Universidade Federal
do Rio de Janeiro
Escola Politécnica

APRENDIZADO POR REFORÇO COM APROXIMADORES NÃO LINEARES APLICADOS A JOGOS ELETRÔNICOS

Lyang Higa Cano

Projeto de Graduação apresentado ao Curso de Computação e Informação da Escola Politécnica da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Computação e Informação.

Orientador: Alexandre Gonçalves Evsukoff

Rio de Janeiro
Agosto de 2019

APRENDIZADO POR REFORÇO COM APROXIMADORES NÃO LINEARES APLICADOS A JOGOS ELETRÔNICOS

Lyang Higa Cano

PROJETO SUBMETIDO AO CORPO DOCENTE DO CURSO DE COMPUTAÇÃO E INFORMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE COMPUTAÇÃO E INFORMAÇÃO.

Examinadores:

Prof. Alexandre Gonçalves Evsukoff, Dr.

Prof. Daniel Ratton Figueiredo, Dr.

Prof. Fernando Gil Vianna Resende Junior, Dr.

RIO DE JANEIRO, RJ – BRASIL
AGOSTO DE 2019

Cano, Lyang Higa

Aprendizado Por Reforço com Aproximadores Não Lineares Aplicados a Jogos Eletrônicos/Lyang Higa Cano. – Rio de Janeiro: UFRJ/POLI – COPPE, 2019.

XII, 58 p.: il.; 29, 7cm.

Orientador: Alexandre Gonçalves Evsukoff

Projeto (graduação) – UFRJ/ Escola Politécnica/ Curso de Computação e Informação, 2019.

Referências Bibliográficas: p. 56 – 58.

1. Aprendizado Por Reforço. 2. IA. 3. Q-Learning. 4. DQN. 5. Redes Neurais - RNA. 6. DQN. I. Evsukoff, Alexandre Gonçalves. II. Universidade Federal do Rio de Janeiro, Escola Politécnica/ Curso de Computação e Informação. III. Título.

"Ninguém ignora tudo. Ninguém sabe tudo. Todos nós sabemos alguma coisa. Todos nós ignoramos alguma coisa. Por isso aprendemos sempre."

— Paulo Freire

Agradecimentos

À minha família, especialmente aos meus pais, que sempre me apoiaram.

Aos meus amigos, os de longa data e os que a UFRJ e o Rio me apresentaram.

Agradeço ao meu orientador, Alexandre Evsukoff, a todos professores, funcionários e colegas da Universidade Federal do Rio de Janeiro.

A todo o povo brasileiro, que possibilitou toda a minha formação em instituições de excelência sempre de maneira gratuita.

Resumo do Projeto de Graduação apresentado à Escola Politécnica/COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Computação e Informação.

APRENDIZADO POR REFORÇO COM APROXIMADORES NÃO LINEARES APLICADOS A JOGOS ELETRÔNICOS

Lyang Higa Cano

Agosto/2019

Orientador: Alexandre Gonçalves Evsukoff

Curso: Engenharia de Computação e Informação

A Inteligência Artificial tem se tornado cada vez mais importante no mundo atual e o Aprendizado por Reforço tem ocupado uma área de destaque. No entanto, ainda há diversos problemas em aberto, como o problema de Convergência de Política Ótima, que ocorre ao combinar três elementos essenciais, o *Bootstrapping*, a metodologia *Off-Policy* e os *Aproximadores Não Lineares*. Até que o Artigo *Human Level Control Through Deep Reinforcement Learning* [1] do grupo de pesquisa *Deep Mind* propõem uma nova abordagem que consegue contornar tal problema.

Este trabalho tem como objetivo apresentar cada um destes três elementos, apresentar o Problema de Convergência de Política Ótima, verificar a abordagem proposta pelo Artigo [1], realizar um novo teste e avaliar se tal proposta é realmente eficaz. A contribuição deste trabalho consiste primeiramente na revisão bibliográfica deste tema, que é extenso e complexo, numa análise profunda do Artigo [1], especialmento do Algoritmo *Deep Q-Network*, e por fim em uma nova avaliação através de experimentos em um ambiente diferente dos utilizados no Artigo. **Palavras-Chave:**

Aprendizado Por Reforço, IA, Q-Learning, DQN, Redes Neurais - RNA, DQN.

Abstract of the Undergraduate Project presented to Poli/COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Computer and Information Engineer.

REINFORCEMENT LEARNING WITH NONLINEAR APPROXIMATION IN ELECTRONIC GAMES

Lyang Higa Cano

August/2019

Advisor: Alexandre Gonçalves Evsukoff

Course: Computer and Information Engineering

Artificial Intelligence has become increasingly important in today's world and Reinforcement Learning has occupied a prominent area. However, there are still several open problems, such as the Optimal Policy Convergence problem, which occurs by combining three essential elements, Bootstrapping, Off-Policy Methodology, and Nonlinear Approximation.

Until the Deep Mind Research Group's Human Level Control Through Deep Reinforcement Learning article proposes a new approach that can circumvent such a problem. This work aims to present each of these three elements, present the Optimal Policy Convergence Problem, verify the approach proposed by the Article, retest and evaluate whether such a proposal is really effective. The contribution of this work consists firstly of the bibliographical revision of this extensive and complex subject, a deep analysis of the Article, especially of the Deep Q-Network Algorithm, and finally a new evaluation through experiments in a different environment from those used in the Article.

Keywords: Reinforcement Learning, AI, Q-Learning, Artificial Neural Network - ANN.

Sumário

Lista de Figuras **x**

Lista de Tabelas **xii**

1 Introdução **1**

1.1 Motivação 1

1.2 Objetivos 2

1.3 Organização 2

2 Aprendizado por Reforço **4**

2.1 Introdução 4

2.2 Markov Decision Processes (MDP) 5

2.3 Recompensas, Políticas e Valores Ótimos 6

2.4 Programação Dinâmica 9

2.4.1 Avaliação de Política (Policy Evaluation) 10

2.4.2 Aprimoramento de Política (Policy Improvement) 10

2.4.3 Iteração de Política (Policy Iteration) 11

2.4.4 Iteração de Valor (Value Iteration) 12

2.4.5 Iteração de Política Vs Iteração de Valor 13

2.5 Model-Free Prediction 14

2.5.1 Monte Carlo Prediction (MC Prediction) 14

2.5.2 Predição por diferenciação temporal (Temporal-Difference Prediction) 16

2.5.3 Monte Carlo Vs Temporal Difference 17

2.6 Model Free Control 17

2.6.1 Monte Carlo Control 18

2.6.2 On-Policy MC Control sem 'Exploring Starts' 19

2.6.3 Sarsa(0): On-Policy TD(0) Control 21

2.7 Metodologia Off-Policy 22

2.7.1 Q-Learning 22

3 Aprendizado por Reforço e Aproximadores de Funções **25**
3.1 Predição com Aproximadores 26
3.1.1 Mínimos Quadrados 27
3.1.2 Gradiente Descendente Estocástico (Stochastic Gradient Descent SGD) 28
3.2 Aproximadores Lineares 29
3.3 Aproximadores não Lineares - Redes Neurais Artificiais (RNAs) . . . 31
3.4 Controle On-Policy com Aproximadores 33

4 Metodologia **35**
4.1 Experience Replay 38
4.2 Atualizações Dessincronizadas dos Pesos da RNA 38
4.3 Algoritmo Deep Q-Network 39
4.4 Flappy Bird 40
4.5 Implementação 42
4.5.1 Módulos 42
4.5.2 Agentes 46
4.6 Treinamento 48
4.6.1 Exploration - Exploitation e o Efeito da Aleatoriedade 49

5 Resultados **51**

6 Conclusão **54**

Referências Bibliográficas **56**

Lista de Figuras

2.1 Interação agente-ambiente em uma MDP [2]. 5
2.2 Ambiente Grid World [3]. 9
2.3 Solução Grid World com Q-values e função de valores. Política ótima é representada pelas setas [3]. 9
2.4 Algoritmo de Avaliação de Política [2]. 10
2.5 Combinação de avaliação e aprimoramento de política retornando valores ótimos [2]. 11
2.6 Algoritmo Policy Iteration [2]. 12
2.7 Algoritmo Value Iteration [2]. 13
2.8 Comparativo tomada de ações entre Iteração de Valor e Iteração de Política [3]. 13
2.9 Algoritmo Monte Carlo Prediction [2]. 15
2.10 Algoritmo TD(0) Prediction [2]. 16
2.11 Comparativo entre a estimativa de cada método [4]. 17
2.12 Algoritmo Monte Carlo Control com Exploring Starts. [2]. 19
2.13 Algoritmo On-Policy Monte Carlo ε-guloso . [2]. 20
2.14 Avaliação e Aprimoramento de Q-Values para SARSA(0) [4]. Seguimos exatamente a mesma lógica de Iteração de Política, mas dessa vez utilizamos um bootstrapping TD(0) 21
2.15 Algoritmo On-Policy SARSA(0) [2]. 22
2.16 Atualização Q-Learning [4]. 23
2.17 Algoritmo Off-Policy Q-Learning [2]. 23

3.1 SGD Algoritmo com Monte Carlo [2] 29
3.2 SGD Algoritmo com TD(0) [2] 29
3.3 Representação de uma RNA. [2] 31
3.4 Representação de uma unidade e das camadas de uma RNA. [5] . . . 32
3.5 Tabela de Convergência [4]. 33
3.6 SARSA com Aproximadores [2]. 34

4.1 Aproximadores de Q-Function [6]. 36

4.2 Arquitetura da RNA [1]. 37
4.3 Parâmetros de cada camada [6]. 37
4.4 Dinâmica Flappy Bird [7]. 41
4.5 Composição de um agente. Cada figura geométrica representa um módulo que um dado agente pode possuir. 43
4.6 Representação das três arquiteturas implementadas. A primeira 'U' é uma Rede com menos camadas, a segunda é a arquitetura DQN, e a terceira 'O' possui mais camadas que a DQN, ou seja, o tamanho de cada figura está relacionado com o número de camadas da arquitetura. 44
4.7 Módulos do Agente Linear. 46
4.8 Módulos do Agente Reutilizado. 46
4.9 Pontuação x Episódios do Agente DQN na fase de Treino. 49

Lista de Tabelas

4.1 Nomenclatura dos Agentes que são construídos a partir da combinação de módulos. 47
4.2 Comparativo entre os valores dos principais hiperparâmetros entre este trabalho e o Artigo [1]. 48

5.1 Desempenho dos diferentes Agentes no Flappy Bird 51
5.2 Desempenho dos diferentes Agentes, organizados por módulos, no Flappy Bird 52

Capítulo 1

Introdução

1.1 Motivação

A inteligência e o processo de aprendizagem sempre foram tidos como características que diferenciam os humanos dos outros animais ou de qualquer tipo de máquina. Há bastante tempo tentamos definir o que é inteligência e o que seria uma máquina inteligente, facilmente recordamos do *Teste de Turing*, que propõem uma forma de avaliar a inteligência de uma máquina, ou de um *Sistema Especialista*, que procura replicar algum conhecimento humano [8].

Mas foi nos últimos anos que vimos uma grande evolução da área de Inteligência Artificial, especialmente através das técnicas de *Deep Learning*, que representam um grande avanço para os problemas de *Aprendizado Supervisionado* e *Não Supervisionado*, já que muitas vezes facilitam a representação dos problemas e fornecem uma resposta mais precisa.

No entanto, o **Aprendizado por Reforço**, que é caracterizado pela aprendizagem através da tentantiva e erro, onde apenas a experiência do próprio ator gera conhecimento, ou seja, não há necessidade de um professor e ao mesmo tempo não queremos resolver um problema de agrupamento. Portanto, fica claro que o Aprendizado por Reforço não é um problema de Aprendizado Supervisionado ou Não Supervisionado.

É a curiosidade em verificar como essa forma de aprendizado mais independente é modelada e as inovações recentes, que aprimoraram muito as técnicas que já eram conhecidas e, as quais prometem revolucionar a sociedade que conhecemos, que motivam este trabalho.

1.2 Objetivos

A princípio o objetivo deste trabalho era exclusivamente discutir e implementar as ideias propostas pelo Artigo Científico [1], que representa um marco ao introduzir algumas técnicas inovadoras ao Aprendizado por Reforço. Já que neste artigo uma solução é proposta para o problema de Convergência de Política Ótima ao combinar os três elementos: *Bootstrapping*, Metodologia *Off-Policy* e Aproximadores Não Lineares, neste trabalho utilizaremos apenas *Redes Neurais Artificiais (RNAs)*, os quais são fundamentais para resolver problemas grandes encontrados no mundo atual e, por isso, esses três elementos e a formulação deste problema serão apresentados e discutidos ao longo deste trabalho.

No entanto, conforme este trabalho foi sendo realizado, ficou explícito que a revisão bibliográfica representaria uma parte tão, ou mais importante, que as soluções propostas, já que este é um tema bastante vasto e complexo, por isso, um estudo aprofundado é fundamental para o bom entendimento do problema e para a construção da solução. Além disso, foi a partir deste estudo que algumas ideias para verificar tais resultados e oportunidades para trabalhos futuros surgiram.

Portanto, o objetivo final deste trabalho é construir uma boa base teórica, principalmente acerca dos três elementos que caracterizam o problema de Convergência de Política Ótima, discutir as soluções propostas pelo artigo [1], que propõem o uso de algumas técnicas, como a *Experince Replay* e as *Atualizações Assíncronas*, que também serão discutidas neste trabalho, implementá-las em um cenário diferente, assim como no artigo utilizaremos um ambiente de jogo virtual, porém, apenas o jogo *Flappy Bird* será utilizado, enquanto no artigo diversos jogos de Atari foram usados para testar o novo algoritmo proposto, que ficou conhecido como **Deep Q-Network (DQN)**. Além disso, será apresentado um comparativo entre diferentes implementações de agentes inteligentes, a fim de verificar a eficácia de cada uma das técnicas propostas como solução.

1.3 Organização

Este trabalho está dividido em seis capítulos, contando com este introdutório. Os capítulos 2 e 3 contêm toda a base teórica. No segundo capítulo introduzimos o Aprendizado por Reforço e suas técnicas fundamentais. No terceiro capítulo estendemos estes princípios para casos mais gerais através de Aproximadores Lineares e não Lineares, também discutimos os problema de Convergência de Política Ótima ao combinar aqueles três elementos: *Bootstrapping*, Metodologia *Off-Policy* e *Redes Neurais Artificiais* .

No quarto capítulo verificamos como os problemas de convergência, introduzidos

no capítulo 3, foram contornados no Artigo [1] e apresentamos o nosso próprio teste em um ambiente diferente do proposto no Artigo. No quinto capítulo apresentamos os nossos resultados e fazemos alguns comparativos com diferentes agentes, que atuaram no mesmo ambiente do jogo *Flappy Bird*. Por fim, o sexto capítulo traz as nossas conclusões e oportunidades que podem ser mais exploradas em trabalhos futuros.

Capítulo 2

Aprendizado por Reforço

2.1 Introdução

Há inúmeras formas de se definir o processo de aprendizagem, no entanto, a partir de agora, trabalharemos com a mais comum, que é o processo empírico, ou seja, aquela que se dá através da tentativa e erro, por meio de interações com um ambiente, que gera uma consequência, a qual nos permite avaliar quais são as melhores estratégias para atingir um determinado objetivo. Na maior parte de nossas vidas utilizamos tal forma de aprendizagem, seja para aprender um esporte ou como devemos nos comportar na sociedade , em geral para essas tarefas não necessitamos explicitamente de um professor, mas sim de experiências, conforme nossas ações geram resultados vamos aprendendo [2].

Essa forma de aprendizagem, inerente ao ser humano, que forma as bases do Aprendizado por Reforço (Reinforcement Learning ou simplesmente RL) e a diferencia de todos os outros paradigmas de aprendizado de máquina (machine learning ou ML). Nessa abordagem não há nenhum tipo de supervisor ou *label* indicando quais ações devem ser tomadas. Por outro lado, temos um agente muito mais independente que deve descobrir quais ações devem ser tomadas a fim de maximizar recompensas, por tais ações, a longo prazo [2] [4]. Traremos uma definição formal no capítulo **Markov Decision Processes (MDP)**.

Conforme a definição simplificada acima podemos verificar que RL é realmente diferente das tradicionais abordagens de ML, já que no aprendizado supervisionado teremos necessariamente um conjunto de treinamento com a "resposta correta" (label) disponibilizado por uma entidade externa (supervisor) e a nossa tarefa é obter um modelo preditivo, como nos problemas de regressão e classificação. Enquanto o aprendizado não supervisionado preocupa-se em encontrar padrões e agrupar conjuntos de dados a partir de características similares. Portanto estamos lidando com uma classe diferente de problemas e, por isso, uma outra abordagem é necessária.

