Universidade de Brasília - UnB
Faculdade de Tecnologia - FT UnB
Engenharia Elétrica

Treinamento de agentes jogadores de futebol usando aprendizado por reforço

Autor: Lucas Bamidele Tinoco Kalejaiye
Orientador: (Dr. Alexandre Ricardo Soares Romariz)

Brasília, DF
2019

Lucas Bamidele Tinoco Kalejaiye

Treinamento de agentes jogadores de futebol usando aprendizado por reforço

Trabalho de Conclusão de Curso submetida ao curso de graduação em Engenharia elétricada Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Elétrica

Universidade de Brasília - UnB

Faculdade de Tecnologia - FT UnB

Orientador: (Dr. Alexandre Ricardo Soares Romariz)

Brasília, DF

2019

Lucas Bamidele Tinoco Kalejaiye
Treinamento de agentes jogadores de futebol usando aprendizado por reforço/ Lucas Bamidele Tinoco Kalejaiye. – Brasília, DF, 2019-
57 p. : il. (algumas color.) ; 30 cm.

Orientador: (Dr. Alexandre Ricardo Soares Romariz)

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade de Tecnologia - FT UnB , 2019.

1. Deep Learning. 2. Aprendizado por reforço. I. (Dr. Alexandre Ricardo Soares Romariz). II. Universidade de Brasília. III. Faculdade de Tecnologia. IV. Treinamento de agentes jogadores de futebol usando aprendizado por reforço

CDU 02:141:005.6

Lucas Bamidele Tinoco Kalejaiye

Treinamento de agentes jogadores de futebol usando aprendizado por reforço

Trabalho de Conclusão de Curso submetida ao curso de graduação em Engenharia elétricada Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Elétrica

Trabalho aprovado. Brasília, DF, :

(Dr. Alexandre Ricardo Soares Romariz)
Orientador

Prof. Mylène C.Q. Farias
Convidado 1

Prof. João Paulo Leite
Convidado 2

Brasília, DF
2019

Agradecimentos

Agradeço a minha mãe, meu irmão e meu pai por sempre me apoiarem em toda minha jornada como estudante.

Ao Professor Romariz, pela disponibilidade, atenção e orientação durante o desenvolvimento deste trabalho.

À equipe DROID, que me proporcionou a chance de conhecer o mundo da robótica.

À equipe UnBall, por terem me ajudado na jornada do desenvolvimento desse trabalho, provendo suporte quando necessário.

Aos meus amigos e companheiros de turma: Ana, Caio, Guilherme, Gustavo, Pedro, Thiago e Vítor.

Resumo

O uso de técnicas de aprendizado de máquina se popularizaram muito com o avanço de tecnologias de computação de alto desempenho. Das técnicas desenvolvidas, uma das que mais tiveram destaque foram as que usam de redes neurais para aperfeiçoar e conseguir resultados. Esse trabalho têm como objetivo fazer um estudo de uso de técnicas de aprendizado de máquina - especificamente aprendizado por reforço - para conseguir treinar robôs jogadores de futebol, para que estes consigam fazer gols de forma consistente. Foram feitos experimentos com várias formas de treinamento, desde redes mais simples até algoritmos considerados estado da arte. O projeto foi motivado pelo trabalho da equipe *UnBall*, equipe de futebol de robôs da UnB, que compete na categoria *IEEE - Very Small Size*, e todo estudo foi feito com base nas regras dessa categoria.

Palavras-chaves: Deep learning. Aprendizado por reforço. Futebol de Robôs.

Abstract

The use of machine learning techniques have become very popular with the advances in high performance computing technologies. Within the developed techniques, the ones that employ neural networks are among the most popular. This work has the objective of studying machine learning techniques - specifically reinforcement learning - to train robot soccer players so that they can score goals consistently. Many experiments with various training methods were made, from the simplests of algorithms to state-of-the-art ones. The project was motivated by the work of *UnBall*, the robot soccer team from UnB that competes in the category *IEEE - Very Small Size*. The study was made using the rules from this category as basis.

Key-words: Deep learning. Reinforcement learning. Robot Soccer.

Lista de ilustrações

Figura 1 – Esquema de jogo na categoria Very Small Size. Imagem: SIRLab () . . 22
Figura 2 – Simulador com agentes jogando. 24
Figura 3 – Interação entre agentes e ambientes numa MDP. Imagem: [Sutton e Barto (2018)] 26
Figura 4 – Exemplo de como endereçar valores Q. 32
Figura 5 – Imagem de um neurônio. Imagem [Borges et al. (2015)] 33
Figura 6 – Esquema de uma rede neural. Imagem: [Facure (2017)] 34
Figura 7 – Neurônio artificial. Imagem: [mc.ai (2018)] 34
Figura 8 – Gráfico mostrando o comportamento da função L em função da razão r, para vantagens positivas e negativas. Imagem: Schulman et al. (2017) 38
Figura 9 – Menu do simulador 39
Figura 10 – Configuração de variáveis 39
Figura 11 – Dicionário de ações 42
Figura 12 – Algoritmo básico de uma DQN 46
Figura 13 – Sequência do agente seguindo a bola. 48
Figura 14 – Sequência do agente seguindo a bola. Podemos ver que o agente consegue realizar curvas suaves com o fim de se alinhar em relação a bola e chuta-lá. 49
Figura 15 – Recompensa média da rede DQN. Podemos observar que o treinamento, por mais ruidoso que seja, se estabiliza em torno de uma recompensa média entre 60 e 70. 50
Figura 16 – Recompensa média da rede DDQN. Podemos observar que o treinamento, mesmo ruidoso, alcança uma recompensa média maior no decorrer dos episódios, em comparação com a rede DQN. 50
Figura 17 – Recompensas média do treinamento em função dos episódios. A função destacada (em azul escuro) representa a função de recompensa suavizada, enquanto a função em azul claro representa todos os pontos de entrada (todas as recompensas por episódio) 52
Figura 18 – Sequência do jogador chutando a bola para o gol. Observamos aqui que o agente tenta se aproximar do centro do campo, para tentar empurrar a bola no centro do gol. 53
Figura 19 – Sequência do jogador chutando a bola para o gol. 54

Lista de tabelas

Lista de abreviaturas e siglas

DQN	Deep Q Network
DDQN	Double Deep Q Network
PPO	Proximal Policy Optimization
TRPO	Trust Region Policy Optimzation
MDP	Markov Decision Process ou Processos de Decisão de Markov
IEEE	Instituto de Engenheiros Elétricos e Eletrônicos
FPS	Quadros por segundo
DP	Programação Dinâmica
TD	Diferença temporal
DNN	Deep Neural Network ou Rede Neural Profunda
SGD	Stochastic Gradient Descent ou Gradiente Descendente Estocástico

