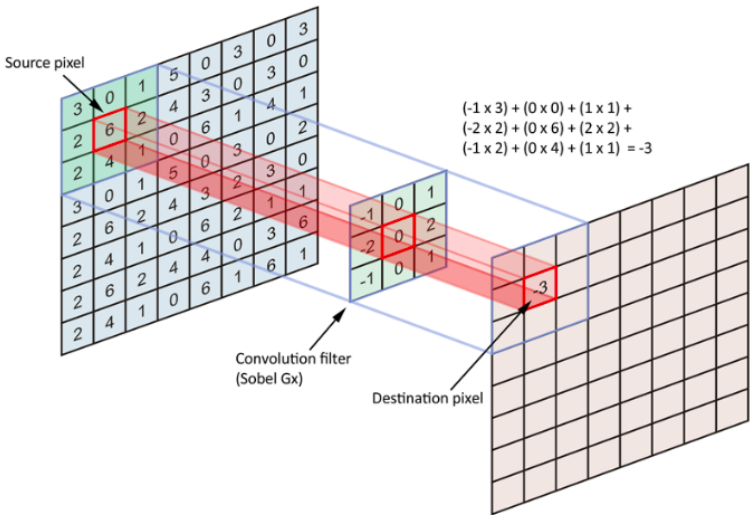

Figura 4.1: *A operação de convolução*

4.1.2 ReLU

Há algumas opções de função de ativação para redes neurais como a função sigmóide, tangente hiperbólica e a unidade linear retificada (ReLU, de *Rectified Linear Unit*). A função ReLU é definida por ReLU(x) = $max(0, x)$. O ReLU, apesar de muito simples, é suficiente para as garantias de representabilidade de Redes Neurais que o utilizam e apresenta uma importante vantagem comparado com as outras funções mencionadas: ele não admite derivadas pequenas, que atrasam o processo de aprendizado. Por outro lado, o ReLU possui o problema de *neurônios mortos*, pois quando seu output é 0, sua derivada também é 0 e não ocorre aprendizado.

Para combater esse problema, foi criada uma variação do ReLU chamada de Leaky ReLU, dado pela função $max(x, \alpha x)$, onde $\alpha < 1$ é um hiperparâmetro.

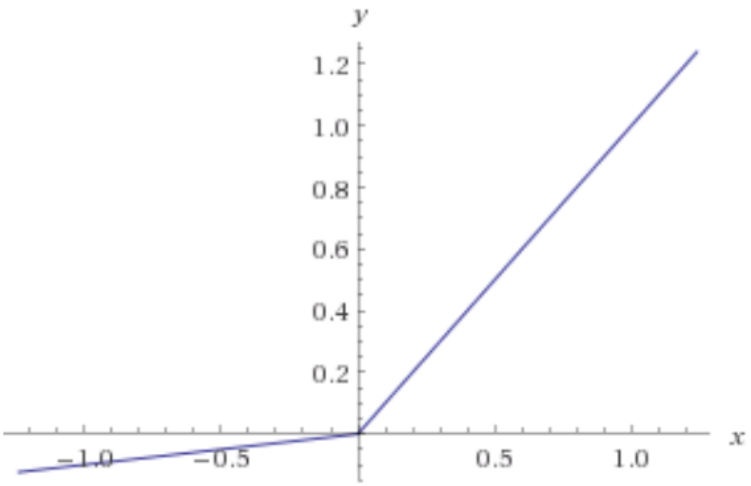

Figura 4.2: *O leaky ReLU*

4.1.3 Pooling

É comum encontrar em arquiteturas de redes convolucionais camadas de *pooling*, que consistem de filtros que diminuem a dimensionalidade dos dados. O mais comum desses é o *max-pool*, que propaga para frente o maior valor dentro de uma área $X \times X$, e ignora o resto.

Capítulo 5

Algoritmos Actor-Critic

Os algoritmos de otimização que veremos em seguida funcionam unicamente na hipótese que temos $\mathcal{J}$ e π diferenciáveis, logo poderemos aplicá-los no nosso modelo baseado em Redes Neurais.

5.1 REINFORCE

Comentamos no capítulo anterior que

$$\nabla \mathcal{J}(\theta) \propto \mathbb{E}_{\pi}\Big[G_t \frac{\nabla_{\theta}\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}\Big] \tag{5.0}$$

Onde os termos são aqueles definidos no capítulo 2.