2.2 Markov Decision Processes (MDP)

MDP é o modelo matemático que utilizaremos para descrever o problema de aprendizado por tentativa e erro, nele temos um Agente, quem toma as decisões e aprende, dentro de um ambiente, que interage com o agente a cada instante de tempo t, um conjunto de ações que podem ser realizadas dependendo do seu estado atual s, que com probabilidade $T(s, a, s')$, o levará a um estado s', recebendo, por isso, uma recompensa $r(s, a, s')$, que não necessariamente é boa. O modelo ainda conta com uma priorização de recompensas imediatas à recompensas futuras através de um mecanismo de depreciação.

Também é importante dizer que esse modelo sempre satisfaz a propriedade de Markov, logo: dado o estado presente s, o próximo estado s' e a recompensa imediata obtida após a tomada da ação a são independentes dos estados antigos, ou seja, o resultado de uma dada ação depende apenas do estado atual.

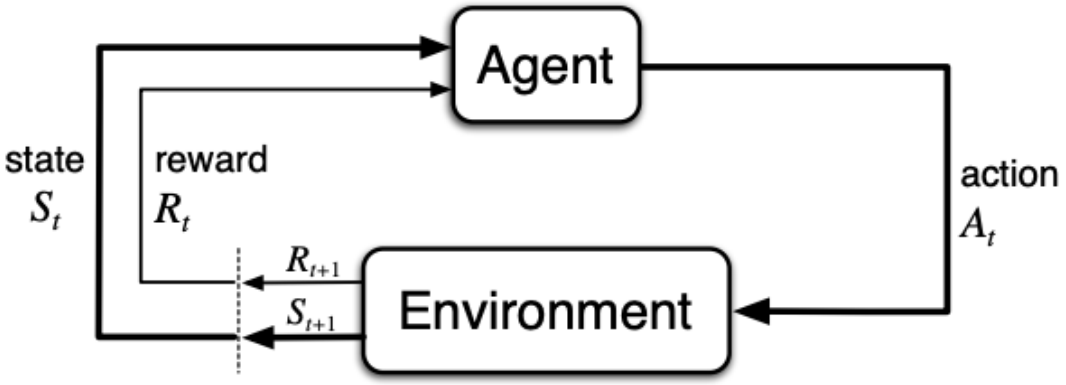

Figura 2.1: Interação agente-ambiente em uma MDP [2].

Portanto podemos definir o modelo através do conjunto $< S, A, T, R >$:

- S Conjunto finito de estados;

$$S = \{s_0, s_1, ..., s_m\}$$

- A Conjunto finito de ações possíveis;

$$A = \{a_0, a_1, ..., a_n\}$$

- $T : S \times A \times S \rightarrow [0, 1]$ Função de probabilidade de Transição $T(s, a, s')$ do estado $s \in S$ para $s' \in S$ através da ação $a \in A$, ou seja, retorna probabilidade de se chegar ao estado s' dado o estado atual s e a açãoa que foi tomada: $P(s'|s, a)$;

- $R : S \times A \times S \rightarrow \mathbb{R}$ Função de recompensa $R(s, a, s')$ retorna a recompensa imediata ao realizar a transição do estado s para o estado seguinte s' através da ação a ;
- fator de depreciação $\gamma \in [0, 1] \subset \mathbb{R}$. Para implementar a política de priorização de recompensas imediatas, as recompensas futuras decaem exponencialmente, ou seja, a cada rodada ou instante de tempo, r é multiplicado por $0 < \gamma < 1$;
- Há um estado inicial $s_0 \in S$;
- Pode existir ou não um ou mais estados finais. $Z \subset S$ onde $Z = \{s_{f_1}, s_{f_2}, ..., s_{f_w}\}$ ou $Z = \emptyset$. Onde s_f representa um estado final e o conjunto Z é aquele a que os estados finais pertencem caso existam.

2.3 Recompensas, Políticas e Valores Ótimos

Informalmente já definimos que o objetivo do aprendizado por reforço é **maximizar o somatório de recompensas a longo prazo**. No entanto, ainda não discutimos exatamente o que esse somatório significa, de maneira intuitiva podemos imaginar uma sequência de ações resultando em recompensas para cada intervalo de tempo t, somando cada um desses valores obtemos:

$$G_t = R_{t+1} + R_{t+2} + \ldots + R_T \tag{2.1}$$

Onde R_T representa a recompensa do estado final. Essa abordagem é bastante eficaz para ambientes que são dividimos naturalmente em subsequências, denominadas *episódios*, que terminam em um **estado terminal**. Por outro lado, essa formulação é problemática em ambientes contínuos, que não possuem necessariamente um estado terminal e não serão abordados neste trabalho.[2]

Apesar da equação 2.1 começar a descrever o nosso problema de maneira mais precisa ainda é necessário acrescentar um fator γ de depreciação, que implementa o conceito de priorizar ações mais recentes, já que os estados futuros são incertos e um estado terminal pode ser encontrado a qualquer instante, assim:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... + \gamma^{T-1} R_T \\ G_t &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.2}$$

Ou ainda:

$$G_t = \sum_{k=1}^{T} \gamma^k R_{t+k} \tag{2.3}$$

Agora que temos a nossa função alvo precisamos definir a estrutura de uma solução, para isso, novamente de forma intuitiva, podemos imaginar que para um dado ambiente possuímos uma *estratégia*, que forneça uma ação a, a qual deve ser tomada dado um estado s. A essa função damos o nome de **política** representada por $\pi : S \rightarrow A$ e uma solução seria uma **política ótima** π^* que retorna uma sequência de ações ótimas, portanto maximiza o retorno (recompensas em longo prazo).

Para chegarmos a uma definição mais precisa é necessário uma maneira de medir o *valor* de um estado, algo que meça quão bom é este estado s, para isso, definimos uma função de valor ou **utilidade**, que nos fornece uma expectativa do somatório das recompensas a partir deste estado s até um estado terminal.

A partir desses dois conceitos podemos definir uma função que avalie o valor de uma política, ou seja, uma função que nos retorne o valor esperado para a soma das recompensas caso uma estratégia seja seguida através de uma política π , portanto:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t|S_t = s] = \mathbb{E}_{\pi}[\sum_{k=0}^{T} \gamma^k R_{t+k}|S_t = s], \forall s \in S \tag{2.4}$$

De maneira análoga podemos definir uma política ótima como aquela que maximiza o valor esperado das recompensas, logo:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{2.5}$$

Agora também podemos introduzir uma função que possa medir o valor de um par estado-ação, ou seja, que avalie para um dado estado s quão bom seria tomar uma ação a, assim obtemos uma nova forma de avaliar uma política π a partir do seu par estado-ação, e a essa função damos o nome de função de valor de ação(*action value function*) ou **Q-Function** que é definida como:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[G_t|S_t = s, A_t = a] = \mathbb{E}_{\pi}[\sum_{k=0}^{T} \gamma^k R_{t+k}|S_t = s, A_t = a], \forall s \in S, \forall a \in A \tag{2.6}$$

Podemos definir uma política ótima a partir de $Q(s,a)$:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{2.7}$$

Com esses novos conceitos podemos reformular o objetivo do Aprendizado por Reforço. Como já sabemos isso passa por maximizar o somatório de recompensas no longo prazo, em outras palavras podemos afirmar que o nosso objetivo é **encontrar uma política π que maximize as recompensas**, ou seja, uma política ótima π^*. É importante dizer que há a possibilidade de existir mais de uma política ótima para

o mesmo problema, mas todas retornam o mesmo valor [2]. Podemos demonstrar que uma política π é melhor ou igual que uma política π' se a utilidade de π no estado s, $V^{\pi}(s)$, é maior ou igual a utilidade de π' para o mesmo estado s, $V^{\pi'}(s)$, conforme a inequação:

$$V^{\pi}(s) \geq V^{\pi'}(s) \forall s \in S \tag{2.8}$$

Estamos em face de um novo problema, já que sabemos que a nossa tarefa é encontrar uma política ótima e ainda não abordamos como isso pode ser realizado. Para tanto, vamos explorar a natureza recursiva já vista em 2.1 . Dado o estado atual s e uma ação a que será tomada, sabemos que com probabilidade $T(s, a, a')$ receberemos uma recompensa $R(s, a, a')$ e que devemos continuar tomando ações ótimas a partir de então, portanto, devemos garantir que elas sejam escolhidas, para tanto, teremos apenas que selecionador o máximo entre cada ação para o próximo estado [3]. Ou seja:

$$\begin{aligned} V^*(s) &= \max_a Q^*(s) \\ Q^*(s,a) &= \sum_{s'} T(s,a,s')[R(s,a,s' + \gamma V^*(s')] \\ V^*(s) &= \max_a \sum_{s'} T(s,a,s')[R(s,a,s' + \gamma V^*(s')] \end{aligned} \tag{2.9}$$

Essas equações são conhecidas como **Equações de Bellman** e a solução delas nos fornecerá os valores ótimos. Podemos ainda simplificar esses conceitos da seguinte forma:

- $V^*(s)$: Recompensa total iniciando no estado s e atuando de maneira ótima;
- $Q^*(s, a)$: Recompensa total iniciando no estado s, tomando ação a e agindo de forma ótima a partir de então.

Para fixar todos esses conceitos podemos exercitá-los em um problema clássico da literatura, o **Grid World**. Nele temos um agente em um ambiente no formato de grid 4x3. Há um estado inicial e a cada instante de tempo t o agente deve tomar uma ação a, apenas movimentos cima, baixo, esquerda e direita, são permitidos, além disso, há uma probabilidade de 0,8 da ação escolhida ser realmente executada e 0,1 para cada outra ação perpendicular. Existem dois estados terminais, um fornece uma recompensa de +1 e o outro de -1, cada um dos outros estados retornam uma recompensa de -0,1, ou seja, há um custo para cada rodada que o jogo não é finalizado. Por fim esse ambiente também conta com um fator de depreciação γ de 0,99.

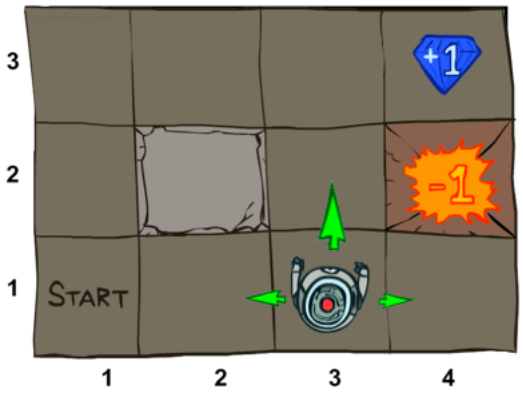

Figura 2.2: Ambiente Grid World [3].

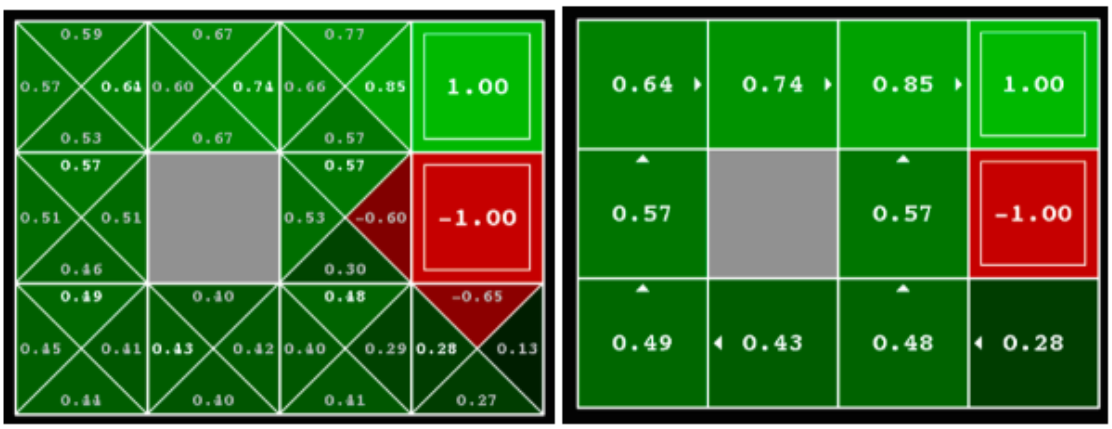

Figura 2.3: Solução Grid World com Q-values e função de valores. Política ótima é representada pelas setas [3].

2.4 Programação Dinâmica

Nas últimas seções definimos as características do problema do Aprendizado por Reforço, o seu propósito e a estrutura da sua solução. A partir desta seção vamos discutir como podemos resolvê-lo e como encontraremos os valores ótimos ou, ao menos, nos aproximaremos deles. Para tal, introduziremos algumas técnicas importantes através da Programação Dinâmica.

Apesar de tal abordagem contar com limitações práticas, já que ela assume um modelo perfeito e um conhecimento total sobre a MDP, ou seja, encaramos o problema com a premissa de que as funções $R(s, a, s')$ e $T(s, a, s')$ são conhecidas, a Programação Dinâmica é uma ferramenta poderosa por construir as bases teóricas para todos os métodos que serão abordados neste trabalho.

2.4.1 Avaliação de Política (Policy Evaluation)

Nossa primeira tarefa será de avaliação, ou seja, dada uma política fixa π e considerando todo MDP conhecido, iremos calcular a value function $V^{\pi}(s)$. Como todos os valores são conhecidos poderíamos apenas resolver o sistema linear, no entanto, para os nossos propósitos iremos executar a política π e construiremos a solução de forma iterativa, ou seja:

$$V^{\pi}_{k+1}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}_{K}(s')] \tag{2.10}$$

Onde $\pi(s)$ representa a ação a que será tomada, segundo a política π, no estado s que leva a estados$'$, dessa forma para cada estado apenas uma ação é considerada, $\pi(s)$, já que estamos seguindo π.

É importante perceber como é realizada cada aproximação de $v_{k+1}(s)$ a partir de $v_k(s')$. Substituímos a antiga value function de s, , pelo valor mais recente da value function de s' mais a recompensa imediata, que é dada pela função $R(s, \pi(s), s')$, também devemos considerar a probabilidade da ação ser devidamente executada, $T(s, \pi(s), s')$, e do fator γ de depreciação de recompensas futuras. Esse tipo de atualização é conhecida como *expected update*. A convergência é garantida conforme $k \to \infty$ [2].

Iterative Policy Evaluation, for estimating $V \approx v_{\pi}$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^{+}$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{a} \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Figura 2.4: Algoritmo de Avaliação de Política [2].

2.4.2 Aprimoramento de Política (Policy Improvement)

Agora que já sabemos como avaliar uma política devemos nos preocupar em como podemos aprimorá-la, uma ideia simples é seguir a política π até um estado s e então testar uma ação a diferente de $\pi(s)$ e verificar se essa nova política π' fornece um $V^{\pi'}(s) > V^{\pi}$, em outras palavras $q^{\pi}(s, \pi'(s)) \geq v^{\pi}(s)$, nesse caso encontramos uma nova política melhorada. Além disso, podemos afirmar que a nova política π' difere apenas em $\pi'(s) = a \neq \pi(s)$.

Podemos estender essa ideia para todos os estados e para toda ação permitida para cada estado, ou seja, podemos utilizar um comportamento guloso para cada estado s buscando maximizar $q^{\pi}(s, a)$, assim:

$$
\begin{aligned}
\pi'(s) &= \arg\max_{a} q^{\pi}(s, a) \\
\pi'(s) &= \arg\max_{a} \mathbb{E}[R_{t+1} + \gamma V_{\pi}(S_{t+1}) | S_t = s, A_t = a] \\
\pi'(s) &= \arg\max_{a} \sum_{s'} T(s, a, s')[R(s, a, s' + \gamma V^{*}(s')]
\end{aligned}
\tag{2.11}
$$

Se obtermos uma nova política π' tão boa quanto π , ou seja, $V^{\pi'}(s) = V^{\pi}$, ou ainda $q^{\pi}(s, \pi'(s)) = v^{\pi}(s)$ então tanto π quanto π' são políticas ótimas.

2.4.3 Iteração de Política (Policy Iteration)

Combinando as duas técnicas acima podemos construir um algoritmo que recebe uma política qualquer e nos fornece uma política ótima. Portanto, podemos aprimorar e reavaliar uma política π aleatória até que ela convirja para π^{*}.