Lista de símbolos

γ	Fator de desconto
π	Política de um agente
θ	Estimação de função de valor de estado ou valor de ação
ϵ	Probabilidade de uma ação aleatória
A_t	Ação dada em um instante de tempo t
S_t	Estado em um instante de tempo t
R_t	Recompensa em um instante de tempo t
G_T	Retorno acumulado dos instantes 1,2,...T
q_π	Função valor de ação π
v_π	Função valor de estado de uma política π
q_*	Função valor de ação ótima
v_*	Função valor de estado ótima
α	Taxa de aprendizagem
$\mathcal{D}$	Conjunto de dados de jogo
V_lin	Velocidade Linear
V_ang	Velocidade Angular

Sumário

1 INTRODUÇÃO 21
1.1 Competição 21
1.2 Sistema da UnBall 23
1.3 Simulador 23
1.4 Objetivos 24
1.5 Estrutura do Trabalho 24

2 APRENDIZADO POR REFORÇO 25
2.1 Processos de Decisão de Markov 25
2.1.1 Agentes e Ambientes 25
2.1.2 MDPs finitas 26
2.1.3 Recompensas 27
2.1.4 Políticas e Funções de valor 28
2.2 Programação dinâmica 29
2.2.1 Desvantagens da Programação Dinâmica 30
2.3 Métodos de Monte Carlo 30
2.3.1 Métodos de atualização de política 30
2.3.2 Exploração e aproveitamento 31
2.4 Aprendizado por diferença temporal 31
2.4.1 Q-learning 32
2.5 Deep Q-Learning 32
2.5.1 Redes neurais profundas 33
2.6 Double Deep Q-Learning 35
2.7 Proximal Policy Optimization 36

3 INFRAESTRUTURA 39
3.1 Simulador 39
3.2 Estrutura do sistema 40
3.2.1 Estados 40
3.2.2 Ações 41
3.3 Função de recompensa 43
3.4 Bibliotecas e Frameworks 44

4 RESULTADOS 45
4.1 Deep Q-Learning 45
4.2 Double Deep Q-Learning 47

4.3 Resultados com Proximal Policy Optimization 51

5 CONCLUSÕES E TRABALHOS FUTUROS 55

REFERÊNCIAS 57

1 Introdução

O estudo de Técnicas de Aprendizado por reforço para aprender a competir em jogos é um objeto de estudo frequente por pesquisadores e em empresas de tecnologia.

Em 2016, o *AlphaGO* - inteligência artificial desenvolvida pela empresa DeepMind - superou um jogador humano no jogo GO. Esta foi uma conquista extremamente importante na área de aprendizado de máquina, pois era estimado que problemas com o nível de complexidade de GO levariam muitos mais anos, e talvez até décadas, para serem resolvidos por computadores. [Silver et al. (2016)]

Um outro grande marco foi em 2013, quando a Deepmind conseguiu, pela primeira vez, criar uma inteligência que lia dados de sensores de várias dimensões para gerar políticas para vários jogos da Atari, inclusive batendo recordes de jogadores profissionais [Mnih et al. (2013)].

Talvez o marco mais impressionante, entrentanto tenha sido em 2017, quando a empresa OpenAI conseguiu criar um jogador artifical do jogo *Defence of the Ancients - Dota*. Nesse jogo de estratégia em tempo real, dois times de cinco jogadores tentam destruir a base inimiga. Os jogos têm uma complexidade altíssima, tendo mais de 100 heróis disponíveis para serem escolhidos, com partidas que podem chegar a durar 2 horas. Mesmo com a complexidade de ambiente reduzida, isso é, apenas em jogos de 1 contra 1 e várias restrições, a inteligência conseguiu derrotar todos os profissionais no maior evento de jogadores. Anos mais tarde, em 2019, o sistema conseguiu derrotar o melhor time do mundo em duas partidas, sem nenhum tipo de restrições [OpenAI (2018)].

Todos esses exemplos funcionam como grandes motivadores para a resolução de problemas de alta complexidade, em que definir estratégias manualmente se provam muito difíceis. Como o uso de técnicas de aprendizado por reforço não era uma área atualmente explorada dentro da equipe *UnBall*, ela se tornou uma ótima oportunidade para iniciação de tal projeto. O estudo pode indicar quais estratégias seriam melhores aproveitadas considerando a realidade da categoria em questão, e quais adaptações seriam necessárias para melhora de resultados.

1.1 Competição

Existem várias competições diferentes de futebol de robôs, e cada categoria possui suas especificidades. As categorias diferem desde a forma física do robô (humanoides, "cubos"ou até robôs simulados), método de aquisição de dados do sistema e quantidade de agentes presentes em campo. A categoria *IEEE Very Small Size* foi a categoria base

para o desenvolvimento do projeto. A categoria *IEEE Very Small Size* é formada por duas equipes de 3 robôs de dimensões máximas de 7.5 x 7.5 x 7.5 cm^3. O controle dos robôs é remoto, feito por um computador. Em nenhum momento um humano pode intervir nas ações dos robôs.

As informações sobre os jogadores e a bola são obtidas por meio de uma câmera, localizada em cima do campo, que manda informações para o computador. O computador processa essas informações e manda remotamente ações que o robô tem que tomar. O robô executa essas ordens, gerando um novo estado (novas posições de bola e dos robôs), que são novamente lidos pela câmera, criando um ciclo. O esquema de um jogo de futebol dessa categoria pode ser visto na Figura 1.

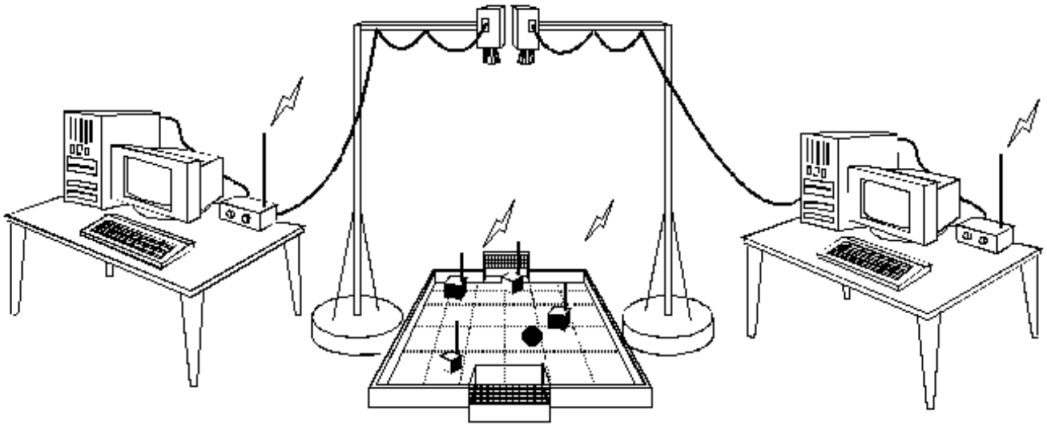

Figura 1 – Esquema de jogo na categoria Very Small Size. Imagem: SIRLab ()

1.2 Sistema da UnBall

A *UnBall* utiliza o sistema *ROS (Robot Operating System)* para dividir o sistema em pequenas partes - chamados de nós. Cada nó representa uma pequena atividade do sistema - controle de baixo nível (minimização de erro), controle de alto nível (definição de rotas), comunicação entre robôs e computador, visão computacional e, quando necessário, simulador. Os nós se comunicam de forma assíncrona e se encaixam para dar forma ao sistema controlador. Um dos nós importantes do sistema é o **simulador**, que representa o jogo de forma virtual.