E que isso sugere um algoritmo para achar um vetor θ, chamado REINFORCE [Sutton e Barto (2011)], que pode ser descrito como:

$$\theta = \theta + \alpha\gamma^t G_t \frac{\nabla_{\theta}\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \tag{5.0}$$

Onde G_t é o retorno como sempre. Os parâmetros γ e α são hiperparâmetros. O primeiro é o fator de desconto da definição do MDP. O segundo é chamado de *taxa de aprendizado* (em inglês *learning rate*). Um α muito pequeno deixa o aprendizado lento, porém um α grandex perde a garantia de melhora da função objetivo. É comum diminuir o α com o número de iterações.

Não existe consenso no número correto de iterações. Depois de muitas iterações, o gradiente tende a ficar cada vez menor, e a função objetivo permanece estável, o que pode significar que o algoritmo encontrou um ótimo local. Quando se faz múltiplos experimentos, é comum fixar um número de iterações entre eles.

Esse algoritmo é próximo da teoria porém é relativamente lento na prática, principalmente pela variância no processo de treinamento, dado que as atualizações dependem totalmente dos episódios observados. Em seguida, faremos uma modificação do algoritmo que ataca esse problema.

5.2 Actor-Critic

Uma modificação relativamente natural do algoritmo REINFORCE é tentar aproximar a função ação-valor na fórmula, como em [Sutton e Barto (2011)]:

$$\nabla \mathcal{J}(\theta) \propto \mathbb{E}_\pi \Big[q_\pi(S_t, A_t) \frac{\nabla_\theta \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \Big] \tag{5.0}$$

Daí surge a classe de algoritmos *Actor-Critic* [Sutton e Barto (2011)]. A versão que exploraremos é a base do algoritmo A3C, que veremos em seguida, e utiliza como aproximação de $q_\pi(S_t, A_t)$ a expressão $R_t + \gamma \hat{v}_{\pi,\mu}(S_{t+1}) - \hat{v}_{\pi,\mu}(S_t)$ onde $\hat{v}_{\pi,\mu}$ é uma aproximação da função valor (aproximação que vêm do fato que $q_\pi(S_t, A_t) = \mathbb{E}_\pi[R_t + \gamma v_\pi(S_{t+1})|S_t, A_t]$ [Sutton e Barto (2011)] e não mudamos a esperança subtraindo da aproximação uma função que depende somente do estado [Sutton e Barto (2011)]), parametrizada por μ. Nos parágrafos em seguida, escreveremos $\hat{v}_{\pi,\mu}$ como $\hat{v}_\mu$.

Vale notar que o valor de um estado terminal deve ser 0, pois o jogo acabando, o valor esperado das recompensas futuras é 0.

Algorithm 1 Actor-Critic

procedure ACTOR-CRITIC($\pi, \hat{v}, \alpha_\theta, \alpha_\mu$)
  Inicialize θ e μ
  while *true* **do** ▷ Não há previsão teórica para o ponto de parada
    $S \leftarrow S_0$
    $I \leftarrow 1$
    while S não é terminal **do**
      $A \sim \pi_\theta(\cdot|S)$
      Tome a ação A, observando R e S'
      $\delta \leftarrow R + \gamma \hat{v}_\mu(S') - \hat{v}_\mu(S)$
      $\mu \leftarrow \mu + \alpha_\mu I \delta \nabla_\mu \hat{v}_\mu(S)$
      $\theta \leftarrow \theta + \alpha_\theta I \delta \nabla_\theta \ln \pi_\theta(A|S)$ ▷ Aqui usando a identidade que $\nabla \ln f = \frac{\nabla f}{f}$
      $I \leftarrow \gamma I$
      $S \leftarrow S'$
    end while
  end while
end procedure

A otimização da função é chamada de TD(0) de subgradiente [Sutton e Barto (2011)]. A ideia é minimizar o erro quadrático da estimação do valor $(v_\pi(S_i) - \hat{v}_{\mu'}(S_i))^2$, utilizando como estimação de $v_\pi(S_i)$ a expressão $R + \gamma \hat{v}_\mu(S')$ e realizando descida de gradiente. Porém ao invés de gradiente é utilizado o subgradiente, pois é ignorado o efeito do parâmetro μ no alvo da otimização.

5.3 A3C

O algoritmo A3C, de *Asynchronous Advantage Actor-Critic* [Mnih *et al.* (2016)], é uma modificação do Actor-Critic que usa fortemente paralelismo para diversificar a experiência do agente a cada atualização dos parâmetros (o que deixa o treinamento mais estável) e usufruir totalmente de processadores multi-core, por esses motivos, o treinamento com o algoritmo A3C tende a ser significativamente mais rápido do que com Actor-Critic puro.