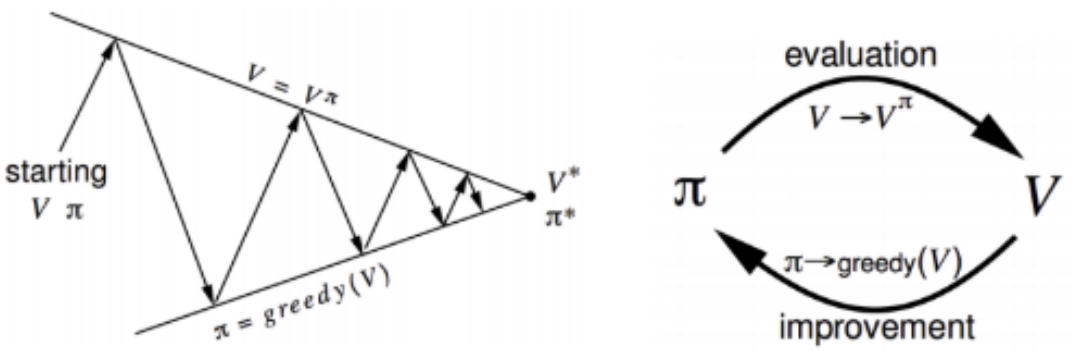

Figura 2.5: Combinação de avaliação e aprimoramento de política retornando valores ótimos [2].

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   - $\Delta \leftarrow 0$
   - Loop for each $s \in \mathcal{S}$:
     - $v \leftarrow V(s)$
     - $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s)) \big[r + \gamma V(s')\big]$
     - $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   - *old-action* $\leftarrow \pi(s)$
   - $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$
   - If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*

   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figura 2.6: Algoritmo Policy Iteration [2].

2.4.4 Iteração de Valor (Value Iteration)

Uma outra ideia razoável para resolvermos a MDP e, assim encontrarmos os valores ótimos, é buscar maximizar diretamente a value function e implicitamente obtemos uma política ótima. Ou seja, devemos construir $V^*(s)$ iterativamente através da escolha de ações que maximizem a value function, podemos representar essa ideia através da equação:

$$V_{k+1}(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_K(s')] \tag{2.12}$$

Assim como a Iteração de Política o algoritmo de Iteração de Valor tem convergência garantida conforme $k \to \infty$ [2].

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $\quad v \leftarrow V(s)$
| $\quad V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

Figura 2.7: Algoritmo Value Iteration [2].

2.4.5 Iteração de Política Vs Iteração de Valor

As duas técnicas podem ser utilizadas para se obter os valores ótimos, $V^*(s)$ e π^*, quando o agente possui todo o conhecimento a respeito da MDP, $T(s, a, s')$ e $R(s, a, s')$. A Iteração de Política é computacionalmente mais eficiente, $O(S^2)$ contra $O(S^2A)$, apesar de cada passo ser mais custoso [3]. A figura a seguir faz uma comparação entre a forma que os algoritmos realizam suas atualizações a cada iteração.

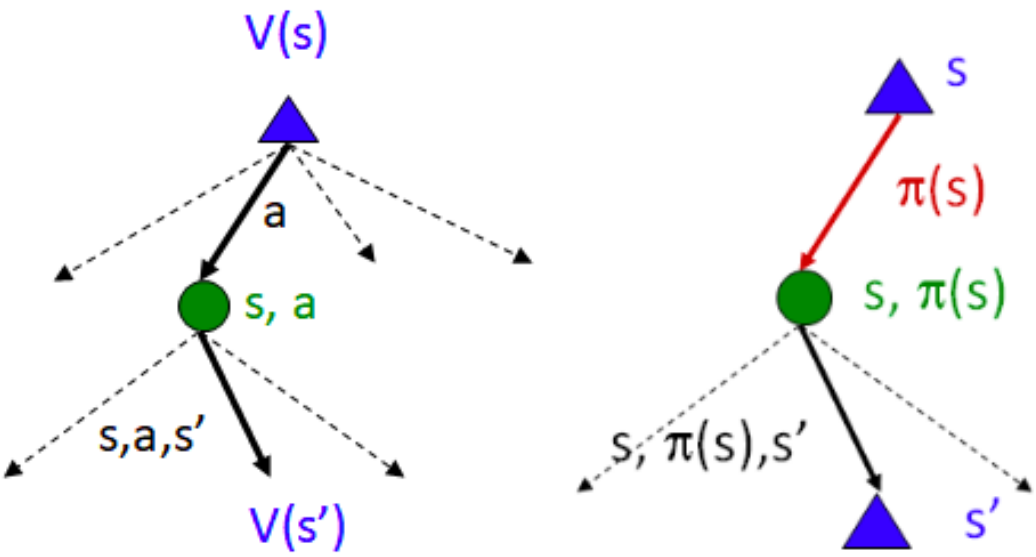

Figura 2.8: Comparativo tomada de ações entre Iteração de Valor e Iteração de Política [3].

2.5 Model-Free Prediction

A partir de agora vamos estudar técnicas que não assumem total conhecimento das dinâmicas da MDP, mas que aprendem através de experiências, seja na forma de interações reais com o ambiente ou a partir de simulações, a única fonte de conhecimento necessária é uma sequência de amostras. Até então estávamos abordando problemas que assumem conhecimento prévio tanto da função $T(s, a, s')$ quanto de $R(s, a, s')$, no entanto, através das técnicas discutidas daqui para frente seremos capazes de encontrar os mesmos valores ótimos mesmo sem o conhecimento dessas funções [2].

Além disso, como nas seções anteriores, vamos começar com uma tarefa mais simples, que é a de predição, ou seja, dada uma política π nosso objetivo será avaliar qual é o retorno $V^{\pi}(s)$ obtido ao segui-la.

2.5.1 Monte Carlo Prediction (MC Prediction)

Nosso primeiro método tem como base a média dos retornos das amostras. Para garantir que esses retornos sejam bem definidos precisamos dividir a experiência em *episódios*, e cada episódio deve ter um fim independentemente das ações escolhidas, além disso, é importante definir que um episódio é a sequência de estado, ação e recompensa desde o estado inicial até um estado final, em um jogo, por exemplo, um episódio representaria uma partida completa desde seu início. Ao fim do episódio novas médias são obtidas e os valores então são atualizados, conforme o número de episódios observados aumenta nossa estimativa se aproxima de $V^{\pi}(s)$ [2].

Mais de um tipo de comportamento é possível para se obter as médias entre os episódios. O nosso objetivo é estimar $V^{\pi}(s)$, ou seja, a utilidade da política π para o estado s. Cada ocorrência de s em um dado episódio pode ser chamada de visita a s. Porém s pode ser visitado mais de uma vez no mesmo episódio, neste trabalho iremos considerar a primeira visita a s para estimar $V^{\pi}(s)$, já que esse tipo de abordagem já foi amplamente estudado para esse caso. Dessa forma, seguindo essa metodologia, para cada passo de um episódio nossa value function é estimada pela soma entre as recompensas já acumuladas nesse episódio e a recompensa imediata pelo passo atual, ou seja:

$$H \leftarrow G_t + R_{t+1} \tag{2.13}$$

Além disso, ao fim de cada episódio devemos atualizar a média da nossa estima-

tiva:

$$
\begin{aligned}
N(s) &\leftarrow N(s) + 1 \\
S(s) &\leftarrow S(s) + G_t \\
V(s) &= S(s)/N(s) \\
V(s) &\rightarrow V^{\pi}(s) \text{ quando } \; N(s) \rightarrow \infty
\end{aligned}
\tag{2.14}
$$

Onde $N(s)$ representa um contador de espisódios, $S(s)$ o somatório da recompensa de todos os episódios. Portanto podemos escrever a nossa atualização da seguinte forma:

$$
\begin{aligned}
V(s_t) \leftarrow V(s_t) + (1/N(s_t))(G_t - V(s_t)) \\
V(s_t) \leftarrow V(s_t) + \alpha(G_t - V(s_t))
\end{aligned}
\tag{2.15}
$$

Usualmente chamamos o fator alfa de *learning rate*, $0 < \alpha < 1$, e a diferença entre o valor obtido na última experiência pela última estimativa de erro de Monte Carlo:

$$
\delta = G(t) - V(s_t) \tag{2.16}
$$

Dessa forma, obtemos o seguinte algoritmo:

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow G + R_{t+1}$
    Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
      Append G to $Returns(S_t)$
      $V(S_t) \leftarrow$ average($Returns(S_t)$)

Figura 2.9: Algoritmo Monte Carlo Prediction [2].

É importante deixar claro que este algoritmo não será utilizado em nenhuma implementação deste trabalho, no entanto, ele é fundamental para o entendimento da técnica de bootstrapping, que será apresentada a partir da próxima seção, e é um dos elementos que caracterizam o problema de Convergência de Política Ótima.

2.5.2 Predição por diferenciação temporal (Temporal-Difference Prediction)

Assim como no Monte Carlo a técnica de Temporal-Difference (TD), não assume total conhecimento das dinâmicas da MDP, ou seja, as funções $T(s, a, s')$ e $R(s, a, s')$ são desconhecidas, por isso, deve aprender através de experiências. Porém, neste algoritmo não é necessário esperar até o final do episódio, aqui podemos realizar atualizações com base em estimativas dos estados futuros, a esse processo damos o nome de **bootstrap** [2]. Dessa forma com apenas um passo obtemos uma estimativa de $V^{\pi}(s)$ que também converge conforme o número de experiências aumentam. Podemos representar a estimativa da value function do estado s, $V^{\pi}(s)$ como a soma da recompensa imediata, que é fornecida pelo ambiente ao se tomar a ação a ao estar no estado s, mais a estimativa da value function do próximo estado $V^{\pi}(s')$ descontando um fator γ de depreciação conforme a equação:

$$H \leftarrow R_{t+1} + \gamma V(s_{t+1}) \tag{2.17}$$

Portanto a atualização da nossa estimativa fica:

$$V(s_t) \leftarrow V(s_t) + \alpha(R_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{2.18}$$

Assim como no método anterior podemos encontrar o erro de TD através da diferença entre o nosso valor obtido pela última estimativa:

$$\delta = R_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.19}$$

Conforme a estimativa da value Function de cada estado vai se aprimorando o processo se aproxima da convergência. Através dessa técnica obtemos o seguinte algoritmo:

Tabular TD(0) for estimating v_{π}

Input: the policy π to be evaluated
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $V(s)$, for all $s \in \mathcal{S}^{+}$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  until S is terminal

Figura 2.10: Algoritmo TD(0) Prediction [2].

2.5.3 Monte Carlo Vs Temporal Difference

A diferença entre os métodos está na forma como estimamos a value function. Como já discutimos o Monte Carlo necessita esperar até o final do episódio, enquanto o TD espera apenas um único passo, por isso, damos o nome a esse tipo de bootstrapping de **TD(0)**, que será a técnica utilizada ao longo desse trabalho. É possível fazer a estimativas utilizando mais passos, um bootstrapping entre o TD(0) e o Monte Carlo, já que podemos usar mais de um passo mas não necessariamente chegar até o final do episódio, esse algoritmo existe e damos a ele o nome de TD(λ), mas não entraremos em detalhes neste trabalho. A figura ilustra como cada método realiza a estimativa de $V^{\pi}(s)$.

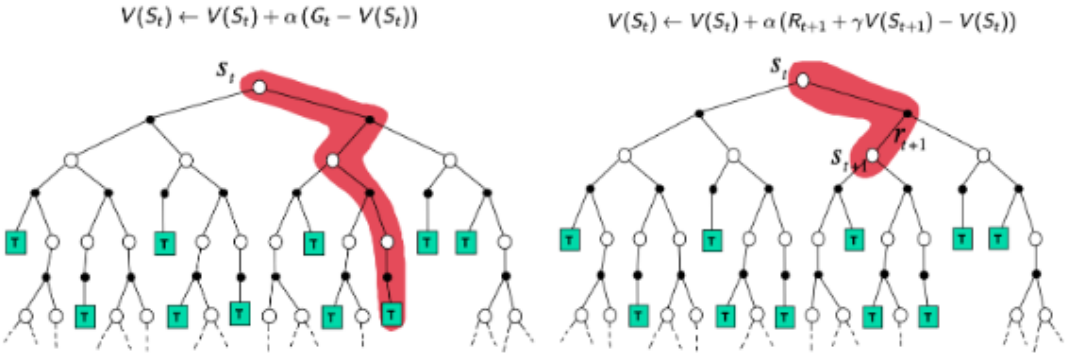

Figura 2.11: Comparativo entre a estimativa de cada método [4].

Com amostras suficientes e um learning rate α suficientemente pequeno ambos os métodos convergem para $V^{\pi}(s)$. No entanto, TD(0) converge primeiro [4].

2.6 Model Free Control

Vamos repetir a metodologia da seção de Programação Dinâmica onde resolvemos primeiro a tarefa preditiva, e agora iremos introduzir métodos que resolvem o problema de controle, ou seja, como podemos construir os valores ótimos, tanto uma política ótima π^* quanto a value function ótima $V^*(s)$ ou a action-value function ótima $Q^*(s, a)$, quando as dinâmicas da MDP são desconhecidas, ou seja, os valores das funções $R(s, a, s')$ e $T(s, a, s')$ não são conhecidos. Para isso iremos utilizar a mesma estratégia de avaliar e aprimorar uma política já existente, conforme representado pelo esquema a seguir:

$$\pi^0 \xrightarrow{A} Q^{\pi^0} \xrightarrow{M} \pi^1 \xrightarrow{A} Q^{\pi^1} \xrightarrow{M} \pi^2 \xrightarrow{A} \dots \xrightarrow{M} \pi^* \xrightarrow{A} Q^*$$

Onde $\xrightarrow{A}$ representa uma avaliação de política e $\xrightarrow{M}$ denota uma melhora de política [2].

A tarefa preditiva será resolvida através dos métodos de Monte Carlo e TD(0) apresentados na seção anterior. Já o aprimoramento de política pode ser feito tornando a política gulosa em relação a value-function. Porém, apenas a value-function não é suficiente para obtermos uma política, por isso, usaremos a action-value function, dessa forma somos capazes de obter uma ação que maximize o valor de um estado, conforme a equação a seguir:

$$\pi(s) = \arg\max_a q(s, a) \tag{2.20}$$

Através dessa ideia podemos construir uma política π^{k+1} gulosa em relação a $Q^{\pi^k}(s, \pi^{k+1}(s))$, como já discutimos anteriormente sabemos que essa nova política será melhor ou tão boa quanto a política atual, nesse caso ambas são ótimas [2], Ou seja:

$$\begin{aligned} Q^{\pi^k}(s, \pi^{k+1}(s) &= Q^{\pi^k}(s, \arg\max_a Q^{\pi^k}(s, a)) \\ &= \max_a Q^{\pi^k}(s, a) \\ &\geq Q^{\pi^k}(s, \pi^k(s)) \\ &\geq V^{\pi^k}(s) \end{aligned} \tag{2.21}$$

No entanto, ainda há uma questão importante que precisa ser resolvida. Ao seguir uma política gulosa talvez alguns estados jamais sejam visitados, por isso, devemos garantir uma exploração uniforme dos pares estado-ação. Esse é um grande dilema do Aprendizado por Reforço, ao mesmo tempo que desejamos que o nosso agente atue de forma ótima seguindo a melhor política encontrada até o momento, é fundamental que procure por políticas alternativas e compare seus resultados, ou seja, queremos tentar ações novas, mas de uma forma equilibrada, que não comprometam completamente a performance já obtida [2]. Nas próximas seções apresentados alguns métodos que resolvem esse problema.

2.6.1 Monte Carlo Control

Nosso primeiro método utiliza a estratégia de Monte Carlo para avaliar estimativa da política e ataca o **Dilema Exploração - Intensificação (Exploration - Exploitation)** de maneira bem simples. Cada episódio começa com um par estado-ação aleatório e todos os pares possuem uma probabilidade maior que zero de ser sorteado, essa técnica é conhecida como *Exploring Starts* [2]. Dessa forma obtemos o seguinte algoritmo:

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
  $\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
  $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
  $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
  Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ such that all pairs have probability > 0 (exploring starts)
  Generate an episode starting from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow G + R_{t+1}$
    Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
      Append G to $Returns(S_t, A_t)$
      $Q(S_t) \leftarrow \text{average}(Returns(S_t, A_t))$
      $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

Figura 2.12: Algoritmo Monte Carlo Control com Exploring Starts. [2].

Apesar da prova da convergência desse método parecer bem intuitivo, já que se a action-value function convergir para um valor subótimo a política vai mudar, logo a action-value function mudará novamente, ou seja, a estabilidade entre a política e a action-value function só acontece quando ambos convergem para um valor ótimo. Ainda não há uma prova matemática formal que demonstre essa convergência [2].