1.3 Simulador

Um simulador é uma parte fundamental para o desenvolvimento de um projeto de aprendizado. Obter dados de um sistema real pode ser uma tarefa lenta e passível de acúmulo de erros de diversas fontes. Se pensarmos em futebol de robôs, estaríamos presos a constantes de tempo e velocidade reais, além de ruídos e erros provenientes da captação de imagens e controle do robô.

Com o auxílio de um simulador podemos processar muito mais experiência de jogo em um espaço de tempo real muito menor. Isso é possível aumentando a quantidade de quadros por segundo (*ou Frames Per Second - FPS*) sem aumentar as velocidades relativas a esse FPS. Ruídos relacionados a captação de dados são extremamente menores, pois não estamos lidando com erros aleatórios do mundo físico. A renderização do simulador pode ser visto na figura 2.

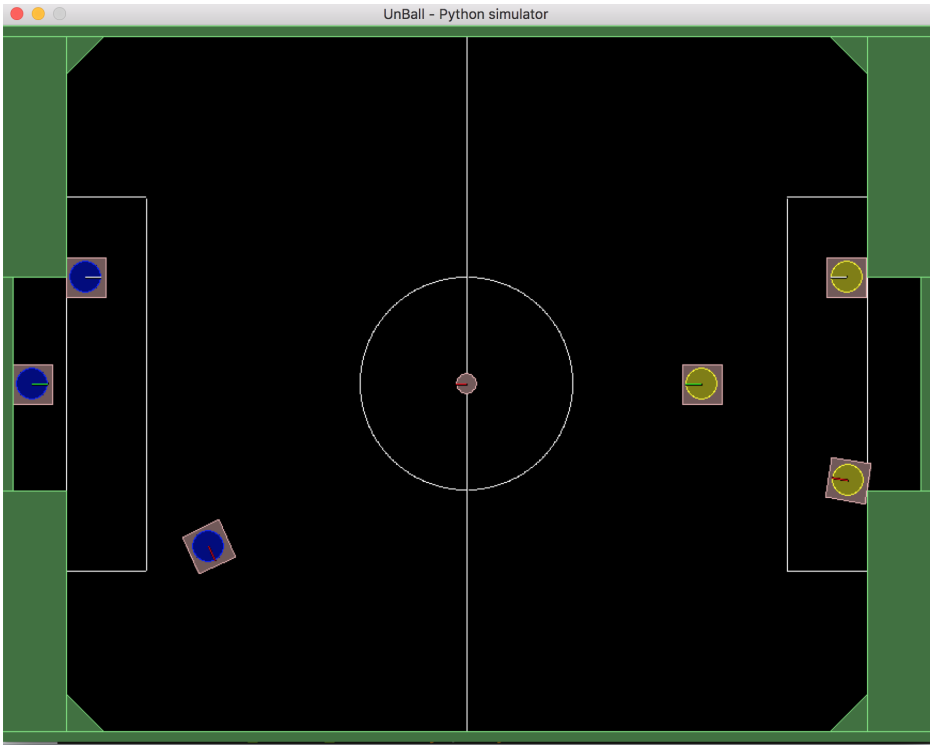

Figura 2 – Simulador com agentes jogando.

1.4 Objetivos

Os objetivos do trabalho são:

- Adaptar um ambiente de simulação para que ele possa comportar os elementos principais de um problema de aprendizado por reforço;
- Comparar diferentes algoritmos de aprendizado por reforço quanto sua eficiência e capacidade de gerar bons resultados;
- Treinar um agente para que ele aprenda a fazer gols de forma consistente.

1.5 Estrutura do Trabalho

No capítulo 2, são discutidos os aspectos teóricos de aprendizado por reforço, definindo seus elementos principais e também os algoritmos usados no projeto. No capítulo 3, são abordadas as questões relacionadas a infraestrutura do projeto, explicitando decisões que serviram como base para o treinamento do sistema. No capítulo 4, são apresentados e analisados os resultados, comparando a capacidade de gerar bons resultados de cada algoritmo. Por último, o 5º capítulo apresenta as conclusões e discute trabalhos futuros.

2 Aprendizado por reforço

O aprendizado por reforço é um dos paradigmas de aprendizado de máquina no qual um agente deve aprender a se comportar da melhor forma possível dentro de um determinado ambiente. Isso acontece por meio de interação com dito ambiente, e uma noção de recompensa cumulativa associada às consequências de suas ações. Boas ações devem ser recompensadas positivamente, e ações ruins recompensadas negativamente. Existem várias abordagens para um problema de aprendizado por reforço, mas todas elas derivam da ideia da existência de agentes - que tomam ações - e ambientes - que reagem a essas ações. Essa é a principal ideia dos **Processos de Decisão de Markov**.

2.1 Processos de Decisão de Markov

Os processos de decisão de Markov - também chamadas de MDPs (*Markov Decision Processes*) - são idealizações matemáticas amplamente usadas em problemas de aprendizado por reforço. As MDPs formalizam as noções de recompensas para uma sequência de ações, assim como os *trade-offs* entre receber uma recompensa atrasada ou imediatamente [Sutton e Barto (2018)].

2.1.1 Agentes e Ambientes

Antes de evocar definições formais, podemos considerar um problema simples, não relacionado a aprendizado. Se formos praticar um esporte - como o futebol propriamente - podemos definir todas as peças importantes do jogo: o campo, a bola, os gols e os jogadores. Numa partida tradicional, os jogadores podem tomar diversas ações - como correr, passar, chutar ou simplesmente ficar parados - e o ambiente em volta dele reagirá às suas ações. No caso de um chute, a bola com certeza mudará de lugar, indo para outro lugar do campo, saindo da zona válida de jogo ou até premiando um time com um ponto. A ação de um jogador poderá influenciar até na ação de outros jogadores, como no caso de um passe.

As MDPs são uma formulação muito intuitiva e direta de um problema de aprendizado. Como no exemplo anterior, precisamos definir as peças importantes do nosso sistema. A primeira peça fundamental são os **agentes**. Estes devem avaliar as condições atuais do ambiente - isto é - o **estado** do agente e tomar uma decisão - uma **ação**. Estas, por sua vez, são introduzidas no ambiente, que se alterará, criando um novo estado, além de uma resposta referente à qualidade dessa ação - uma **recompensa**. Esse novo estado deve ser lido pelo agente, que tomará uma nova ação, e assim fechando o ciclo.