No código abaixo, t_{max} é um hiperparâmetro que dita quão frequente é a atualização dos parâmetros globais e T_{max} é outro hiperparâmetro que indica o número total de atualizações dos parâmetros.

Algorithm 2 A3C - pseudocódigo para cada thread

procedure A3C($\pi, \hat{v}$)
  Assuma que existem parâmetros globais θ e μ e um contador global $T = 0$
  Inicialize um contador da thread $t \leftarrow 1$
  repeat
    Inicialize gradientes $d\theta \leftarrow 0$ e $d\mu \leftarrow 0$
    Sincronize parâmetros da thread $\theta' \leftarrow \theta$ e $\mu' \leftarrow \mu$
    $t_{start} \leftarrow t$
    Recolha estado S_t
    repeat
      $A_t \sim \pi_{\theta'}(\cdot|S_t)$
      Tome a ação A_t, observando R_t e S_{t+1}
      $t \leftarrow t + 1$
      $T \leftarrow T + 1$
    until S_t é terminal ou $t - t_{start} = t_{max}$
    $R \leftarrow \hat{v}_{\mu'}(S_t)$
    for $i \in \{t-1, ..., t_{start}\}$ **do**
      $R \leftarrow R_i + \gamma R$
      $d\mu \leftarrow d\mu + \nabla_{\mu'}(R - \hat{v}_{\mu'}(S_i))^2$
      $d\theta \leftarrow d\theta + (R - \hat{v}_{\mu'}(S_i))\nabla_{\theta'} \ln \pi_{\theta'}(A_i|S_i)$ ▷ Novamente $\nabla \ln f = \frac{\nabla f}{f}$
    end for
    Atualize θ e μ com os gradientes $d\theta$ e $d\mu$
  until $T > T_{max}$
end procedure

5.4 Subida de gradiente

Nas seções acima, vimos como é calculado o gradiente da função objetivo. Agora, falaremos mais sobre a atualização dos parâmetros com esse gradiente. O *otimizador* utilizado será o *Adam* [Kingma e Ba (2014)]. O Adam é uma das várias versões de otimizadores baseados em subida de gradiente [Ruder (2016)]. Ele consiste da união de duas modificações ao algoritmo base: momento e taxa de aprendizado adaptável.

Momento consiste em atualizar os parâmetros utilizando uma combinação dos gradientes anteriores, ao invés de somente usar o gradiente mais recente. Essa alteração acelera o treinamento quando há várias atualizações seguidas na mesma direção, rapidamente aproximando os parâmetros de um ótimo local, e demorando um pouco mais para convergir uma vez lá. Também há uma vantagem quando a função objetivo "se move"como é o caso de estimativas estatísticas da função verdadeira.

A taxa de aprendizado adaptável mistura também duas ideias: ela varia de acordo com parâmetro e diminui quando o gradiente aumenta. A ideia aqui é lidar com gradiente esparsos e gradientes "explosivos", garantindo uma atualização mais consistente de todos os parâmetros.

Capítulo 6

Resultados

A arquitetura da nossa rede consiste de duas camadas convolucionais (utilizando o ReLU como função de ativação) e uma camada de max-pool. Temos então duas operações lineares, uma determinando a política e a outra determinando o valor do estado.

Utilizamos lotes de tamanho 128 para a subida de gradiente, utilizamos $T_{max} = 20$ no A3C, $\gamma = 0.99$ para o retorno descontado do MDP e 10^{-3} como taxa de aprendizado.

Nossos experimentos foram feitos conforme [Tian *et al.* (2017)]. Nosso modelo é uma simplificação do modelo do artigo (principalmente para economizar recursos computacionais): as redes utilizadas no artigo podem ser formadas concatenando múltiplas das nossas, e possivelmente adicionando camadas de normalização de lote depois de cada camada convolucional. Nossos hiperparâmetros são os mesmos utilizados no artigo.

Os experimentos foram todos calculados em um computador com 16 CPUs, 42 GB de memória e 1 GPU. Cada experimento teve entre 2 e 5 horas de execução.

6.1 Treinamento

Reportamos os resultados do treinamento contra 2 estratégias fixas, que chamamos de AI_SIMPLE e AI_HIT_AND_RUN, conforme [Tian *et al.* (2017)]. Chamamos de *acurácia* a fração $\frac{\text{Vitórias}}{\text{Jogos Totais}}$. Nos gráficos, vemos a melhor acurácia atingida até determinado episódio de treinamento, calculada em 1000 jogos.