Embora o método acima seja uma solução para o problema, a forma como ele contorna o Dilema Exploração - Intensificação é bastante rústica, assim, podemos pensar em outra estratégia mais elaborada. Na realidade há duas abordagens clássicas na literatura que dividem os métodos em duas classes. A primeira lida com essa questão da forma que trabalhamos até aqui, ou seja, a ideia consiste em avaliar e aprimorar a própria política que está sendo utilizada para a tomada de decisões, esse tipo de técnica é chamada de **On-Policy**. Enquanto a outra classe de métodos, conhecidos como **Off-Policy** preocupa-se em avaliar e aprimorar uma política que não está sendo utilizada para a tomada de decisões, ou seja, há duas políticas, uma é responsável por orientar o agente enquanto a outra será otimizada, conforme veremos nas seções seguintes.

2.6.2 On-Policy MC Control sem 'Exploring Starts'

Novamente iremos utilizar uma estratégia a técnica de Monte Carlo para estimar a action-value function da política que seguimos. No entanto, não iremos utilizar uma política gulosa, já que sabemos que *sempre* selecionar a ação que maximiza $Q(s,a)$ impossibilita a exploração de todos os pares ação-estado. Por isso, devemos pensar em outra forma de contornar esse dilema. Sabemos que ao iniciar o processo On-Policy ainda não coletamos muitas amostras, portanto ainda não temos muito conhecimento, logo temos uma ótima oportunidade de explorar mais pares ação-estado. Conforme vamos coletando mais amostras com um alto índice de exploração

o conhecimento do nosso agente cresce, portanto, é desejável um comportamento menos exploratório e mais guloso em relação aos valores já obtidos.

Em outras palavras, buscamos uma política que com uma probabilidade $\varepsilon > 0$ selecione uma ação aleatória e com $1 - \varepsilon$ atue de forma gulosa. Ou seja, queremos uma distribuição $\pi(a|s) > 0$ para todo $s \in S$ e $a \in A(s)$, onde $A(s)$ representa a quantidade de ações possíveis em um dado estado s. Como todas as ações disponíveis em um estado possuem a mesma chance de serem selecionadas, obtemos a probabilidade $\frac{\varepsilon}{A(s)}$ para uma ação não gulosa, e $1 - \varepsilon + \frac{\varepsilon}{A(s)}$ para ação gulosa [2].

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$
Initialize:
  $\pi \leftarrow$ an arbitrary ε-soft policy
  $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
  $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow G + R_{t+1}$
    Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
      $A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)
      For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Figura 2.13: Algoritmo On-Policy Monte Carlo ε-guloso . [2].

Podemos mostrar que para uma política ε - gulosa em relação a $Q_\pi(s, \pi'(a))$ a política π' é melhor ou tão boa quanto π, dessa forma ambas são ótimas, de maneira similar as situações discutidas em seções anteriores, mas dessa vez a partir das seguintes equações [2]:

$$\begin{aligned} Q^\pi(s, \pi'(s)) &= \sum_a \pi'(a|s) Q^\pi(s,a) \\ &= \frac{\varepsilon}{A(s)} \sum_a Q^\pi(s,a) \quad + (1-\varepsilon) \max_a Q^\pi(s,a) \\ &\geq \frac{\varepsilon}{A(s)} \sum_a Q^\pi(s,a) \quad + (1-\varepsilon) \sum_a (\pi(a|s) - \frac{\varepsilon}{A(s)}) \frac{1}{(1-\varepsilon)} Q^\pi(s,a) \\ &= \frac{\varepsilon}{A(s)} \sum_a Q^\pi(s,a) \quad \frac{\varepsilon}{A(s)} \quad + \sum_a \pi(a|s) Q^\pi(s,a) = V^\pi(s) \end{aligned} \tag{2.22}$$

2.6.3 Sarsa(0): On-Policy TD(0) Control

O nosso próximo método utiliza a técnica de TD(0) para avaliar a estimativa da action-value function. Na seção em que estudamos como TD(0) avalia uma estimativa utilizamos a value-function, a abordagem ainda é a mesma, apenas teremos que alterar a tarefa preditiva para action-value function, conforme já discutimos a value-function não é suficiente para construção de uma política gulosa, portanto a única alteração necessária será na atualização da estimativa que ficará da seguinte maneira:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{2.23}$$

Assim como na tarefa preditiva do TD(0) essa atualização é feita para todo estado s_t não terminal, ou seja, somaremos a recompensa imediata R_{t+1}, fornecida pelo ambiente ao se tomar a ação a_t ao estar no estado s_t, com a estimativa da action value function da próxima combinação de estado-ação considerando o fator de depreciação, no entanto, caso s_t seja terminal então $Q(s_{t+1}, a_{t+1})$ vale zero, já que não existirá um estado futuro. Dessa maneira, para toda atualização utilizaremos os elementos: $s_t, a_t, R_{t+1}, s_{t+1}, a_{t+1}$ e daí o nome **SARSA** [2].

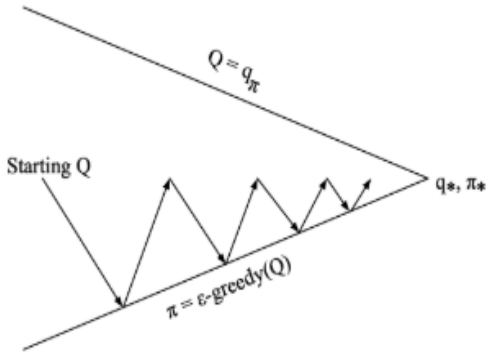

Figura 2.14: Avaliação e Aprimoramento de Q-Values para SARSA(0) [4]. Seguimos exatamente a mesma lógica de Iteração de Política, mas dessa vez utilizamos um bootstrapping TD(0)

A outra questão fundamental para um método de controle é como lidar com o Dilema Exploração - Intensificação. Bem como no método anterior iremos utilizar uma política ε-gulosa para contornar o problema. Assim SARSA(0) converge da mesma maneira que o método anterior, e ainda se beneficia da estratégia de bootstrapping TD(0), portanto não há necessidade de esperar o fim do episódio para realizar cada atualização. Com isso o algoritmo SARSA pode ser escrito da seguinte maneiro:

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figura 2.15: Algoritmo On-Policy SARSA(0) [2].

Novamente devemos perceber que embora nenhuma implementação deste trabalho utilize alguma dessas técnicas On-Policy, elas são fundamentais para o bom entendimento do boostrapping na tarefa de Controle, da utilização de política ε-gulosa e são a base para a construção da metodologia Off-Policy, que é fundamental para este trabalho.

2.7 Metodologia Off-Policy

Até agora estudamos métodos On-Policy, ou seja, aqueles que avaliam e aprimoram a própria política que define o comportamento do agente. Daí que o Dilema Exploração- Intensificação se acentua, já que desejamos encontrar uma política ótima, mas também queremos um comportamento exploratório ao mesmo tempo e na mesma política. A partir dessa seção vamos introduzir uma nova técnica, que utiliza duas políticas diferentes, uma que deve ser otimizada conhecida como **target policy** e outra que define o comportamento exploratório do nosso agente denominada **behavior policy**, ou seja, teremos uma política a ser otimizada e uma política que orienta o comportamento do agente [2].

De maneira geral podemos dizer que os métodos On-Policy são mais simples e, por isso, podem convergir mais rapidamente que os métodos Off-Policy, que são mais poderosos, no sentido de enfrentar o Dilema Expliração- Intensificação de uma forma mais inteligente. Além disso, podemos ver os métodos On-Policy como um caso especial de Off-Policy, na qual a target policy e a behavior são iguais.

2.7.1 Q-Learning

O Q-Learning é considerado uma das maiores inovações do Aprendizado por Reforço, introduzido por Watkins em 1989, porque ele não só funciona como uma técnica Off-Policy como também garante que a target policy irá convergir para uma política ótima desde que todos os pares estado-ação continuem sendo visitados e atualizados,

independetemente da behavior policy seguida [2], tal propriedade é garantida através da atualização:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.24}$$

Tal propriedade simplifica muito o algoritmo e a garante a convergência para $Q^*(s, a)$ desde que os pares ação-estado continuem sendo visitado e seus valores sejam atualizados. Na figura a seguir podemos verificar como a atualização do Q-Learning funciona, os arcos ao redor dos nós, que representam quais ações podem ser escolhidas, simbolizam a maximização da próxima ação.

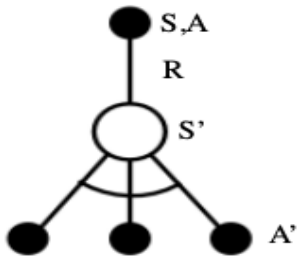

Figura 2.16: Atualização Q-Learning [4].

Dessa forma, podemos verificar que a target policy, que sempre é aprimorada de forma gulosa em relação a $Q(s, a)$, ou seja, $\pi'(s) = \max_{a'} Q(s', a')$, e a behavior policy também será aprimorada, já que assim como nos algoritmos anteriores utilizaremos uma política ε-gulosa em relação a $Q(s, a)$, para orientar o nosso agente. Com isso, a atualização do algoritmo Q-Learning segue exatamente a equação 2.24, ou seja, recebemos do ambiente a recompensa imediata ao se tomar a ação a, que é fornecida pela política ε-gulosa que orienta o agente (behavior policy), mais a estimativa gulosa, em relação as ações de do próximo estado s', que aprimora a target policy multiplicado pelo fator de depreciação [4].

Seguindo essa lógica obtemos o seguinte algoritmo:

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$
  until S is terminal

Figura 2.17: Algoritmo Off-Policy Q-Learning [2].

Durante esse capítulo apresentamos as principais definições do Aprendizado por Reforço, e dois dos três elementos essenciais para este trabalho, o Bootstrapping e a metodologia Off-Policy, que nos forneceram o algoritmo Q-Learning, que será a base para a implemtação desse trabalho. No capítulo seguinte vamos introduzir a ideia de Aproximadores Lineares e Não Lineares para o problema de Aprendizado por Reforço, assim estaremos prontos para discutir o Problema de Convergência de Política Ótima nos capítulos seguintes. Para um estudo mais aprofundado destes temas recomendo as seguintes referências [2], [4] e [3].

Capítulo 3

Aprendizado por Reforço e Aproximadores de Funções

Nas seções anteriores estudamos alguns métodos que resolvem diversas questões do Aprendizado por Reforço. Porém essas soluções são construídas através de método tabulares, ou seja, vamos construindo grandes *arrays* iterativamente com os valores mais atuais. Apesar dessa abordagem ser fundamental como base teórica ela não é aplicável para a maioria dos problemas reais, que possuem um espaço combinatório muito grande, muitas vezes maior que o número de átomos no universo, por exemplo. Com problemas grandes e recursos de computação e tempo limitados devemos repensar nosso objetivo, que agora passa a ser encontrar uma boa aproximação da solução ótima [2].

Para esse tipo de problema o nosso maior desafio é a capacidade de generalização dos nosso métodos. Já que maioria dos estados encontrados durante a construção da solução nunca haviam sido vistos anteriormente e, ao mesmo, são bem parecidos e apresentam características bem similares de estados já conhecidos. Em outras palavras, a principal dificuldade nesse caso é construir uma boa solução a partir de um subconjunto, que generalize, através de aproximações, para o conjunto muito maior de estados e ações.

Esse tipo de generalização que buscamos já foi largamente estudado e é conhecido como **Aproximadores de Função (Function Approximation)** e é uma técnica amplamente utilizada nos problemas de Aprendizado Supervisionado, que a partir de amostras pode generalizar uma função. Teoricamente qualquer tipo de aproximador pode ser utilizado, porém alguns se enquadram melhor a essa tarefa do que outros. Isso se deve principalmente às especificidades do Aprendizado por Reforço, que introduz problemas como o *bootstrapping* e as recompensas, que são encaradas como sinais de labels com atrasos pelo aproximador [2]. Neste trabalho discutiremos alguns aproximadores, no entanto, para implementação será utilizado Redes Neurais Artificiais.

Novamente iremos organizar os nossos estudos por métodos de Predição e de Controle On-Policy e Off-Policy. Lembrando que nenhum método On-Policy será implementado, mas para bom entendimento da metodologia Off-Policy com Aproximadores de Função eles são fundamentais.

3.1 Predição com Aproximadores

Bem como nas seções anteriores a nossa primeira tarefa será a de avaliar uma política, porém, dessa vez, não a faremos através de uma representação por tabelas, ela será feita por uma função parametrizada por um vetor de pesos $W \in \mathbb{R}^d$, dessa forma podemos escrever:

$$\hat{V}(s, W) \approx V^{\pi}(s) \tag{3.1}$$

Onde W pode representar o vetor de pesos para um conjunto de *features* no caso de um aproximador linear, também podendo representar o vetor de pesos que conecta cada camada de uma Rede Neural, ou seja, a estrutura do vetor W está intrinsecamente ligada ao Aproximador utilizado. Mas em linhas gerais pode-se admitir que a dimensionalidade de W é muito menor do que o número de estados, ou seja, $d \ll |S|$. Portanto, ao atualizar o valor de um estado através do vetor W diversos outros estados também serão atualizados de uma só vez [2].

Todos os métodos estudados funcionam através de atualizações dos valores estimados. Podemos imaginar cada atualização como um indicador do comportamento desejado, ou seja, a cada atualização dizemos que a estimativa para o estado s deve ser mais parecida com alvo U_t. Dessa forma, podemos escrever uma atualização como $S_t \rightarrow U_t$. Assim a atualização da Value-Function através do Monte Carlo e TD(0) são respectivamente $S_t \rightarrow G_t$ e $S_t \rightarrow R_{t+1} + \gamma\hat{V}(S_{t+1}, W_t)$. Até agora essas atualizações são triviais, apenas alteramos o valor da Value-Function correspondente ao estado s pelo novo valor estimado, enquanto os valores correspondentes aos demais estados se mantêm inalterados [2].

A partir de agora permitimos outras formas de realizar essas atualizações, que ao fazer a alteração em um estado também generalizam para diversos outros estados. Isso é feito através dos aproximadores, que utilizam cada atualização $S_t \rightarrow U_t$ como uma amostra de treino, exatamente como no Aprendizado Supervisionado, mas com um atraso na label, que é representada pela recompensa. Dessa forma obtemos um regressor para a value-function através das atualizações realizadas a cada experiência, que funcionam como um conjunto de treinamento. Portanto, teoricamente qualquer método de aproximação pode ser utilizado e essa mesma ideia pode ser estendida para a Q-Function [2].

3.1.1 Mínimos Quadrados

Até agora não foi necessário discutir uma métrica que avalie a qualidade de uma estimativa, já que nos métodos tabulares acabam convergindo para o valor real e uma atualização de um estado não interfere nos valores correspondentes a outros estados. Mas como já começamos a discutir nenhuma dessas duas afirmações são válidas quando utilizamos aproximadores. Além disso, como também já sabemos, a dimensionalidade de W é muito menor que o número de estados, portanto, inevitavelmente alguns estados terão um valor mais preciso do que outros. Dessa forma devemos especificar o quanto nos preocupamos com o erro em cada estado, para isso, definimos uma distribuição de estados $\mu \geq 0$, $\sum_s(s) = 1$, que representa um grau de exatidão para cada estado. Por fim, definimos erro como a diferença entre o valor estimado $\hat{V}(s, W)$ e o valor real $V^\pi(s)$, assim definimos o *Erro Médio Quadrático (EMQ)* ou $(\overline{VE})$ como:

$$\overline{VE}(W) = \sum_{s \in S} \mu(s)[V^\pi(s) - \hat{V}(s, W)]^2 \tag{3.2}$$

Também podemos definir $\mu(s)$ como a fração de tempo gasta em cada estado, com isso, quanto maior o tempo gasto em s mais devemos nos preocupar em minimizar o seu erro. Para problemas organizados em episódios, que é o foco deste trabalho, $\mu(s)$ também depende de como os estados iniciais de cada episódio são escolhidos, logo:

$$\begin{aligned} \eta(s) &= h(s) + \sum_{\overline{s}} \eta(\overline{s}) \sum_a \pi(a|\overline{s}) p(s|\overline{s}, a) \\ \mu(s) &= \frac{\eta(s)}{\sum'_s \eta s'} \end{aligned} \tag{3.3}$$

Onde $h(s)$ representa a probabilidade de um episódio começar em cada estado s e $\eta(s)$ representa cada passo gasto, em média, em um estado s, em um único episódio. Além disso, um passo é gasto em um estado s se o episódio começou em s ou se houve uma transição de um estado $\overline{s}$ para s [2].

A ideia de se usar o Erro Médio Quadrático é encontrar um ótimo global, ou seja, um W^* que satisfaça $\overline{VE}(W^*) \leq \overline{(}VE)(W)$ para todos os valores de W possíveis. Porém, nem todos aproximadores conseguem cumprir tal condição, as Redes Neurais Artificiais, por exemplo, muitas vezes convergem para ótimos locais, ou seja, essa condição só é válida para uma pequena vizinhança de W^*, às vezes, elas podem até mesmo divergir [2]. Mesmo sem tal garantia de convergência, nas próximas seções, iremos abordar modos de contornar esse problema e encontrar ótimos globais.