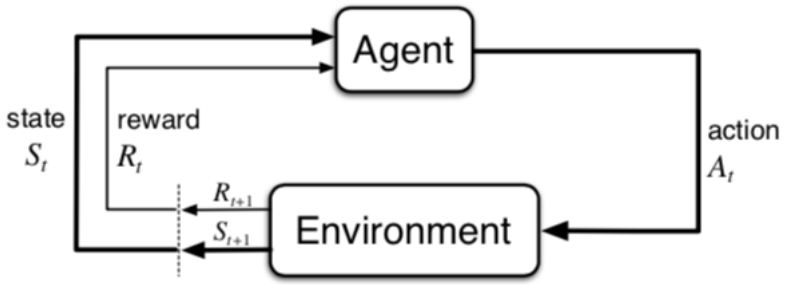

Figura 3 – Interação entre agentes e ambientes numa MDP. Imagem: [Sutton e Barto (2018)]

Essa forma de descrição do sistema se assemelha muito à forma como vemos o mundo real. No nosso exemplo do jogo de futebol, podemos chamar os jogadores de agentes e o campo, bola e gols como partes do ambiente. Os estados podem ser definidos pelas posições dos objetos, e a recompensa definida pela ocorrência de um gol.

A Figura 3 mostra um esquema de como um agente recebe um estado S_t, produz uma ação A_t e o ambiente (*Environment)* retorna um novo estado S_{t+1} e uma recompensa R_{t+1}.

É interessante observar também que o aprendizado por reforço foi muito estudado como uma das formas de resolver problemas clássicos de controle, como a estabilidade de um pêndulo invertido. É possível então, dentro desse paradigma, chamar os agentes de *controladores*, os estados de *sistema de controle* ou *planta* e as ações de *sinais de controle*. Neste trabalho, porém, não será usada essa nomenclatura.

O limite entre um agente e o ambiente depende muito das especificações do problema em questão. No caso de futebol de robôs, por exemplo, não existem dúvidas que o gol e a bola são partes do ambiente. Porém, partes do próprio robô também fazem parte do ambiente em questão. O atrito que o robô tem, as não linearidades do motor ou até sua velocidade real são variáveis que se encaixam melhor na definição de ambiente do que na de agente. Como regra geral, tenta-se limitar o máximo o "alcance" - isso é - o que é definido como agente em uma MDP.

2.1.2 MDPs finitas

Consideramos que o nossos espaços S_t, A_t e R_t são finitos, e representam os estados, ações e recompensas respectivamente. Dessa forma, sabemos que uma função de probabilidade associada a S_t e A_t são distribuições discretas bem definidas. Formalmente, podemos escrever que a probabilidade p do agente ser levado de um estado s a um estado s' a partir de uma ação a e receber uma recompensa r como:

$$p(s', r|s, a) \doteq Prob\{S_t = s', R_t = r|S_{t-1} = s, A_t = a\} \tag{2.1}$$

Sabemos que o somatório das probabilidades deve ser 1.

Considerando que R, S e A são, respectivamente, os espaços das recompensas, estados e ações possíveis, podemos definir uma recompensa r como a recompensa esperada dado um par estado-ação:

$$r(s,a) \doteq E\{R_t|S_{t-1}=s, A_{t-1}=a\} = \sum_{r\in R} r \sum_{s'\in S} p(s',r|s,a) \tag{2.2}$$

É importante salientar que a recompensa R_t está associada a um estado, isso é, o quão "bom" é estar nesse estado (incluindo a possibilidade desse estado ser um caminho para outros estados melhores).

Existem outras formas de expressar essas equações (dependendo da forma como são definidas as transições), mas usaremos majoritariamente essas definições daqui em diante.

2.1.3 Recompensas

A ideia principal de aprendizado por reforço foi formada a partir da observação de animais. A tarefa de adestrar um cachorro, por exemplo, se dá por recompensar petiscos a comportamentos bons e repreender comportamentos ruins. Depois de várias repetições dessas mesmas situações, um cachorro consegue, mesmo que sem recompensas, aprender a sentar ou deitar a partir de comandos, e também de não deixar dejetos em lugares inadequados. Essa noção de recompensa é equivalente em MDPs, onde recompensamos positivamente ações boas, e negativamente as ruins.

De forma simplificada, para cada estado S em um tempo t, temos uma recompensa R_t associada. Essa recompensa pode ser um simples número que indique o quão bom (ou ruim) é esse estado. Com a existência desse número, a tarefa do agente se torna tomar uma ação que o retorne a maior recompensa.

Para fins de treinamento, podemos delimitar o tempo de atuação do agente. Essa delimitação é chamada de **Episódio**, e o fim dele geralmente significa a chegada em um estado terminal (a existência de um gol, por exemplo). É possível não usar essa delimitação, isso é, em tarefas contínuas, mas essas não serão exploradas nesse trabalho.

Como citado anteriormente, as MDPs possuem uma característica de sequenciamento de estados. Desta forma, o objetivo de um algoritmo de aprendizado se torna maximizar o retorno esperado, isso é, o retorno acumulado depois de um número N de iterações. Podemos expressar isso matematicamente como G_t:

$$G_t = R_1 + R_2 + R_3 + ... + R_N = \sum_{t}^{N} R_t \tag{2.3}$$

onde R_t é a recompensa obtida durante o instante t de um episódio.

Porém, se consideramos a equação 2.3, podemos ver que não existe diferença entre chegar num estado ótimo rapidamente ou lentamente. Dessa forma, não há nada que diferencie um agente que chegou em um estado ótimo em 20 passos, e um que teve o mesmo resultado, porém com 2000. De forma geral, o primeiro cenário é muito mais vantajoso do que o segundo. Para abordar esse problema trabalhamos com a ideia de descontos. Para cada passo t, multiplicamos a recompensa por um fator de desconto γ, em que $0 < \gamma \leq 1$. A nova função de retorno acumulado se torna:

$$G_t^{'} = R_1 + \gamma R_2 + \gamma^2 R_3 + ... = \sum_{t=0}^{N} \gamma^t R_{t+1} \tag{2.4}$$

Podemos observar que para valores de γ próximos de 1, o agente pode aprender a tomar ações piores imediatamente que levem eventualmente a ações melhores no futuro. Para valores baixos, o agente tenderá a escolher as melhores ações imediatas, desconsiderando recompensas futuras. Não existe um valor único de γ, e ele deve ser ajustado juntamente com outros hiper parâmetros de treinamento para se adequar ao problema em questão.

2.1.4 Políticas e Funções de valor

Voltando novamente ao nosso exemplo de um jogo de futebol, podemos nos perguntar: o que define um bom jogador? De maneira geral, podemos afirmar que é a capacidade que ele tem de tomar boas decisões em situações variadas de jogo. No caso de aprendizado por reforço, isso também é verdade, porém aqui modelamos matematicamente o que são boas decisões e bons estados para se estar. Para isso, definimos **políticas** - mapeamento entre ações e estados - para auxiliar no processo de decisão. Essa política irá ditar qual ação eu devo tomar dado o estado em que eu me encontro. Usamos o símbolo π para se referir a política de um agente.