AI_SIMPLE constrói 3 trabalhadores, junta recursos, constrói um quartel, e então constrói gladiadores. Quando atinge 5 gladiadores, ataca a base inimiga.

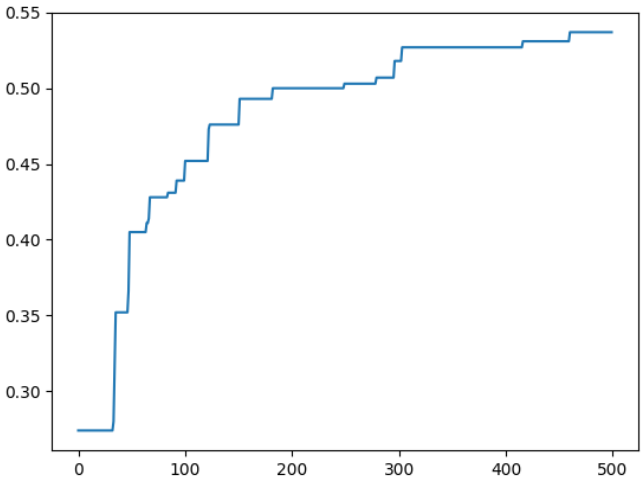

Figura 6.1: *Acurácia durante o treinamento contra AI_SIMPLE*

AI_HIT_AND_RUN constrói 3 trabalhadores, junta recursos, constrói um quartel, e então constrói tanques. Quando atinge 2 tanques, ele ataca o inimigo, caso o inimigo contra-ataque, ele recua, se mantendo fora de alcance.

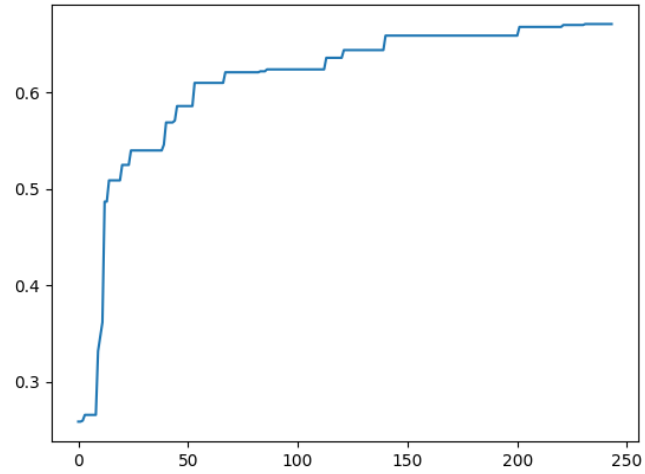

Figura 6.2: *Acurácia durante o treinamento contra AI_HIT_AND_RUN*

6.1.1 Informação perfeita

Experimentamos também treinar o mesmo modelo no caso de informação perfeita, isto é, removendo a *fog-of-war* (visto na Figura 6.3) e dando ao agente toda a informação do mapa.

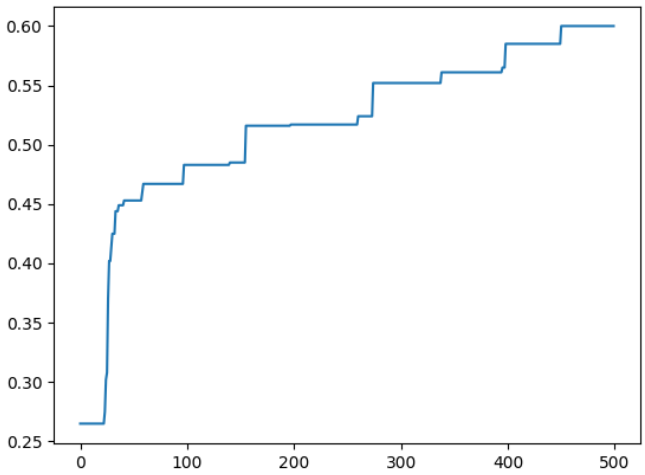

Figura 6.3: *Acurácia durante o treinamento contra AI_SIMPLE com informação perfeita*

6.2 Resultados finais

Abordagem	Acurácia
Agente aleatório vs AI_SIMPLE	0.242
Agente aletório vs AI_HIT_AND_RUN	0.259
vs AI_SIMPLE	0.537
vs AI_HIT_AND_RUN	0.671
Tian et al. vs AI_SIMPLE	0.684
Tian et al. vs AI_HIT_AND_RUN	0.636

Tabela 6.1: *Comparação dos principais resultados*

Os resultados obtidos foram consistentes com os experimentos equivalentes no artigo.