3.1.2 Gradiente Descendente Estocástico (Stochastic Gradient Descent SGD)

Nesta seção iniciaremos um breve estudo sobre alguns dos Aproximadores, especialmente os Aproximadores Lineares e as Redes Neurais Artificiais. Nossa primeira classe de métodos funcionam através da técnica do Gradiente Descendente Estocástico. Aqui o nosso W representa um vetor coluna composto por números reais $(W = w_1, w_2, ..., w_d)^T$, dessa forma, temos um $\hat{V}(s, W)$ que é diferenciável em W para todo $s \in S$. Para cada passo $t = 0, 1, 2, ...$ iremos atualizar W, para tanto, usamos uma notação W_t para representar o vetor coluna W em um determinado passo t.

Vamos assumir que todos os estados aparecem nas amostras seguindo a distribuição μ. Nossa tarefa agora é minimizar $\overline{VE}$, para resolver tal problema o *SGD* ajusta W, para cada amostra obtida, na direção que reduz o erro ao máximo, de acordo com uma pequena taxa α, que representa o nosso *learning rate* [2]. Conforme a equação:

$$\begin{aligned} W_{t+1} &= W_t - \frac{1}{2}\alpha\nabla[V^\pi(s_t) - \hat{V}(s_t, W_t)]^2 \\ &= W_t + \alpha[V^\pi(s_t) - \hat{V}(s_t, W_t)]\nabla\hat{V}(s_t, W_t) \end{aligned} \tag{3.4}$$

Onde $\nabla f(W)$ representa, para qualquer função escalar $f(W)$, o gradiente com relação ao vetor W, ou seja:

$$\nabla f(W) = (\frac{\partial f(W)}{\partial w_1}, \frac{\partial f(W)}{\partial w_2}, ..., \frac{\partial f(W)}{\partial w_d})^T \tag{3.5}$$

O nome Gradiente Descendente é adequado, já que para cada passo o vetor W_t é ajustado proporcionalmente ao gradiente negativo obtido através do EQM da amostra, e é exatamente nessa direção que o erro cai mais rapidamente. Além disso, ele é estocástico porque utiliza uma única amostra aleatória, que é definida de acordo com a transição dos estados, para realizar a atualização de W. Após várias amostras e com um *alpha* suficientemente pequeno, que decai com o tempo, temos garantia de convergência [2].

Agora que já temos uma boa ideia de como o SGD funciona em cenário praticamente ideal podemos focar o nosso estudo para o caso em que a nossa função alvo não é precisamente a value-function, mas na realidade uma aproximação, já que o real valor de $V^\pi(s)$ é desconhecido, utilizaremos U_t para representar nossa função alvo para uma amostra t. Logo, podemos substituir o valor exato $V^\pi(s)$ da equação 3.4 por U_t.

$$W_{t+1} = W_t + \alpha[U_t - \hat{V}(s_t, W_t)]\nabla\hat{V}(s_t, W_t) \tag{3.6}$$

Caso U_t seja uma estimativa sem *bias* como é o caso do método de Monte Carlo, onde $U_t = G_t$, a convergência é garantida. No entanto, se houver bootstrapping como em TD(0), onde $U_t = R_{t+1} + \gamma \hat{V}(s_{t+1}, W)$, a convergência só é garantida para aproximadores lineares [2], nas próximas seções também discutiremos como contornar esse problema para Redes Neurais Artificias. Os algoritmos para cada métodos são apresentados a seguir:

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

Figura 3.1: SGD Algoritmo com Monte Carlo [2]

.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S' is terminal

Figura 3.2: SGD Algoritmo com TD(0) [2]

.

3.2 Aproximadores Lineares

Começamos nossos estudos pelos Aproximadores Lineares. Nesse cenário a nossa aproximação $\hat{V}(., W)$ é uma função linear em relação ao vetor de pesos W. Também utilizamos um vetor $X(s) = (x_1(s), x_2(s), ..., x_d(s)^T)$ denominado *feature vector* que representa um estado através de suas características ou propriedades. Dessa forma,

obtemos:

$$\hat{V}(s, W) = W^T X(s) = \sum_{i=1}^{d} w_i x_i(s) \tag{3.7}$$

Portanto a atualização do **SGD** para o caso linear, seguindo a equação 4.1 é:

$$W_{t+1} = W_t + \alpha[U_t - \hat{V}(s_t, W_t)]X(s) \tag{3.8}$$

Os Aproximadores Lineares possuem um único ponto ótimo, portanto não há o risco deles ficarem presos em torno de um ótimo local. Podemos, inclusive, provar a convergência do Semi-Gradiente TD(0) aplicado com um Aproximador Linear, a um ponto muito próximo do ótima global [2], através da equação 3.8, logo:

$$\begin{aligned} W_{t+1} &= W_t + \alpha(R_{t+1} + \gamma W_t^T X_{t+1} - W_t^T X_t)X_t \\ &= W_t + \alpha R_{t+1} X_t - X_t(X_t - \gamma X_{t+1})^T W_t \end{aligned} \tag{3.9}$$

Para simplificar a notação utilizamos $X_t = X(s_t)$, além disso, podemos escrever:

$$\begin{aligned} b &= \mathbb{E}[R_{t+1} X_t] \in \mathbb{R} \\ A &= \mathbb{E}[X_t(X_t - \gamma X_{t+1})^T] \in \mathbb{R}^d \times \mathbb{R}^d \end{aligned} \tag{3.10}$$

Assim a esperança para o próximo vetor W pode ser escrita como:

$$\mathbb{E}[W_{t+1}|W_t] = W_t + \alpha(b - AW_t) \tag{3.11}$$

Dessa forma, para o sistema convergir, o vetor de pesos deve convergir para W_{TD}, seguindo a equação:

$$\begin{aligned} b - AW_{TD} &= 0 \\ b &= AW_{TD} \\ W_{TD} &= A^{-1} b \end{aligned} \tag{3.12}$$

Esse ponto é conhecido como *TD fixed Point*. Mais detalhes, como a demonstração que a inversa A^{-1} existe, podem ser encontrados em [2].

Outra questão importante de se notar é que, os métodos tabulares, estudados anteriormente, são casos particulares dos Aproximadores Lineares, onde o vetor de features $X(s)$ representa qual é o estado atual, ou seja, é um vetor de dimensão igual ao número de estados e aceita apenas valores binários, onde o estado atual possui valor 1 e todos os outros 0, já o vetor W fornece o valor individual de cada estado

[4]. Ou seja:

$$
\begin{aligned}
X(s_t) &= (0, 0, 0, ..., 1(s = s_t), ...0) \\
W_t &= (V^\pi(s1_t), ..., V^\pi(|S|_t)) \\
\hat{V}(s, W) &= W^T X(s)
\end{aligned}
\tag{3.13}
$$

3.3 Aproximadores não Lineares - Redes Neurais Artificiais (RNAs)

As Redes Neurais Artificiais RNAs são os principais Aproximadores para este trabalho, nesta seção apresentaremos um breve resumo do seu funcionamento e vamos começar a abordar um dos principais desafios do Aprendizado por Reforço que é lidar com problemas de convergência quando algumas características importantes estão presentes, e as RNAs é uma delas.

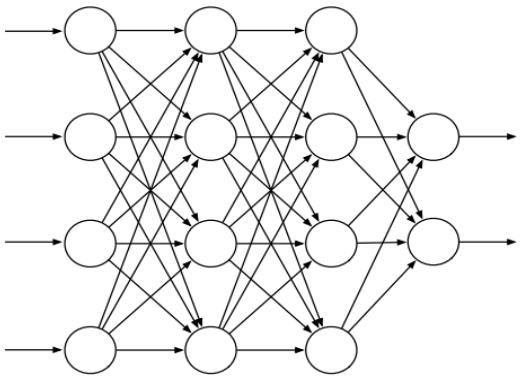

Figura 3.3: Representação de uma RNA. [2]

As RNAs procuram simular o funcionamento do sistema nervoso humano, a **unidade** básica de uma Rede Neural representa o funcionamento de um neurônio, que se conecta a outras unidades, a cada ligação um vetor de pesos W está associada, ao receber um sinal de entrada X cada unidade realiza uma soma com pesos WX e aplica uma função não linear denominada **função de ativação**, inúmeras funções podem ser utilizadas como a sigmóide, a arco-tangente e a **ReLU**, que será a Função de Ativação utilizada neste trabalho. Por fim, cada unidade passa o valor obtido para a próxima camada de unidades. Existem três tipos de camadas, a de entrada, as intermediárias (*hidden layer*) e a de saídas, conforme as figuras abaixo:

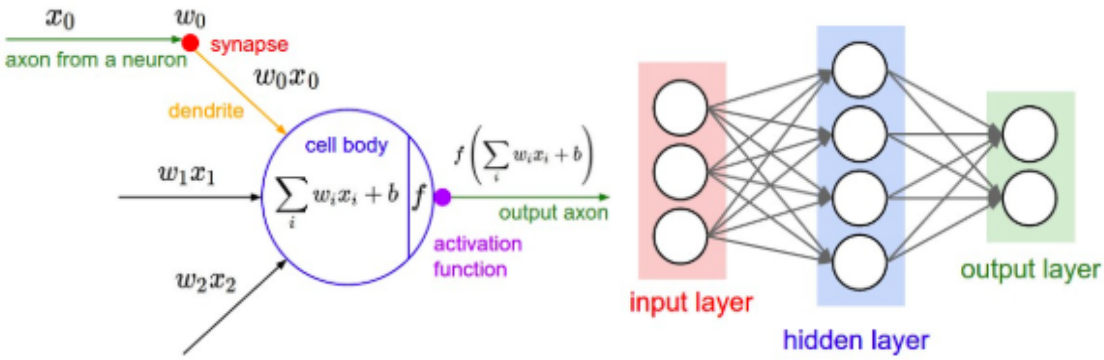

Figura 3.4: Representação de uma unidade e das camadas de uma RNA. [5]

Podemos afirmar que uma RNA com ao menos uma camada intermediária e com unidades suficientes podem aproximar qualquer função contínua, a única condição necessária é a não linearidade nas funções de ativação. Além disso, as RNAs não dependem de *features* criadas manualmente. Em linhas gerais o processo de aprendizado é realizado através do SGD, onde cada peso de W é ajustado na direção que maximiza o desempenho da Rede. Podemos utilizar o erro obtido ao aplicar o TD(0) e ajustar os pesos da RNA por meio do algoritmo de *backpropagation*. Além disso, alguns obstáculos podem ser encontrados na implementação desse aproximador, como o **overfitting**, que é um problema clássico na literatura e, por isso, há inúmeros métodos de **regularização** capazes de resolvê-lo, também existem algumas variações de RNAs como as CNNs (Convolutional Neural Network) e as RNNs(Recurrent Neural Network) [2].

RNA é um tema bastante amplo e com bastante relevância para o Aprendizado de Máquina, no entanto, para este trabalho será tratado apenas como uma ferramenta para aproximarmos a action value function, um estudo detalhado pode ser encontrado em [9], [5] e [10].

Por outro lado, ao introduzir o conceito de Aproximadores Não Lineares, estamos aptos a discutir um tema central para o Aprendizado por Reforço, que é a convergência dos seus métodos. Como já dissemos anteriormente o tipo de Aproximador utilizado por um método de aprendizado interfere diretamente na sua convergência. Na realidade há três características fundamentais de um método que podem prejudicar a convergência da Política Ótima, e elas são *Aproximadores Não Lineares, Bootstrapping e métodos Off-Policy*. Frequentemente são referenciadas como a *The Deadly Triad* (Tríade Mortal) [2]. A figura abaixo mostra uma relação de convergência de alguns métodos ao utilizarem as técnicas estudadas até agora.

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

Figura 3.5: Tabela de Convergência [4].

Cada uma dessas três características é fundamental para o bom desempenho de um método, que resolva problemas reais. As RNAs possuem um poder de generalização muito maior que os Aproximadores Lineares, o bootstrapping permite um estimador muito mais ágil e computacionalmente muito mais eficaz, já que não é necessário esperar todo o episódio, que provavelmente será bastante grande. O Q-Learning é a uma das principais inovações do Aprendizado por Reforço nos últimos anos e possibilita que uma política ótima seja encontrada independemente de qual política seja seguida.

O grande problema está no fato de que se essas três características estiverem presentes a convergência da Política Ótima não é garantida, como já vimos nas seções anteriores, quando até duas delas estão presentes ainda é possível contornar e encontrar um ponto de convergência [2]. No entanto, o algoritmo mais poderoso que podemos estudar, seguindo a linha deste trabalho, necessariamente possui essas propriedades.

Felizmente esse algoritmo já existe e foi amplamente divulgado, pelo grupo de pesquisa *Deep Mind*, em 2015, ao publicarem o Artigo Científico *Human-level Control Through Deep Reinforcement Learning* [1], que foi o principal estímulo para o desenvolvimento deste trabalho. No capítulo de Metodologia, iremos discutir detalhadamente as ideias propostas nesse artigo e, principalmente como foi possível contornar tais dificuldades.

3.4 Controle On-Policy com Aproximadores

Nesta seção retomamos o problema de Controle On-Policy, onde avaliamos e aprimoramos a mesma política que define o comportamento do agente, mas desse vez buscaremos soluções através de Aproximadores para a Q-Function $\hat{Q}(s, a, W) \approx Q^*(s, a)$ e com bootstrapping, ou seja, nossa abordagem será realizada através do TD(0).

Anteriormente escrevemos uma atualização como $S_t \rightarrow U_t$, já que buscávamos uma aproximação para a value-function, podemos estender essa ideia para o pro-

blema de Controle On-Policy, dessa forma, obtemos a atualização $S_t, A_t \rightarrow U_t$ e a função alvo passa a ser a Q-Function. Portanto a atualização do SGD, seguindo a equação 4.1 para a Q-Function, em uma abordagem equivalente ao algoritmo SARSA discutido anteriormente é:

$$W_{t+1} = W_t + \alpha[R_{t+1} + \gamma\hat{Q}(s_{t+1}, a_{t+1}, W_{t+1}) - \hat{Q}(s_t, a_t, W_t)]\nabla\hat{Q}(s_t, a_t, W_t) \quad (3.14)$$

Esse tipo de atualização converge com Aproximadores Lineares da mesma forma que o *Semi Gradiente TD(0)*, conforme foi mostrado através da equação 3.9. Com isso, avaliando e aprimorando a nossa política seguindo a lógica ε- gulosa, que com probabilidade $1 - \varepsilon$ realiza a ação gulosa $A^* = \arg\max_a \hat{Q}(s_t, a_t, W_t)$, obtemos o seguinte algoritmo:

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

Figura 3.6: SARSA com Aproximadores [2].

Ao longo deste Capítulo apresentamos a proposta de Aproximadores de Funções, que introduzem a ideia de utilizar um Regressor para a Q-Function, a fim de ganhar em generalização, dessa forma, as experiências do Agente funcionam como um Conjunto de Treinamento. Apresentamos uma métrica para o erro de um Aproximador, o Erro Quadrático Médio, e um método de Ótimização que minimiza esse erro, o SGD. Além de apresentar um Aproximador Linear e um Não Linear, e mostrar que para algumas situações específicas a convergência de $Q^*(s, a)$ é garantida. No próximo Capítulo iremos discutir sobre o problema de Convergência de Política Ótima ao combinar os três elementos: Bootstrapping, Q-Learning e o Redes Neurais Artificiais.

Capítulo 4

Metodologia

Neste capítulo vamos investigar como o *Deep Mind* conseguiu, através do Aprendizado por Reforço, vencer jogadores humanos em jogos de Atari. Discutiremos quais técnicas foram utilizadas para representar os estados somente por pixels. Como foi possível que a mesma arquitetura obtenha resultados tão bons, melhores que quaisquer outras técnicas anteriores, independente do jogo. Neste trabalho, inclusive, mostraremos os resultados obtidos ao aplicar essa mesma estrutura no jogo **Flappy Bird**, que não é um dos jogos em que o algoritmo foi testado no Artigo Científico [1]. Além disso, vamos verificar a questão fundamental, de que forma eles conseguiram contornar os problemas de convergência de Política Ótima.