Funções de valor (ou valor de estado - v_π) efetivamente avaliam o quão bom é estar em um estado sob a política π. Elas estimam quanto deve ser a o retorno esperado dado o meu estado atual S. Formalizando:

$$v_\pi(s) \doteq E_\pi[G_t|S_t = s] \tag{2.5}$$

É mais interessante, porém, estimar o retorno esperado de determinada ação a em um estado s, seguindo-se u política π.

$$q_\pi(s, a) \doteq E_\pi[G_t|S_t = s, A_t = a] \tag{2.6}$$

A função q_π se chama função valor de ação. Essa função é muito usada para técnicas *q-learning* e derivadas, que serão discutidas nos capítulos seguintes.

Trabalhando com outras definições e alguns arranjos matemáticos, determina-se a seguinte equação:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.7}$$

No caso da equação 2.7, a função $\pi(a|s)$ representa a probabilidade que $A_t = a$ e $S_t = s$.

A equação 2.7 é conhecida como equação de **Bellman** para v_π, e relaciona um estado com seus estados subsequentes. Ela combina as equações de retorno esperado e descontos para representar a soma das probabilidades de cada evento. Podemos notar que essa equação é, em sua essência, uma média ponderada dos retornos esperados com suas devidas probabilidades de acontecerem. A demonstração dessa equação não será abordada nesse trabalho.

Para todos os problemas de aprendizado, partimos do princípio em que existe uma política ótima, isso é, onde as ações tomadas culminarão no retorno esperado máximo. O objetivo em um treinamento de uma agente inteligente é computar tal política.

2.2 Programação dinâmica

A programação dinâmica - *DP - Dynamic Programming* se baseia em diminuir um problema grande e complexo de se resolver em vários subproblemas menores e mais simples. Técnicas de programação dinâmica foram estendidas para diversas áreas de engenharias pelo seu poder de resolver (ou às vezes apenas aproximar) problemas que de outra forma seriam impossíveis de serem resolvidos. Em áreas da ciência da computação, diversos problemas com complexidade exponencial $O(2^n)$ podem ser simplificados para problemas de complexidade quadrática ($O(n^3)$) ou cúbica ($O(n^3)$)

Para técnicas de aprendizado por reforço usando programação dinâmica, voltamos à equação 2.7. Podemos observar que a melhor ação, isso é, a ação a sobre uma política ótima v_* (valor de estado) ou q_* (valor de ação) deverá ser a ação que tem o maior retorno esperado. Nesse caso, temos:

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.8}$$

ou

$$q_*(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_a q_*(s', a')] \tag{2.9}$$

Existem 2 formas principais de abordar uma solução com programação dinâmica: *Iteração de Valor* e *Iteração de Política*

2.2.1 Desvantagens da Programação Dinâmica

Um maior problema dos métodos descritos anteriormente é que a ordem de grandeza do tempo necessário para o treinamento de uma solução cresce muito rápido, dependendo da complexidade do sistema em questão. Em sistemas muito complexos, com muitos estados e muitas ações, usar esses métodos se torna computacionalmente ineficiente. Nos últimos anos, soluções de programação dinâmica não são muito comuns pois vários problemas atuais de aprendizado lidam com espaços de estados e ações enormes.

Outro problema dessa classe de algoritmos são a necessidade de **conhecimento total do ambiente**. Isso significa que precisamos ter informações de todos os estados antes de iniciar o treinamento. Isso se torna um problema também quando não temos essa informação, e ela depende de iterações com o ambiente.

Mesmo com esses problemas, esses métodos e essas equações ainda são muito citados em estudos. Isso é consequência do fato de eles utilizarem de forma muito direta o Processo de Decisão de Markov, e o entendimento de que suas equação auxiliam o desenvolvimento de técnicas mais complexas.

2.3 Métodos de Monte Carlo

Os métodos de Monte Carlo contam com a exploração aleatória do sistema para se obter conhecimento sobre o sistema. Isso é uma melhora considerável em relação a programação dinâmica pois não necessitamos de conhecimento prévio do ambiente. Nesses métodos a ideia principal é calcular as médias dos retornos (G_t) relacionada a uma sequência de ações.

De forma geral, o agente deve coletar uma quantidade de experiências, isso é, uma amostra de estados, ações e recompensas para computar o treinamento. Uma sequência de exploração é chamada de **episódio**, e a política deve ser atualizada a cada fim destes.

Para realizar o treinamento, deve-se iterar sobre as amostras de um episódio, computando os retornos parciais G_t para cada passo. Então, deve-se calcular a média dos retornos parciais e os atribuir ao valor de ação $Q(s, a)$. A melhor ação então deve ser o argumento máximo desse valor de ação.

2.3.1 Métodos de atualização de política

Existem diversas formas de gerar episódios para os Monte Carlo. Enquanto explorar de forma aleatória pode, eventualmente, gerar uma estratégia ótima, não podemos garantir o tempo necessário para que isso ocorra. Existem formas de acelerar - ou ao menos refinar - a política de um agente, por meio de usar a própria política para a geração de um episódio. Métodos **on-policy** avaliam e melhoram a política a partir de ações tomadas

por ela mesma. Métodos **off-policy** separam essas duas partes, onde os dados de treinamento não são atualizações advindas de interação com o ambiente (isso é, o aprendizado é feito a partir de um comportamento pré definido).

2.3.2 Exploração e aproveitamento

Em um projeto de aprendizado por reforço, podemos perceber que um agente precisa explorar o ambiente tanto quanto usar de conhecimento prévio para aperfeiçoamento de sua política. Isso é uma dualidade comum em problemas de aprendizado de máquina, conhecido como *Exploration* (ou **exploração** - aprender sobre o sistema) contra *Exploitation* (ou **aproveitamento** - aproveitar de seu conhecimento prévio para melhorar).

Uma alternativa para equilibrar exploração e exploitação, é o método ϵ-*greedy*. Nesse nétodo, o agente tem chance ϵ de tomar uma ação dentro da política, e uma chance $1 - \epsilon$ de escolher uma ação aleatória. Isso produz um balanço interessante em exploração ou exploitação, que depende fortemente do valor de ϵ.

2.4 Aprendizado por diferença temporal

Assim como métodos de Monte Carlo, métodos que utilizam diferença temporal aprendem por interagir com o ambiente. Isso permite que se aprenda políticas sem a necessidade de conhecimento prévio do ambiente. Porém, diferentemente de Monte Carlo, um agente não precisa esperar o fim de um episódio para poder atualizar suas funções. Esses métodos podem ser vistos como uma junção de ideias de programação dinâmica e Monte Carlo.

As regras de atualização para um aprendizado por diferença temporal pode ser dado por:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.10}$$

O termo $R_{t+1} + \gamma V(S_{t+1})$ é chamado de *TD Target* ou alvo do treinamento, e V aproxima a política de valor de estado v. O termo α representa a taxa de aprendizado do sistema.

Usando a equação 2.10, podemos definir o algoritmo mais básico dessa classe, o TD(0). Iremos, para cado passo de um episódio, tomar uma ação, e observar as recompensas R e o estado novo do ambiente. Com esses valores, atualizamos nossa função de valor usando a equação acima. É importante que a ação tomada, porém, seja a dita pela política π para um estado S.