Contra AI_HIT_AND_RUN, os resultados encontrados foram melhores, porém dentro da variação estatística reportada no artigo. É possível que nosso modelo simplificado é capaz de rapidamente convergir para uma solução simples e eficiente contra essa estratégia, superando o resultado do artigo.

Contra AI_SIMPLE, os resultados do artigo foram decisivamente melhores. Podemos atribuir essa diferença ao modelo mais complexo, e a um processo de otimização diferente: utilizando *curriculum learning*, onde o algoritmo primeiro observa o AI_SIMPLE jogando

contra si mesmo (aprendendo *off-policy*) e depois começa a ser treinado a partir da observação de sua própria política interagindo com o ambiente (aprendendo *on-policy*). No artigo, o agente também foi treinado a partir de estados aletórios, aumentado a diversidade do treino.

Abordagem	Acurácia
vs AI_SIMPLE	0.537
vs AI_SIMPLE, com informação perfeita	0.604

Tabela 6.2: *Comparação de informação perfeita/imperfeita*

Na tabela 6.2, podemos ver que a aproximação desse ambiente com informação imperfeita como um MDP tem um custo, porém este é relativamente pequeno, e o agente é capaz de aprender o suficiente para passar a linha de 50% de vitória.

Capítulo 7

Conclusão

Terminamos aqui essa investigação inicial da área de Aprendizado de Reforço Profundo. Passamos pelos principais conceitos e algoritmos essenciais da área, construindo uma base para se aprofundar no assunto. Muito do trabalho atual na área é uma continuação natural do que é visto aqui, como a construção de redes neurais mais complexas para a representação do agente e algumas mudanças no processo de treinamento, buscando aumentar a qualidade da experiência inicial do agente, que sob os algoritmos descritos nesse trabalho, consistem somente em uma busca aleatória a vitória e portanto recompensas que possibilitam o processo de aprendizado.

Apesar disso, os algoritmos aqui estudados já são suficientemente poderosos para atacar vários problemas, podendo lidar com um conjunto de estados grande e complexo, distinguindo-se dos algoritmos tabulares clássicos de Aprendizado de Reforço. Poder que vêm também com desvantagens, pois os algoritmos empregados requerem uma quantidade grande de tempo de treinamento.

A área de Aprendizado de Reforço tem um grande potencial, dada a generalidade de suas hipóteses básicas. Os experimentos realizados nesse trabalho foram possibilitados por contribuições recentes: o ambiente de treinamento ELF (2017), a *framework* PyTorch (2016) e até mesmo o Google Compute Engine (2012), de onde foi alugado o poder computacional para a realização dos experimentos. Esperamos que esse trabalho faça também sua pequena contribuição para o avanço da área.

Referências Bibliográficas

Arulkumaran *et al.***(2017)** Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage e Anil Anthony Bharath. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866. URL http://arxiv.org/abs/1708.05866. Citado na pág. 1, 3

Kaelbling *et al.***(1996)** Leslie Pack Kaelbling, Michael L Littman e Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285. Citado na pág. 1

Kingma e Ba(2014) Diederik P. Kingma e Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980. URL http://arxiv.org/abs/1412.6980. Citado na pág. 15

LeCun *et al.***(2015)** Yann LeCun, Y Bengio e Geoffrey Hinton. Deep learning. *Nature*, 521: 436–44. doi: 10.1038/nature14539. Citado na pág. 1, 7, 8, 9

Mnih *et al.***(2016)** Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver e Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783. URL http://arxiv.org/abs/1602.01783. Citado na pág. 15

O'Shea e Nash(2015) Keiron O'Shea e Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458. URL http://arxiv.org/abs/1511.08458. Citado na pág. 7

Paszke *et al.***(2017)** Adam Paszke, Sam Gross, Soumith Chintala e Gregory Chanan. Pytorch, 2017. Citado na pág. 4

Ruder(2016) Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747. URL http://arxiv.org/abs/1609.04747. Citado na pág. 15

Sutton e Barto(2011) Richard S Sutton e Andrew G Barto. Reinforcement learning: An introduction. Citado na pág. 1, 3, 9, 13, 14

Tian *et al.***(2017)** Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu e C Lawrence Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. Em *Advances in Neural Information Processing Systems*, páginas 2659–2669. Citado na pág. 2, 4, 17