Para resolver esse problema eles introduziram o conceito de **Deep Q-Network (DQN)**, que é capaz de combinar os métodos de Aprendizado Por Reforço com RNAs profundas, que permitem representações de dados sem a necessidade de pré-processamento de entrada. Ou seja, a ideia consiste em receber apenas os pixels da tela, que são uma representação do estado, e retornar os valores possíveis da Q-Function, assim podemos tomar a ação que retorna o maior valor. É importante notar que é uma abordagem diferente da que estamos acostumados, já que usualmente tratamos as ações possíveis de um estado como uma entrada para a Q-Function, isso deve ser levado em consideração, porque ao utilizarmos essa estratégia podemos encontrar a ação que maximiza a Q-Function em um única passagem pela RNA, enquanto pela abordagem tradicional o número de passagens necessárias seria igual ao número de ações disponíveis para tal estado. Conforme a figura a seguir.

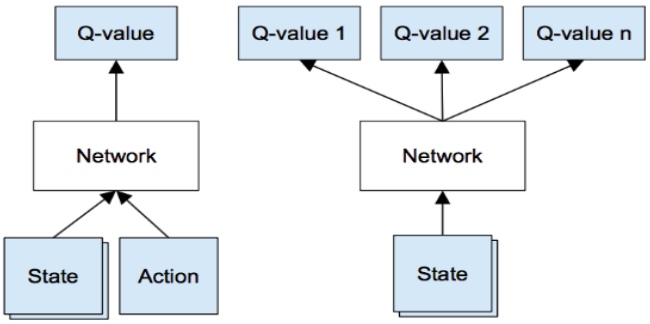

Figura 4.1: Aproximadores de Q-Function [6].

A imagem acima representa duas formas de se utilizar os Aproximadores de Função, neste trabalho, assim como no Artigo [1], utilizaremos a segunda abordagem, representada pela imagem da direita, que toma como entrada apenas o estado atual e retorna o valor encontrado da Q-Function para cada uma das n ações permitidas pelo estado atual s. Dessa forma, se buscamos a maximazação ds Q-Function basta tomar o maior valor, ou ainda, se buscamos o valor da Q-Function para uma ação específica teremos apenas que tomar o valor retornado pela ação correspondente.

Além disso, ao utilizar uma representação através dos pixels da tela o poder de generalização aumenta muito, já que todos os jogos terão seus estados representados da mesma forma. Podemos analisar uma outra abordagem baseada em features produzidas manualmente em outros trabalhos, como no Artigo [11], que também realiza testes em jogos do Atari, e verifica qual é a melhor forma de selecionar as características mais representativas de cada jogo, ou seja, o tipo de abordagem proposta pelo Artigo [1] e adotada neste trabalho já resolve este grande problema.

Apesar de não ser necessário realizar nenhum tipo de pré-processamento algumas transformações são feitas nas imagens antes delas serem inseridas na RNA, isso foi feito tanto para diminuir o tamanho da entrada, já que as imagens terão seu tamanho reduzido e as cores serão substituídas, como para ser possível representar a velocidade e a direção, que são elementos de diversos jogos e que não são determinados por apenas uma imagem. Para tanto, cada imagem foi redimensionada para 84 x 84, alterada para escala de cinza, com 256 níveis, e as quatro últimas imagens são utilizadas para dar a noção de velocidade e direção. Com isso, obtemos $256^{84\times84\times4} \approx 10^{67970}$ estados possíveis [6]. Os valores destes parâmetros foram determinados experimentalmente no Artigo [1] e definem a função de pré-processamento de imagens, que no Artigo é denotada pelo símbolo ϕ. Neste trabalho, para simplificar questões de nomenclatura assumiremos que um estado s é representado pelas transformações de pré-processamento descritas acima, dessa forma, não representaremos a função ϕ como em [1] no pseudocódigo do algoritmo.

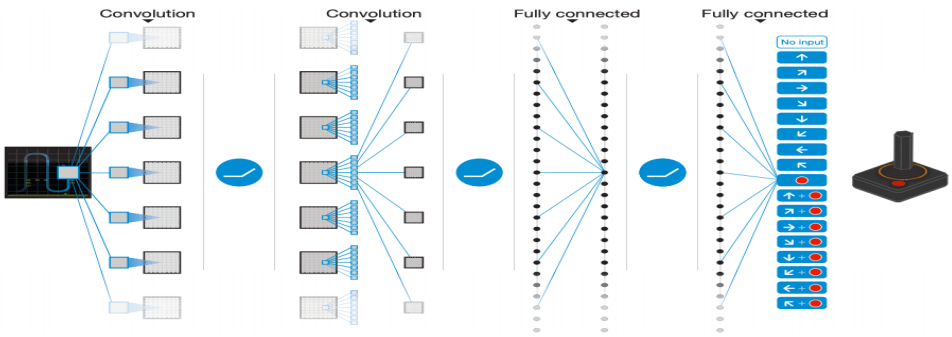

Figura 4.2: Arquitetura da RNA [1].

A arquitetura dessa RNA segue um formato clássico de Redes Convolucionais (CNNs), com três camadas de convolução, duas camadas inteiramente conectadas e a saída é o valor da Q-Function para cada uma ação das 18 ações, que são possíveis no Atari, para um dado estado s. As figuras a seguir mostram o formato da arquitetura e as características de cada camada.

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

Figura 4.3: Parâmetros de cada camada [6].

Apesar da proposta elaborada no Artigo [1] mostrar uma ótima alternativa ao problema de representação de estados e seleção de features, o principal mérito deste Artigo está maneira como contornam o problema de Convergência de Política Ótima, que será discutida a partir de agora, para uma discussão mais aprofundada, já que trata-se de um tema bastante complexo, a palestra [12] e a seção 'The Deadly Triad' do livro [2], ambos do Richard Sutton, são obrigatórios. Há algumas causas para tal comportamento, o seu principal responsável é a correlação entre uma sequência de amostras subsequentes, o fato de pequenas alterações na Q-Function poderem provocar mudanças consideráveis na política e a correlação entre os Actions-Values $\hat{Q}(s, a, W)$ e a função alvo $R_t + \gamma \max_{a'} \hat{Q}(s', a', W)$, que será bem explorada na seção de Atualizações Assíncronas. Para contornar tais problemas duas ideias principais foram utilizadas, a primeira conhecida como **Experience Replay** e uma atualização dessincronizada dos pesos da RNA para a estimativa $\hat{Q}(s, a, W)$ e a função alvo $U_t \rightarrow R_t + \gamma \max_{a'} \hat{Q}(s', a', W)$.

4.1 Experience Replay

É uma técnica baseada num processo biológico, onde as experiências ($e_t = s_t, a_t, r_t, s_{t+1}$) são armazenadas em uma memória $D = (e_1, e_2, ..., e_n)$, dessa forma, ao realizar o treinamento, através de *mini-batches*, podemos coletar exemplos aleatórios a partir de D, logo a correlação entre amostras subsequentes é removida e, portanto, a variância entre cada atualização da RNA, que busca otimizar a target policy, diminui, porque ao tomar as amostras aleatoriamente as experiências contidas em D funcionam como um conjunto de treinamento independente [1]. Além disso, em contraste com o treinamento convencional do SGD, onde cada amostra é utilizada uma única vez, cada exemplo e pode ser usada diversas vezes, enquanto estiver contida na memória D, o que representa um ganho de eficiência de dados. De maneira geral podemos afirmar que através dessa técnica a behaviour policy terá mudanças muito mais suaves, já que vários cenários anteriores serão considerados, por isso, a aproximação da Q-Function sofrerá menos oscilações e a chance de divergência diminuirá drasticamente [1].

No entanto, a Experience Replay também possui algumas limitações, ela é estruturada em formato de fila e N amostras e são permitidas, dessa forma, quando esse limite é superado as experiências mais antigas são substituídas pelas experiências mais recentes através de operações POP. Porém, táticas mais sofisticadas podem ser aplicadas, uma boa ideia seria aumentar a probabilidade de manter as amostras que contém mais informação, ou seja, aquelas em que o agente pode aprender mais, em um jogo, por exemplo, manter uma experiência em que o agente encontra um desafio que ele ainda não está acostumado seria melhor do que manter outra experiência que já foi encontrada diversas vezes e o agente já se especializou em como vencer. Essa mesma abordagem poderia ser utilizada na hora de selecioná-las para o mini-batch de treinamento, o qual deveria priorizar esses exemplos de maior valor. Podemos encontrar alguns exemplos onde tal estratégia foi aplicada como em [13].

4.2 Atualizações Dessincronizadas dos Pesos da RNA

Outra proposta fundamental para a estabilidade da DQN foi a utilização de diferentes pesos W_t para o cálculo da estimativa de $\hat{Q}(s, a, W)$ e a função alvo $U_t \rightarrow R_t + \gamma \max_{a'} \hat{Q}(s', a', W^-)$, lembrando que a recompensa imediata R_t é fornecida ao agente pelo ambiente ao tomar a ação a no estado s, ou seja, é um valor exato dado pelo ambiente, logo, não será estimado. Podemos dizer, em outras palavras, que os valores de W para o cálculo da função alvo só é atualizado a cada C atualizações de $\hat{Q}(s, a, W)$. Ao utilizar parâmetros mais antigos para o cálculo

da função alvo, a atualização das estimativas $\hat{Q}(s, a, W)$ não interferem mais em $\hat{Q}(s', a', W^-)$, já que agora os pesos são independentes. Portanto as oscilações também são bastante reduzidas, por isso, a chance de divergência também diminui [1]. Com isso, a atualização de W segue a equação abaixo e W^- realiza a cópia de W a cada C atualizações.

$$W_{t+1} = W_t + \alpha[R_t + \gamma \max_{a'} \hat{Q}(s', a', W_t^-) - \hat{Q}(s, a, W_t)]\nabla\hat{Q}(s, a, W_t) \tag{4.1}$$

Por fim, os valores das recompensas foram alterados para apenas três possibilidades, menos um para recompensas negativas, mais um para as positivas e as recompensas de valor zero tiveram seu valor mantido. Com isso, o erro também fica delimitado ao intervalo menos um e um, o que também aumenta a estabilidade do algoritmo [1]. A seguir apresentamos a versão do pseudocódigo do algoritmo Deep Q-Network usado neste trabalho.

4.3 Algoritmo Deep Q-Network

Algorithm 1: Deep Q-Network

```
Inicializar memória Experience Replay D com capacidade N
Inicializar Q(s, a, W) e Q̂(s, a, W⁻) com W = W⁻
Observar estado inicial s_0
for Passo = 1 até M do
    Com probabilidade ε selecionar ação aleatória a
    Senão selecionar a = max_a' Q(s, a', W)
    Executar ação a e observar recompensa r e novo estado s'
    Armazenar experiência e = (s, a, r, s') em D
    Sortear um minibatch aleatório de experiências e da memória D
    for cada experinecia e_i = (s_i, a_i, r_i, s'_i) do minibacth do
        if s'_i é um estado final then
            y_i = r_i
        else
            y_i = r_i + γ max_a'_i Q̂(s'_i, a'_i, W⁻)
        end
        Treinar rede Q através do erro (y_i − Q(s_i, a_i, W))²
    end
    A cada C passos atualizar W⁻ = W
    s = s'
end
```

O algoritmo começa com a inicialização da memória Experience Replay D e dos

pesos W e W^-, em seguida recebemos o primeiro estado s_0 do ambiente. Então entramos num laço, que deve ser repetido M vezes. Nele, uma ação é selecionada seguindo a política ε-gulosa. Dessa forma, ao executar a ação a uma nova experiência $e = (s, a, r, s')$ será gerada, em seguida ela deve ser armazenada na memória D. Um conjunto de experiências é sorteada aleatoriamente para treinar a RNA de pesos W, após esse processo de minização do EQM entre y_i e a estimativa da Q-Function, o vetor de pesos W^- pode ser atualizado dependendo dos C passos e, por fim, passamos para o próximo estado s' e o laço se repete.

Este algoritmo pode ser visto como uma extensão do Q-learning, expresso pela figura 2.17. Onde utilizamos duas políticas uma a ser seguida e outra a ser otimizada, para a primeira utilizaremos a mesma estratégia ε-gulosa e a segunda sempre buscará uma ação que maximize as recompensas a longo prazo.

No entanto, agora utilizamos RNAs para estimar o valor da Q-function, que são treinadas através da minimização do EMQ, entre o valor mais atual y_i, que pode valer r_i caso s'_i seja um estado terminal ou a soma da recompensa imediata r_i com a estimativa de $\hat{Q}(s'_i, a'_i, W^-)$, que é a rede com pesos mais antigos utilizada para diminuir a correlação entre as estimativas. Para realizar essa otimização, através do SGD, são utilizadas as experiências armazenadas na memória Experience Replay D, que são sorteadas aleatoriamente, por isso, o índice i que indica que todas as experiências do minibatch serão utilizadas. Dessa forma, a correlação entre experiências subsequentes é removida [1]. Também é importante lembrar que as experiências mais antigas são substituídas por valores mais novos ao ultrapassar a capacidade N da memória.

Além disso, a cada C passos o valor de W^- é atualizado com o valor de W, outra questão importante é que neste algoritmo já consideramos as transformações de pré-processamento para representar um estado, ou seja, um estado s é composto por quatro quadros seguidos e redimensionados como já descrito. Outra questão que pode gerar dúvidas é como o valor de r e qual o próximo estado s' são determinados, esses valores são retornados pelo ambiente, no nosso caso são fornecidos pelo emulador, que retorna uma pontuação no jogo para uma dada ação assim como os próximos estágios do jogo, que são utilizados para capturar as quatro imagens subsequentes e definir s'.

4.4 Flappy Bird

Neste jogo controlamos um pássaro, que representa o nosso agente, e o nosso objetivo é obter o maior número possível de pontos. Para pontuar devemos atravessar os obstáculos formados por pares de canos, que surgem aleatoriamente na tela e de forma independente para cada partida. Não existe uma pontuação máxima, ou seja,

caso o jogador seja bom o suficiente o jogo continua indefinidamente, se o pássaro tocar qualquer obstáculo ou o solo a pontuação é zerada e o e uma nova partida independente das anteriores é iniciada.

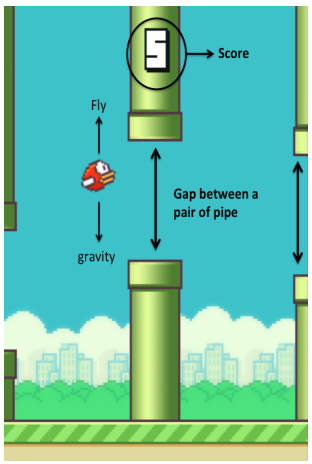

Figura 4.4: Dinâmica Flappy Bird [7].

Utilizaremos esse ambiente para realizar testes tanto para o algoritmo DQN quanto para outras variações, que buscam verificar a eficácia das técnicas proposta pelo algoritmo, como a arquitetura da RNA, a memória Experience Replay e as Atualizações Assíncronas. Optamos por utilizar este ambiente porque além de ser bem conhecido pelo público comum também já foi alvo de trabalhos similares como [14] [7] [15].

Todo algoritmo foi implementado em Python. Utilizamos a biblioteca **Pygame** [16] para emular o ambiente, que possui o funcionamento conforme o esquema a seguir:

- 1 ponto de recompensa para cada tubo ultrapassado;
- Recompensa de -1 se o agente perder a partida, que representa um estado terminal;
- Duas ações são possíveis: voar ou não voar;
- Não existe pontuação máxima;
- Os obstáculos aparecem de forma aleatória;
- As partidas são indepentes entre si;
- O jogo opera em trinta quadros por segundo, o que significa que existem 7,5 estados por segundo, já que são utilizados 4 quadros para representar um estado. Além disso, para cada estado há duas ações possíveis, voar ou não voar, portanto, em um segundo há a possibilidade de 15 ações.

4.5 Implementação

Para implementar o algoritmo tivemos como referência alguns projetos similares [17] [18] [19], que apesar de não utilizarem o Flappy Bird como ambiente implementam grande parte dos algoritmos discutidos ao deste trabalho, inclusive o DQN.

O primeiro [17] é bastante semelhante a este, no sentindo de utilizar o [1] como principal referência e o objetivo dele é a implementação do DQN, assim como no Artigo ele utiliza jogos do Atari para teste. Também é construído em Python, mas utiliza bibliotecas diferentes das utilizadas por este trabalho, com exceção a biblioteca Keras.

Já o segundo [18] é um projeto oficial do Keras, bastante extenso, que implementa diversos algoritmos em vários ambientes, além de disponibilizá-los para novos testes, utiliza outro conjunto de bibliotecas e como destaque possível uma boa documentação e fácil instalação através do PIP.

Enquanto nossa terceira referência [19] para implementação possui uma característica muito mais didática ao resolver os problemas propostos pelo livro [2] e pelo curso [4]. É indispensável para quem busca entender mais a fundo os algoritmos e implementá-los. O principal destaque entre as bibliotecas foi a escolha da ferramenta TensorFlow para implementação das Redes Neurais.