Apesar da simplicidade desse método, ele pode ser bem robusto e prover resultados interessantes. Variações desse algoritmo já foram usados para jogar gamão [Tesauro (1995)]. O método **Q-learning** têm suas raízes nessa forma de aprendizado.

2.4.1 Q-learning

O *Q-learning* é uma forma de aprendizado que tenta aprender uma função de valor de ação Q que aproxima uma política ótima q_*.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_{t+1}, A_t)] \tag{2.11}$$

O algoritmo para Q-learning é quase o mesmo do método TD(0) abordado anteriormente, com a diferença da função de atualização é a 2.11. A ação escolhida pode ser tomada a partir de um paradigma ϵ-*greedy* ou *greedy* (melhor ação definida pela política atual).

A maneira mais comum de representar essas funções são com tabelas - as **Q-Tables** - onde um eixo representa as ações que podem ser tomadas e outra representa os estados do ambiente, representadas na Figura 4.

Q-TABLE	Ação 1	Ação 2	...	Ação N
Estado 1	Q(S1, A1)	Q(S1, A1)	...	
Estado 2	Q(S2, A1)	Q(S2, A2)	...	
...	...	...	...	
Estado N				Q(Sn, An)

Figura 4 – Exemplo de como endereçar valores Q.

Podemos analisar o padrão de crescimento dessas tabelas em relação ao número de estados e ações. Para um agente que pode tomar um número N de ações, num ambiente que pode ser descrito por M estados, a quantidade de valores Q possíveis são M*N. Para valores grandes de M e N, essa solução pode se tornar inviável, pois o custo computacional seria muito grande para ser resolvido em uma escala de tempo razoável.

Note também que essa solução, assim como todas discutidas anteriormente, funcionam apenas para ambientes que têm um número discreto e finito de estados. Muitos problemas, inclusive o abordado neste trabalho, não possuem essas características, o que torna essa solução inadequada. Serão discutidas nos próximos tópicos modos de adequar essa solução a problemas maiores e mais complexos.

2.5 Deep Q-Learning

Todas as soluções anteriores dependiam muito do uso de vetores ou tabelas para descrição de funções de estado ou valor de ação. Para problemas muito complexos, vimos

que essas soluções ficam computacionalmente inviáveis. Com isso em mente, temos que achar alternativas que possam descrever essas funções com segurança.

Uma aproximação interessante é tentar estimar essas funções com outras funções mais simples de serem computadas. Existem vários métodos na área de aprendizado de máquina que nos permite fazer esse tipo de aproximação, com algoritmos de classificação e regressão. Essa é a proposta do **Deep Q-Learning** ou **DQN**: usar redes neurais profundas - *DNN*, do inglês *Deep Neural Networks* - para obter uma aproximação de uma função valor de ação Q. As redes neurais são uma ferramenta muito poderosa para aproximação de funções, e por isso têm sido extensamente usadas para resolução de problemas de aprendizado.

2.5.1 Redes neurais profundas

As redes neurais foram formuladas para tentar simular o jeito que o cérebro de animais funcionam - mais especificamente os neurônios (ilustrados na Figura 5). O cérebro possui milhares de neurônios interconectados que podem se ativar e desativar - provocando uma infinidade de comportamentos diferentes. Esse ato de ativar é chamado **disparo** de um neurônio.

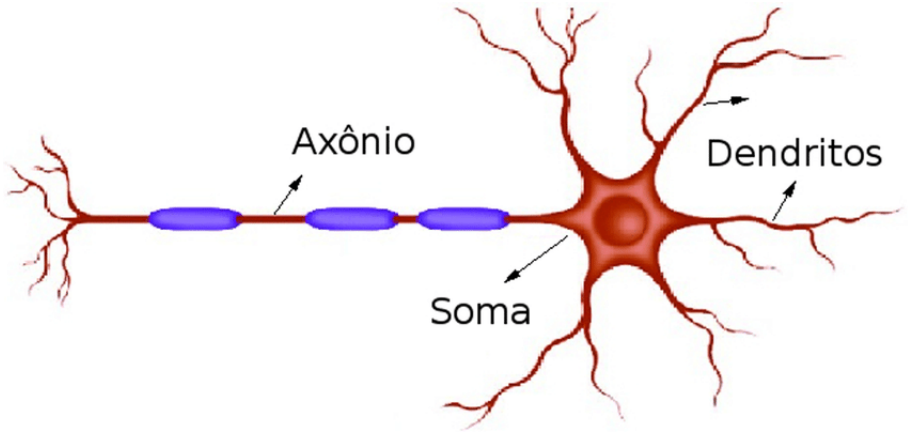

Figura 5 – Imagem de um neurônio. Imagem [Borges et al. (2015)]

Muitas vezes quando sentimos um cheiro ou ouvimos algum som, podemos observar comportamentos específicos que são reações a esses sentidos. Um cachorro que sente o cheiro de comida e corre para o seu prato de ração ou uma pessoa que se emociona ouvindo uma música da infância são exemplos dessas reações. Essas reações são efeito dos disparos de uma sequência de neurônios que ativam essas memórias específicas. A ideia das redes é imitar esse comportamento.

A estrutura da rede neural é uma concatenação de diversas camadas de processamento que tentam aprender a mapear um conjunto de entradas a um conjunto de saídas. A

figura 6 ilustra as conexões dos neurônios. Cada camada possui um conjunto de neurônios - célula básica da rede.

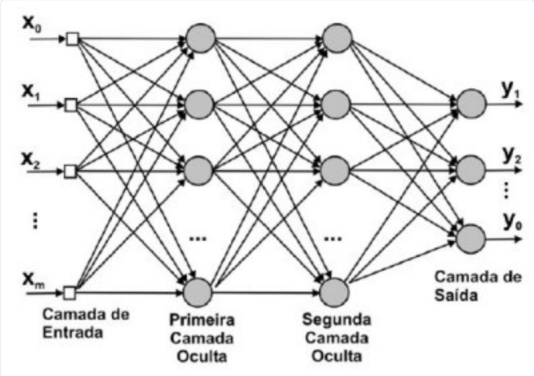

Figura 6 – Esquema de uma rede neural. Imagem: [Facure (2017)]

O neurônio artificial, ilustrado na Figura 7, é uma estrutura básica que recebe um conjunto de entradas x_i, multiplica por pesos p_i, o que produz uma saída y. Essa saída y pode se tornar uma entrada de outros neurônios ou a saída do sistema, no caso deste estar na última camada.