Como última referência de destaque, para implementação, utilizamos o projeto [20], é importante salientar que parte do código deste projeto foi reutilizado, aproveitamos a construção da arquitetura da RNA do agente DQN, em Keras, que foi implementada de forma simples e eficaz, assim como a estrutura da memória Experience Replay, feita a partir da collection Deque do Python, traremos mais detalhes da implementação na próxima seção de módulos.

Como principais diferenças do nosso trabalho aos demais, especialmente ao trabalho [20], que foi reutilizado, é que construímos outras duas arquiteturas de Redes Neurais, implementamos a técnica de Atualizações Dessincronizadas discutida nas seções anteriores, utilizamos uma outra estratégia para avaliar um dado agente, que é definido pelo algoritmo que é seguido, além de introduzirmos a ideia de módulos, que permite o treino e a avaliação de diferentes agentes ao ativar ou desativar determinada técnica do algoritmo. Mais detalhes serão apresentados nas seções de Módulos, Agentes e Treinamento. Todas as referências e o código utilizado estão dísponíveis em `https://github.com/LyangHiga/Undergraduate-Final-Project/tree/master/codes/v3`.

4.5.1 Módulos

A fim de simplificar a explicação de cada agente utilizaremos o conceito de módulos, que representam componentes indispensáveis na construção de cada agente. Alguns

módulos possuem mais de uma implementação disponível e a escolha de uma ou outra afeta diretamente o resultado obtido pelo agente. Nesta seção apresentaremos cada um dos módulos, suas respectivas implementações e o que é esperado de cada uma delas.

É importante deixar claro que um agente é definido a partir das técnicas que foram utilizadas no algoritmo durante o treinamento, ou seja, um dado agente pode possuir ou não a memória Experience Replay, pode utilizar um Aproximador Linear ou uma Rede Neural e assim por diante. Todos os Agentes implementados serão apresentados na próxima seção.

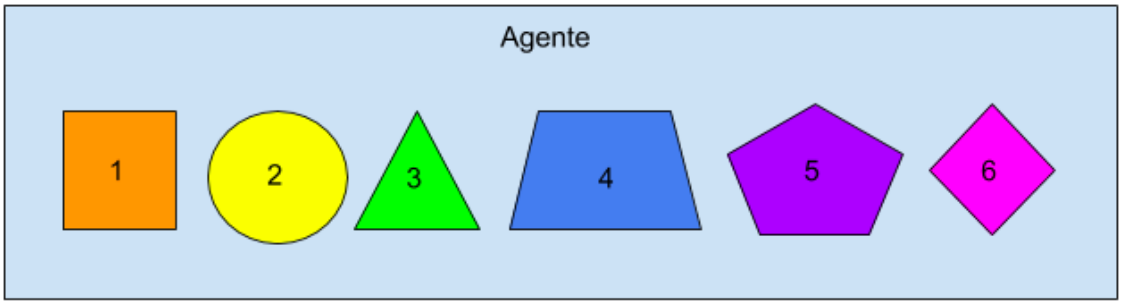

Figura 4.5: Composição de um agente. Cada figura geométrica representa um módulo que um dado agente pode possuir.

1: Captura de Frames e Representações de Estados.

O primeiro módulo é responsável pela representação dos estados estados a partir das imagens fornecidas pelo emulador, que também nos retorna a recompensa imediata de cada estado, ou seja, ao se tomar uma ação o emulador nos fornece o seu retorno instantâneo R_t, e se esse estado é terminal. Assim como no Artigo [1] as imagens passaram por um pré-processamento, ou seja, foram convertidas para escala de cinza com 256 níveis e redimensionadas para 84 X 84, também utilizamos as quatro últimas imagens para ter informações de direção vertical e velocidade. Para isso foi utilizada a biblioteca **scikit-image** que permite essa manipulação de forma bastante simples. Este módulo possui uma única implementação.

2: Modelo de Aprendizado e Arquitetura RNA

Este módulo conta com três implementações diferentes de arquitetura de rede, além de funções lineares de ativação para o agente linear, que será discutido na próxima seção.

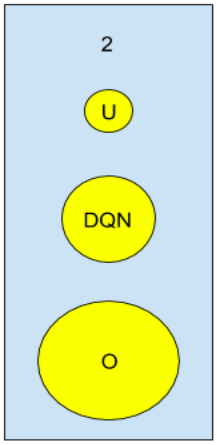

Figura 4.6: Representação das três arquiteturas implementadas. A primeira 'U' é uma Rede com menos camadas, a segunda é a arquitetura DQN, e a terceira 'O' possui mais camadas que a DQN, ou seja, o tamanho de cada figura está relacionado com o número de camadas da arquitetura.

A primeira implementação de arquitetura de rede procura verificar se há efeitos de *underfitting*, ou seja, utilizamos uma rede consideravelmente menor a fim de observar uma redução no poder de generalização dos agentes que utilizam tal arquitetura, que é composta por uma única camada Convolucional, uma camada inteiramente conectada e uma camada de saída.

A segunda implementação é a própria arquitetura DQN já discutida anteriormente, com o objetivo de verificar se ela realmente possui o melhor poder de generalização quando comparada com as outras alternativas implementadas.

A terceira implementação busca verificar os efeitos de *overfitting*, para isso, utilizamos uma rede maior, composta por cinco camadas de convolução, quatro completamente conectadas e a saída. Uma arquitetura consideravelmente maior , pode acabar se especializar demais em amostras obtidas durante o treinamento e, portanto ter dificuldades em expandir para a etapa de teste.

Além disso, é este módulo responsável por determinar qual função de ativação será utilizada por cada camada da RNA, dessa forma, mesmo utilizando uma Rede Neural é possível implementar um Aproximador Linear. Foi implementado a partir da biblioteca Keras [21] e o framework TensorFlow.

3: Experience Replay

O terceiro módulo é responsável pela memória Experience Replay, na nossa implementação, o parâmetro N, que representao seu limite, vale 50 mil imagens pré-processadas, conforme proposto em [1], no entanto, essa memória não cresce ao longo do treinamento como no Artigo, já que utilizamos uma quantidade menor de passos, mais detalhes serão encontrados na seção de Treinamento.

Foi construído seguindo a lógica de uma Fila, onde as experiências, $e_t = (s_t, a_t, r_t, s_{t+1})$, mais antigas vão para o fundo até sofrerem uma operação POP. Foi implementada através da collection Deque (*"double-ended queue"*), para mais detalhes [22], do Python já que cada operação de append e pop pode ser realizada em $O(1)$.

Além disso, para facilitar os testes utilizamos uma abordagem simples, ou ele está ativado e o agente possui o módulo, ou está desativado e o agente não conta com o componente. A ideia é verificar como o módulo influencia o desempenho do agente através da comparação da sua presença ou não. Conforme veremos no capítulo de resultados.

4: Atualizações Assíncronas

Para implementar este módulo utilizamos a mesma abordagem do anterior, ou seja, uma flag indica se o componente está presente ou não, e verificamos sua efetividade através da comparação de agentes.

Assim como foi proposto pelo artigo os parâmetros da função alvo são atualizados a cada dez mil passos. Ou seja, as atualizações são feitas exatamente como definido pela equação 4.1 para C igual a dez mil.

5: Política Ótima

Uma parte importante para o projeto é diferenciar a etapa de treinamento, onde as ações aleatórias são desejadas e controladas pelo parâmetro ε, conforme já discutimos anteriormente, da fase de teste, que é quando queremos avaliar o conjunto de melhores ações possíveis segundo o nosso modelo, por isso, as ações aleatórias devem ser proibidas nessa etapa.

Apesar de parecer uma observação trivial grande parte dos projetos similares não levou esses detalhes em consideração e seus agentes obtiveram resultados consideravelmente piores, por isso, utilizamos esse módulo em todos os agentes que implementamos.

6: Clipping de Recompensas

Utilizamos o próprio mecanismo de pontuação do jogo, com isso, não foi necessário implementar um política de clipping de recompensas, já que cada recompensa positiva já está definida, vale um ponto positivo e cada recompensa negativa retorna menos um.

4.5.2 Agentes

Testamos dezesseis agentes, sendo doze deles combinações entre os módulos, e outros quatro que não seguem esse mesmo padrão.

Linear

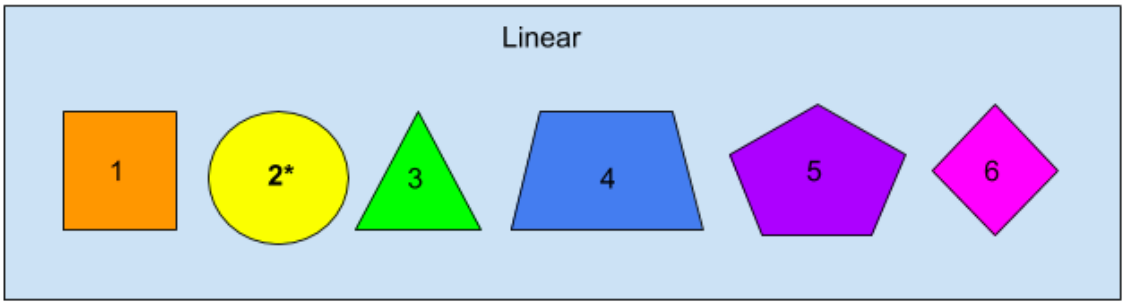

Figura 4.7: Módulos do Agente Linear.

O agente Linear conta com todos os módulos, no entanto, assim como no Artigo original substituímos as funções ReLu de ativação das RNAs por funções Lineares, para informações sobre as funções de ativação [23],mantemos a mesma arquitetura de rede DQN. Dessa forma, não foi necessário construir manualmente as nossas próprias features, como realizado em [11], assim obtemos um Aproximador Linear, já que a combinação de funções lineares resulta em uma função linear.

Aleatório

Também foi implementado um agente que sempre toma ações aleatórias, com a mesma probabilidade para cada ação. Sabemos que o resultado esperado é bastante ruim, mas assim como no Artigo vamos utilizá-lo a fim de comparações.

Reutilizado

Figura 4.8: Módulos do Agente Reutilizado.

Conforme já mencionado houve reutilização de código, por isso, acreditamos que uma parte importante do trabalho é mostrar o que já havia sido implementado e o que foi introduzido e como isso afeta os resultados. O agente construído no projeto [20] não conta com os módulos de Atualizações Assíncronas e de Política Ótima, como arquitetura de rede utiliza a estrutura DQN. É importante lembrar que a implementação desta RNA e a memória Experience Replay usados neste agente não foram feitas por mim.

Combinações de Módulos

A fim de verificar a efetividade de cada módulo realizamos diversos teste que combinam diferentes possibilidades. Todos esses agentes contam com os módulos 1, 5 e 6 exatamente como descritos na seção anterior. O módulo 2, onde a Rede Neural é implementada foi feita para todos esses agentes utilizando funções ReLu de ativação, o que variamos foi a arquitetura da rede. Já os módulos 3 e 4, Experience Replay e Atualizações Assíncronas, estão presentes em metade dos agentes. Em outras palavras, esse agentes variam entre uma das três arquiteturas possíveis (U,DQN,O), possuem ou não Experience Replay e Atualizações Assíncronas. Conforme a tabela a seguir, que contém o nome dado a cada agente, e os números se referem a cada módulo. Tais informações são importantes para a chamada do algoritmo.

2 / 3 e 4	**Sem 3 e 4**	**Sem 3 com 4**	**Com 3 Sem 4**	**Com 3 e 4**
Under	NAUWER	NAUWERAS	NAUER	NAUERAS
DQN	DQNWER	DQNWERAS	DQNER	DQN
OVER	NAWER	NAWERAS	NAER	NAERAS

Tabela 4.1: Nomenclatura dos Agentes que são construídos a partir da combinação de módulos.

Esses nomes são importante porque a partir deles a nossa implementação verifica quais componentes devem ser ativados, este controle é feito através de um Dicionário do Python, que utiliza como chave o nome do Agente e como valor possui uma flag para cada módulo que possui mais de uma implementação e indica qual delas deve ser utilizada.

O nome desses agentes segue a seguinte lógica: a primeira parte indica qual arquitetura RNA ele possui, 'NAU' para arquitetura U, que verifica underfitting, 'DQN' para arquitetura DQN e 'NA' para a arquitetura O. A segunda parte se refere a presença da memória Experience Replay, 'WER' indica que o módulo não está presente, enquanto 'ER' indica que o agente possui o módulo, e por último as sigla 'AS' quer dizer que é utilizada a técnica de Atualizações Dessincronizadas, caso esta sigla não esteja na parte final do Agente ele não conta com essa características.

4.6 Treinamento

Nesta seção apresentaremos os valores dos hiperparâmetros utilizados neste trabalho durante a fase de treino, além disso, faremos uma pequena comparação com os valores utilizados no ARtigo [1] e justificaremos as principais diferenças.

Neste trabalho cada agente foi treinado com M igual a 1 milhão de passos com o mesmo learning rate, α de 10^{-4}, uma taxa de depreciação γ de 0,99. Os agentes que utilizam o módulo de Experience Replay contam com uma memória D de 50 mil experiências, $e_t = s_t, a_t, r_t, s_{t+1}$, mais recentes, e usam um mini batch de 32 experiências aleatórias para o treinamento da RNA. além disso, os agentes que utilizam Atualizações Assíncronas igualam o valor dos pesos das RNAs a cada 10 mil atualizações.

Seguindo a dinâmica do jogo o nosso agente possui apenas duas ações possíveis voar ou não voar. Durante a fase de treinamento seguimos uma política ε - guloso, ou seja, com uma probabilidade ε realiza uma ação aleatória e com probabilidade $1 - \varepsilon$ toma a ação que maximiza a Q-Function. Assim como nos projetos [7] e [15] optamos por utilizar um ε_0 inicial de 0,1 que decai linearmente até ε_f igual a 0,0001. Enquanto o Artigo original utiliza o intervalo de 1 a 0,1.

Hiperparâmetro	**Valor neste Trabalho**	**Valor no Artigo [1]**
M	1 Milhão	50 Milhões
D_0	50 mil	50 mil
D_f	50 mil	1 milhão
C	10 mil	10 mil
γ	0,99	0,99
α	1×10^{-4}	$2,5 \times 10^{-4}$
ε_0	0,1	1
ε_f	1×10^{-4}	0,1
Tamanho Mini-Batch	32	32

Tabela 4.2: Comparativo entre os valores dos principais hiperparâmetros entre este trabalho e o Artigo [1].

Podemos justificar essa escolha a característica inerente ao jogo de ser extremamente sensível a ações aleatórias, ou seja, ao se tomar uma ação aleatória a chance do episódio se encerrar é bastante grande, por isso, ao se utilizar um ε alto as partidas iniciais provavelmente serão perdidas antes mesmo de se encontrar o primeiro obstáculo e dessa forma o aprendizado será mais demorado. Também há outros valores discrepantes como o número de passos M e o tamanho final D_f da memória Experience Replay, justificamos tais escolhas por questões computacionais,

apesar dos valores serem realmente menores, os efeitos no algoritmo DQN já serão percebidos, conforme veremos no próximo capítulo.

4.6.1 Exploration - Exploitation e o Efeito da Aleatoriedade

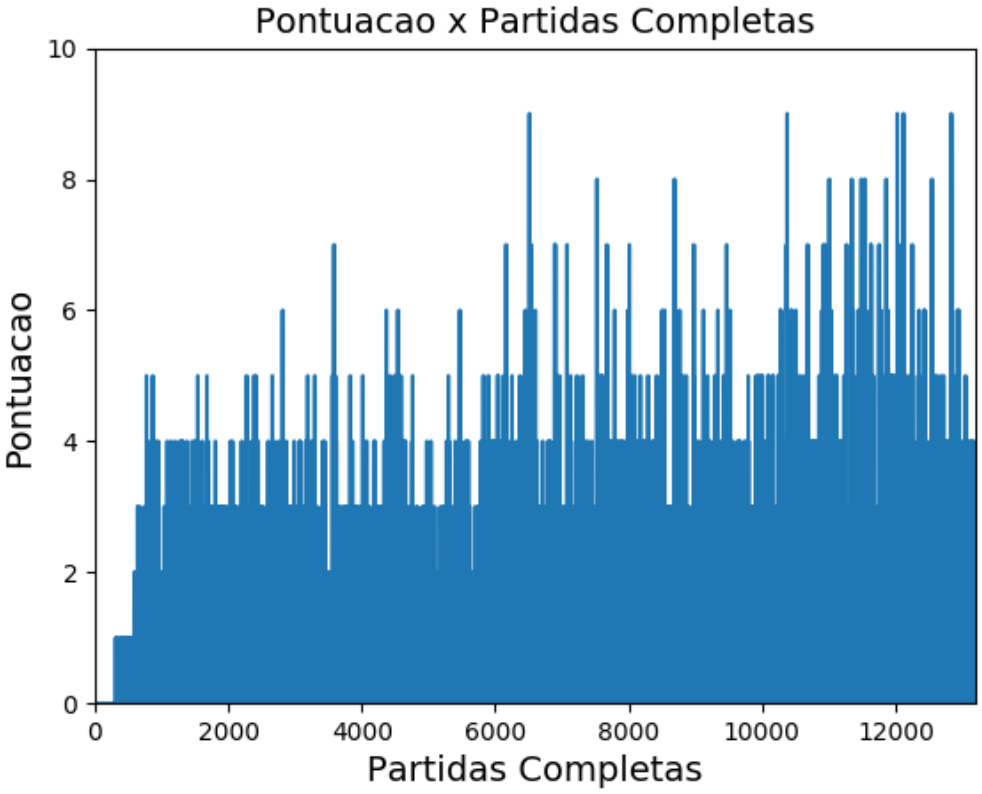

Figura 4.9: Pontuação x Episódios do Agente DQN na fase de Treino.

A figura acima mostra a pontuação obtida em cada partida completa durante a fase de treino do agente DQN. Através dela podemos verificar o efeito da aleatoriedade neste jogo. Como era esperado ao se tomar uma ação aleatória a chance do episódio se encerrar é grande, por isso, nos primeiros episódios uma quantidade considerável de pontuações nulas são obtidas e nenhuma partida alcança uma pontuação elevada. No entanto, conforme o número de episódios vai aumentado, o número de passos também aumenta, o valor de ε vai diminuindo e, por isso, o número de ações aleatórias também diminui e assim a pontuação começa a aumentar.

Enquanto a alguns ambientes podem se beneficiar de um comportamento bastante explorador, ou seja, que possui um ε próximo de 1, conforme podemos verificar na tabela 4.2, que mostra que no Artigo [1] foi usado um $\varepsilon_0 = 1$ e em boa parte dos jogos eles obtiveram excelentes resultados. Em contrapartida, o jogo Flappy Bird se beneficia de um comportamento explorador apenas nas primeiras partidas, e assim que o agente começar a ultrapassar os primeiros obstáculos ainda é interessante

que haja um comportamento aleatório, mas que seja menor. Dessa forma, o agente consegue encontrar os obstáculos e conforme os ultrapassa tem a oportunidade de aprender a escapar de barreiras de diferentes alturas. Por isso, optamos por um ε_0 consideravelmente menor com valor de 0,1.

Capítulo 5

Resultados

Após todos os processos descritos no capítulo anterior encontramos as seguintes médias de pontuação e desvios padrão, entre parênteses na segunda tabela, que foram calculados através de 200 partidas consecutivas independentes:

Agente	**Pontuação Média**	**Desvio Padrão**
Ações Aleatórias	0,0	0,0
Aproximadores Lineares	0,0	0,0
Humano	12,30	6,57
Reutilizado	53,15	22,80

Tabela 5.1: Desempenho dos diferentes Agentes no Flappy Bird

Essa primeira tabela mostra os resultados obtidos por quatro agentes, primeiro temos aquele que sempre toma ações aleatórias, que assim como o segundo, que utiliza Aproximadores Lineares, não conseguiram ultrapassar nenhum obstáculo em duzentas partidas. O terceiro agente, "Humano", obteve uma média de 12,3. Para avaliar esse agente, selecionei entre meus amigos o melhor jogador de Flappy Bird, que após um dia de treino, realizou dez partidas consecutivas utilizadas para obter este resultado. Por fim, avaliamos o agente Reutilizado, que é aquele que não foi desenvolvido por mim, ele obteve uma média de 53,15 após as duzentas partidas.

Esta tabela revela um resultado relativamente inesperado. Imaginávamos que o agente Linear obteria pontuações bem inferiores aos agentes que utilizam Aproximadores Não Lineares, mas observar que ele não conseguiu ultrapassar um único obstáculo sequer, igualando-se ao agente que age de forma completamente aleatória, foi uma surpresa e demonstra a importância dos Aproximadores Não Lineares.

O agente Reutilizado não obteve um resultado tão ruim, porém, a questão de ainda utilizar ações aleatórias com probabilidade ε durante a fase de teste compromete uma parte dos seus resultados, especialmente quando comparamos com os agentes que serão apresentados na próxima tabela, que sempre agem de forma ótima

durante a fase de teste.

2 / 3 e 4	**Sem 3 e 4**	**Sem 3 com 4**	**Com 3 Sem 4**	**Com 3 e 4**
Under	14,14 DP(4,75)	34,45 DP(9,18)	66,94 DP(18,73)	84,77 DP(23,82)
DQN	31,12 DP(8,78)	72,09 DP(21,62)	136,76 DP(38,94)	153,94 DP(41,69)
OVER	52,78 DP(16,67)	93,02 DP(27,50)	164,27 DP(43,71)	207,97 DP(52,23)

Tabela 5.2: Desempenho dos diferentes Agentes, organizados por módulos, no Flappy Bird

Já esta segunda tabela apresenta os resultados obtidos pela combinação das diferentes arquiteturas de RNA, a presença ou não das técnicas Experience Replay e a Atualização Assíncrona dos pesos da RNA. Outra questão importante para avaliar os resultados dessa tabela é que não existe uma pontuação máxima, ou seja, não há um obstáculo que seja insuperável, dessa forma, se um agente for bom o suficiente uma partida continua indefinidamente.

Diferente do esperado o agente DQN não foi o agente que obteve as melhores pontuações. Acreditava-se que a terceira arquitetura de rede iria proporcionar um efeito de overfitting e prejudicar os resultados de todos os seus agentes, no entanto, tal problema não ocorreu e, por isso, a arquitetura com mais camadas obteve os melhores resultados. Outra questão que pode ser levantada é se com um treino extenso esse tipo de arquitetura, com mais camadas, não poderia perceber quais ações devem ser tomadas para cada cenário do jogo e, assim vencer todos obstáculos sem realmente utilizar estimativas que maximizem a Q-Function. Se isso ocorresse os agentes que possuem esta arquitetura nunca perderiam e as partidas seguiriam indefinidamente. No entanto, é possível que para obstáculos que apareceram mais frequentemente, durante a fase do treino, estes agentes tenham aprendido exatamente como agir, mas com certeza isto não ocorreu para todos os cenários do jogo. Para avaliar melhor esta situação um novo estudo, mais profundo acerca destas questões de overfitting, seria necessário.

Já os agentes que utilizam uma arquitetura de RNA menor do que a DQN tiveram um resultado consideravelmente pior que aqueles agentes que utilizam outras arquiteturas de rede. Isso se deve ao efeito de Underfitting, que já era esperado. Mas mesmo assim ficou evidente a importância dos módulos 3 e 4, já que mesmo com uma arquitetura menor esses módulos elevaram muito a pontuação média.

Por fim, os agentes que utilizam a arquitetura DQN obteve bons resultados, especialmente ao compararmos com a arquitetura com menor camadas. Além disso, uma boa comparação pode ser feita entre o agente que possui Experience Replay, DQNER, com o agente reutilizado, que também conta com o módulo 3, a única diferença entre eles está no módulo 5, porém, há um grande reflexo na pontuação,

desse modo, percebemos a efetividade deste módulo.

Outro ponto importante a ser discutido é o Desvio Padrão. Inicialmente foram utilizadas poucas partidas para se obter a média, por isso, boa parte dos agentes ficavam com um Desvio Padrão acima dos 50 por cento, já que em ao menos uma partida era obtida uma pontuação muito alta ou muito baixa. Ao aumentarmos o número de experimentos para 200 partidas os casos em que um dado agente obteve uma pontuação muito alta ou muito baixa diminui percentualmente. Desta forma, o Desvio Padrão ficou em torno dos 30 por cento para cada agente. Ainda pode parecer um valor bastante alto, mas ao compararmos com alguns ambientes testados pelo Artigo [1] verificamos que esse é um valor aceitável.

O Desvio Padrão foi calculado da seguinte forma: $DP = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}}$. Onde n representa o número de partidas, neste caso duzentas, x é a pontuação de cada partida e $\overline{x}$ é a pontuação média após as duzentas partidas.

Não fizemos um comparativo explícito com os outros trabalhos citados, porque, como podemos verificar, cada um utiliza diferentes ferramentas, apesar de construir modelos bem parecidos, todos inspirados em [1]. Também não nos preocupamos em comparar outros métodos de Aprendizado além do Q-Learning, já que a principal intenção deste trabalho é verificar a convergência do modelo ao combinar bootstrapping, métodos Off-Policy e Aproximadores Não Lineares.

Os resultados mais expressivos se devem aos módulos 3 e 4, Experience Replay e Atualizações Assíncronas, conforme verificado na segunda tabela, ao acrescentá-los os resultados melhoram consideravelmente, e quanto mais camadas a rede possuía maiores foram os seus efeitos. Também é importante destacar que esses valores refletem o treino realizado conforme descrito pela tabela 4.2 e as características do jogo Flappy Bird, ou seja, para que novas conclusões possam ser obtidas seriam necessários outras formas de treinamento e de avaliação, inclusive utilizando outros ambientes, não necessariamente jogos virtuais.

Apesar destes experimentos não nos trazerem novas conclusões eles são suficientes para o objetivo deste trabalho, que é verificar a efetividade das técnicas propostas pelo Artigo [1]. Dessa forma, podemos concluir, que independentemente da arquitetura de RNA utilizada, tanto a memória Experience Replay quanto as Atualizações Assíncronas são eficazes e a utilização delas beneficia o agente.

Capítulo 6

Conclusão

Neste trabalho conseguimos cumprir os objetivos propostos. Como foi verificado o Artigo Científico [1] realmente nos apresenta uma boa solução para o problemas proposto, que foi discutido, aos poucos, no decorrer deste trabalho. Ou seja, conseguimos mostrar que, mesmo sem garantias formais de convergência, a metodologia proposta foi capaz de manter a estabilidade do modelo e, por isso, converge para uma política aproximadamente ótima e isso se reflete nos resultados obtidos, que, apesar de não trazerem novas conclusões confirmam a importância do Artigo.

Além disso, realizamos um estudo extenso, com uma boa base teórica, fomos capazes de implementar diversos agentes em um novo cenário e, não apenas replicar exatamente os ambientes descritos no Artigo, por fim, realizamos comparativos com os resultados obtidos.

Ao decorrer desse trabalho percebemos que havia ainda diversas oportunidades de pesquisa, que não foram abordados aqui. Dentre eles podemos destacar o bootstrapping, já que não discutimos outros métodos que utilizam bootstrapping além do TD(0), já que somente ele já é suficiente para um bom entendimento do Q-Learning. Podemos explorar mais os Aproximadores de Funções, especialmente os Não-Lineares, e verificar as consequências ao utilizar outros modelos.

Outra ideia interessante seria explorar outras formas de treinamento além das utilizadas neste trabalho, que são expressas pela tabela 4.2. Realizar mais testes de avaliação em outros ambientes, inclusive em ambientes que não sejam de jogos. Ou seja, acreditamos que ainda exista uma grande oportunidade em explorar mais os experimentos, tanto na questão de variar o treinamento, quanto nos agentes implementados e nos ambientes testados.

Também podemos nos preocupar, em trabalhos futuros, com questões mais específicas como a memória Experience Replay, que pode ser utilizada de formas mais inteligentes conforme podemos verificar nos Artigos [24] e [13]. Além disso, podemos estender essas técnicas para ambientes contínuos como em [25]. Ou ainda podemos buscar um aprimoramento no Q-Learning através de um processo de aprendizado

duplo, conhecido como *Double Q-Learning* como em [26].

Referências Bibliográficas

[1] MNIH, V., KAVUKCUOGLU, K., SILVER, D., et al. "Human-level control through deep reinforcement learning". . http://https://arxiv.org/abs/1511.05952, 2015. Acessado em Setembro de 2018.

[2] RICHARD S. SUTTON, A. G. B. *Reinforcement Learning: An Introduction*. 2 ed. Cambridge, MA, USA, MIT Press, 2018. http://incompleteideas.net/book/the-book-2nd.html.

[3] DAN KLEIN, PIETER ABBEEL. "UC Berkeley CS188 Intro to AI". . http://ai.berkeley.edu/home.html, 2014. Acessado em Setembro de 2018.

[4] SILVER, D. "COMPM050: UCL Course on Reinforcement Learning". . http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html, 2015. Acessado em Setembro de 2018.

[5] FEI-FEI LI ET AL. "CS231n: Convolutional Neural Networks for Visual Recognition". . http://cs231n.github.io/, Jan. 2018. Acessado em Janeiro de 2018.

[6] MATIISEN, T. "Demystifying Deep Reinforcement Learning". . https://www.intel.ai/demystifying-deep-reinforcement-learning/#gs.Z7G6yV9P, 2015.

[7] LOUIS-SAMUEL PILCER, ANTOINE HOORELBEKE, A. D. "Playing Flappy Bird with Deep Reinforcement Learning". . https://www.researchgate.net/publication/324066514_Playing_Flappy_Bird_with_Deep_Reinforcement_Learning, 2018. Acessado em Dezembro de 2018.

[8] EVSUKOFF, A. G. *ENSINANDO MÁQUINAS*.

[9] ANDREW NG. "Neural Networks and Deep Learning". . https://www.coursera.org/learn/neural-networks-deep-learning/home/welcome. Acessado em Agosto de 2018.

[10] GOODFELLOW, I., BENGIO, Y., COURVILLE, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[11] MARC G. BELLEMARE, YAVAR NADDAF, J. V. M. B. "The Arcade Learning Environment: An Evaluation Platform for General Agents". . https://arxiv.org/abs/1207.4708, 2013. Acessado em Janeiro de 2019.

[12] SUTTON, R. S. "Tutorial: Introduction to Reinforcement Learning with Function Approximation". . https://www.youtube.com/watch?v=ggqnxyjaKe4, 2016.

[13] ANDREW W. MOORE, C. G. A. "Prioritized Sweeping"Reinforcement Learning With Less Data and Less Time". . https://link.springer.com/article/10.1007/BF00993104, 1993. Acessado em Janeiro de 2019.

[14] CHEN, K. "Deep Reinforcement Learning for Flappy Bird". . https:http://cs229.stanford.edu/proj2015/362_report.pdf. Acessado em Janeiro de 2019.

[15] GLOVA, M. "REINFORCEMENT LEARNING ALGORITHMS IN THE COMPUTER GAME FLAPPY BIRD". . http://www.itspy.cz/wp-content/uploads/2018/11/IT_SPY_2018_Diplomova_prace_55.pdf, 2018. Acessado em Janeiro de 2019.

[16] LENARD LINDSTROM, RENÉ DUDFIELD, P. S. N. D. T. K. O. "Pygames". . https://www.pygame.org/news, 2018.

[17] DANIELE GRATTAROLA, CARLO D'ERAMO, M. P. "Deep Q-learning for Atari Games". . https://github.com/danielegrattarola/deep-q-atari, 2013.

[18] MATTHIAS PLAPPERT, RAPHAEL MEUDEC, R. R., OTHERS. "Deep Reinforcement Learning for Keras". . https://github.com/keras-rl/keras-rl, 2019.

[19] DENNY BRITZ, ALEX BAILO, A. K. S. K. "Implementation of Reinforcement Learning Algorithms". . https://github.com/dennybritz/reinforcement-learning, 2018.

[20] BEN LAU, K. G. "Keras-FlappyBird". . https://github.com/yanpanlau/Keras-FlappyBird, 2018.

[21] CHOLLET, F. "Keras". . https://keras.io/, 2018.

[22] VAN ROSSUM, G., DRAKE JR, F. L. "Python documentation - Deque". . https://docs.python.org/2/library/collections.html#collections.deque.

[23] CHOLLET, F. "Keras documentation - activation functions". . https://keras.io/activations/.

[24] TOM SCHAUL, JOHN QUAN, I. A., SILVER, D. "PRIORITIZED EXPERIENCE REPLAY". . https://www.researchgate.net/publication/324066514_Playing_Flappy_Bird_with_Deep_Reinforcement_Learning, 2016. Acessado em Dezembro de 2018.

[25] TIMOTHY P. LILLICRAP, JONATHAN J. HUNT, A. P. N. H. T. E. Y. T. D. S. . D. W. "CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING". . https://arxiv.org/abs/1509.02971, 2016. Acessado em Janeiro de 2019.

[26] VAN HASSELT, H., GUEZ, A., SILVER, D. "Deep Reinforcement Learning with Double Q-learning". . https://arxiv.org/abs/1509.06461, 2015. Acessado em Janeiro de 2019.