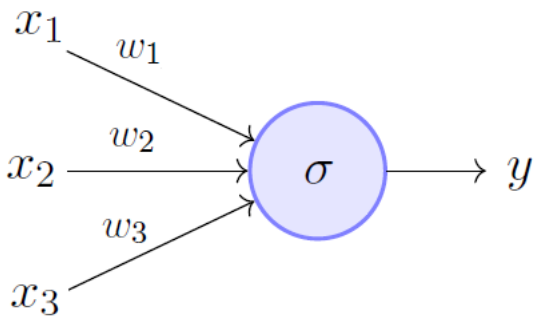

Figura 7 – Neurônio artificial. Imagem: [mc.ai (2018)]

Isoladamente, podemos ver um neurônio como regressor. O treinamento de um neurônio é justamente achar pesos ideais que generalizem um mapeamento entre entradas e saídas. Porém, os neurônios não conseguem aprender não linearidades em uma função. Mesmo com a existência de vários neurônios, eles por si só não conseguem descrever não-linearidades num sistema. Isso acontece pois a saída seria apenas uma soma ponderada de pesos e entradas, isso é, uma função linear. Para aumentar a capacidade de uma rede de neurônios, adicionamos uma **função de ativação** para cada neurônio.

Com essas funções de ativação, podemos escolher ativar ou não o neurônio dependendo do valor da saída. Isso introduz não-linearidades no sistema, o que aumenta

a capacidade de representação. Essa ativação do neurônio é similar ao disparo de um neurônio biológico, discutido no início dessa seção.

O treinamento de uma rede se dá por alimentá-la com entradas e observar as saídas que a rede gera com os pesos de cada neurônio. Devemos então comparar a saída obtida com a saída esperada dado tal conjunto de entrada. A partir desses valores, podemos calcular o grau de diferença entre eles. Essa é a função de custo L, também chamada de *loss function* ou *função de perda*. O objetivo é minimizar a diferença entre os valores obtidos pela rede e os valores esperados. Isso é equivalente a procurar minimizar a função de custo L.

A função de custo pode ser definida de várias formas. Uma maneira seria calculá-la como o módulo da diferença entre os valores de saída reais e esperados, aqui representados por y_{real} e $y_{esperado}$:

$$L = |y_{real} - y_{esperado}| \tag{2.12}$$

O módulo permite que o limite inferior da função seja 0, logo o processo de minimização não cai no risco de achar uma função infinitamente negativa. Apesar dessa função ter suas vantagens, a função de perda de quadrados mínimos é geralmente preferida.

$$L = (y_{real} - y_{esperado})^2 \tag{2.13}$$

Dessa forma, não só temos o mesmo limite inferior como estamos penalizando mais agressivamente divergências grandes. Isso se deve ao termo quadrático, que exagera desvios grandes. Esse critério é usado amplamente em estatística, como no caso de regressões lineares ou polinomiais.

A partir do cálculo da função de perda, podemos usar um mecanismo para propagar esse erro pela rede, ajustando todos os seus pesos e fazendo com que a rede se adéque melhor. Esse mecanismo é o **backpropagation**, que consegue computar os gradientes da função de perda em função dos pesos de toda rede. Com ajuda desse mecanismo, podemos usar o algoritmo Gradiente Descendente Estocástico (ou *SGD*) para achar os mínimos da função desejada.

2.6 Double Deep Q-Learning

Uma das características mais problematicas do Q-learning é a instabilidade de treinamento dessas redes. Isso ocorre pois o treinamento depende de uma etapa de maximização sobre valores estimados, o que resulta em uma superestimação de valores de ação. Essa sobrestimação pode ser também uma consequência de ambientes ruidosos de treinamento, ou valores imprecisos de treinamento. A falha se encontra na necessidade de selecionar e avaliar uma ação a partir de uma mesma rede, que promove a escolha de

valores superestimados, resultando em estimações super otimistas. [Hasselt, Guez e Silver (2016)]

Como o nome sugere, o algoritmo Double Deep Q learning (ou *DDQN*) é uma adaptação de redes *DQN*, que foram desenvolvidas para remediar os problemas de sua antecessora. As DDQNs são implementadas usando duas redes neurais: uma para selecionar uma ação e outra para avaliar a ação. A rede que seleciona ações é atualizada de forma equivalente a rede DQN normal, enquanto a segunda copia os pesos da primeira a cada periodicamente, em uma frequência menor de atualização.

2.7 Proximal Policy Optimization

A *Proximal Policy Optimization* ou *PPO*, é uma algoritmo considerado estado da arte em aprendizado por reforço. Ela surgiu com o objetivo de diminuir as instabilidades de outros métodos de aprendizado por reforço ao mesmo tempo de ser fácil de implementar, ser eficiente com suas amostras e ser fácil de ajustar os parâmetros de treinamento.

O algoritmo pertence a classe dos *Policy Gradient Methods* (ou métodos de gradiente de política), que essencialmente não derivam a escolha da ação de uma aproximação da função de valor de ação ou valor de estado. Essas funções podem ser usadas durante a etapa de treinamento, porém não são necessárias para escolha de ações.[Sutton e Barto (2018)]

O método se baseia na noção de vantagem, ou *Advantage*, que é uma computada como sendo a diferença entre os retornos reais G_t de um episódio e uma estimativa $\hat{e}_t$ destes:

$$\hat{A}_t = G_t - \hat{e}_t. \tag{2.14}$$

Definimos também uma *política estocástica* π_θ. Isso implica que a saída da política é uma distribuição de probabilidade. O índice θ indica que a política é parametrizada (no caso, por uma rede neural).

A partir da função de vantagem, podemos calcular a função de custo L, que pode ser dada de várias formas. Uma forma bastante comum é usar o logaritmo das probabilidades das ações de uma política π multiplicado por essa vantagem:

$$L = \hat{E}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t]. \tag{2.15}$$

Uma vantagem positiva significa que a ação teve um retorno melhor do que o esperado. Consequentemente, aumenta-se a probabilidade de tal ação ocorrer. No caso contrário, em que a vantagem é negativa, a ação foi pior do que a esperada, e diminuímos a probabilidade dela ocorrer nesse estado. Um problema dessa abordagem acontece quando se tenta fazer várias atualizações de descida de gradiente em um *batch* de treinamento. Isso

gera atualizações muito grandes na política, que prejudicam o treinamento. [Schulman et al. (2017)]

O método PPO deriva diretamente de outra técnica, a *Trust region policy optimization*, ou *TRPO*. Com o objetivo de evitar que a política nova seja muito distante da política anterior, usa-se de um limitador, mantendo a política em regiões de confiança. O método TRPO usa uma método chamado *KL-constraint*, em que a atualização da rede está sujeita a uma delimitação por uma função KL.

$$\text{Maximizar } \theta$$

$$\hat{E}_t[\frac{\pi_\theta(a_t|s|t)}{\pi_{\theta_{old}}(a_t|s|t)}\hat{A}_t]. \tag{2.16}$$

$$\text{sujeito à}$$

$$\hat{E}_t[KL[\pi_{\theta_{old}}(\cdot|s|t), \pi_\theta(\cdot|s|t)]] < \delta. \tag{2.17}$$

A equação 2.17 é uma forma de impedir que novas políticas sejam muito diferentes de políticas anteriores. O termo a esquerda (chamado de divergencia Kullback–Leibler, ou KL) mede o quanto a política nova é diferente da política atualizada.

O método PPO sugere uma outra forma de limitar grandes atualizações na função θ. Usando r_t para representar o termo que multiplica a função de vantagem, temos:

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t]. \tag{2.18}$$

A função *clip* "poda" variações muito grandes na política, deixando o valor do produto $r_t(\theta)\hat{A}_t$ truncado entre o intervalo $[1-\epsilon, 1+\epsilon]$. Essa é uma alternativa mais simples do que o método KL, porém é empiricamente mais efetivo.

Se observamos os gráficos da Figura 8, notamos que, para vantagens muito altas, a variação da política será limitada. Isso impede que a probabilidade de uma ação se torne elevada (possivelmente não sendo a melhor ação do agente), e depois em estágios mais avançados de treinamento, ela não consiga ser substituída. O contrário também vale, pois isso impede que valores fiquem muito baixos e depois não possam ser reavaliados.

O algoritmo formula que se deve considerar também a perda da estimação da função de valor. Para realçar ainda mais o algoritmo, é adicionado um bônus de entropia S da distribuição de probabilidade. A função final que deve ser maximizada é dada como:

$$L(\theta) = \hat{E}_t[L^{CLIP}(\theta) - c_1 L^F V + c_2 S[\pi_\theta](s_t)]. \tag{2.19}$$

O termo $L^F V$ é a perda do valor de função (no caso a perda do estimador de vantagem), S é o bônus de entropia da distribuição de probabilidades (o que garante que

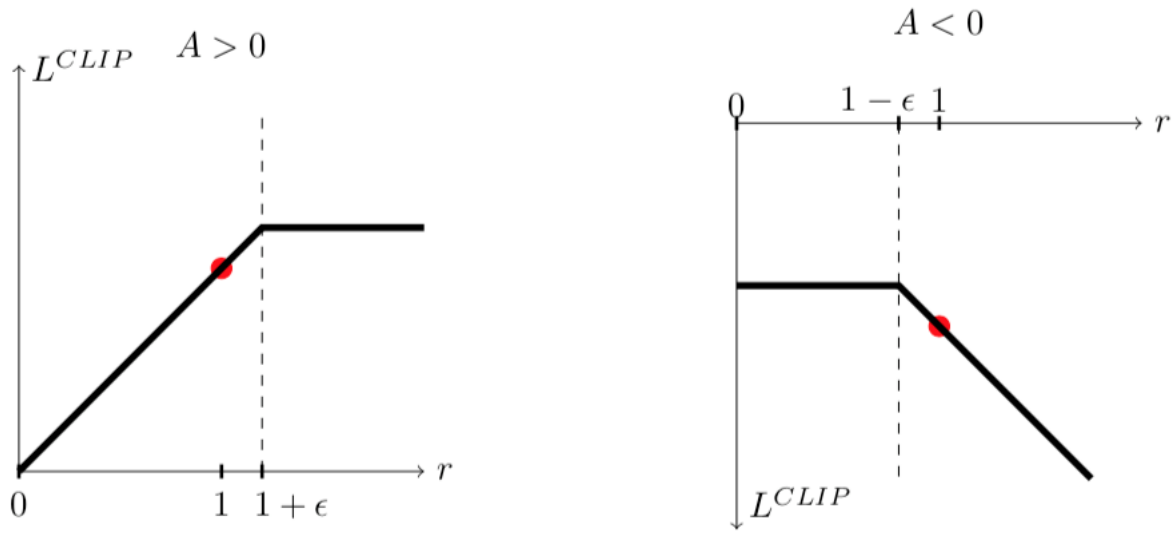

Figura 8 – Gráfico mostrando o comportamento da função L em função da razão r, para vantagens positivas e negativas. Imagem: Schulman et al. (2017)

ocorra exploração), e c1 e c2 são hiper parâmetros que devem ser ajustados. O termo c1 só deve ser considerado no caso em que existe um compartilhamento de parâmetros entre a política e a função de valor. Devemos então usar a subida de gradiente, a fim de achar os máximos da função.

3 Infraestrutura

3.1 Simulador

O simulador, como discutido anteriormente, é uma parte muito importante de um projeto de aprendizado de máquina de sistemas reais. Com ele, podemos implementar um Processo de Decisão de Markov, necessitando apenas delimitar agentes, ambiente e recompensas.

O simulador da UnBall é um dos nós do sistema ROS (citado anteriormente), que é usado apenas para situações de testes. Ele foi desenvolvido em *Python 3.6*, com o auxílio da biblioteca *Pygame*, *Box2D* e *PyMenu*. A primeira é uma biblioteca usada para facilitar a criação de jogos simples com interface gráficas, e a segunda cuida das interações entre os corpos no ambiente (movimentações e colisões). A terceira biblioteca é uma extensão da biblioteca de jogos, feita para criar menus interativos onde se pode modificar facilmente váriaveis de jogo. Diversas variáveis podem ser modificadas para adequar o simulador a algum teste, como número de jogadores do lado esquerdo, número de jogadores do lado direito, constantes de atrito e velocidade do jogo.

Figura 9 – Menu do simulador

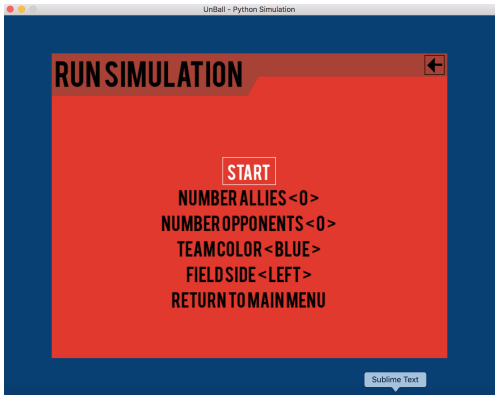

Figura 10 – Configuração de variáveis

O simulador funciona como um ambiente definido por uma MDP. Ele provê informações do ambiente, como velocidades lineares, velocidades angulares, posições e direções de cada objeto no campo. Com essas informações podemos formar o nosso estado S para cada instante de tempo. Para movimentação dos robôs, ele recebe um vetor de velocidades, sendo que cada elemento possui uma velocidade angular e linear. Com essas velocidades, o simulador produz impulsos até que a velocidade do robô se iguale à desejada. A velocidade com que isso acontece depende de configurações do sistema.

Ainda assim, algumas modificações precisaram ser feitas para facilitar o uso do simulador. Primeiramente, foi retirada toda a dependência do *ROS* do sistema, e com-

plementada manualmente qualquer parte necessária. Como o *ROS* se comunicava exclusivamente com o nó de estratégia, esse nó foi substituído por uma parte do código que roda de forma síncrona com o simulador. Isso permitiu que o desenvolvimento não fosse dependente de sistemas operacionais compatíveis com a versão do *ROS*. Além disso, foi retirada a necessidade de uma interface gráfica para o funcionamento. Sem a necessidade de renderização, obtive um aumento médio de quadros por segundo de mais de 30 vezes. Além disso, o treinamento em nuvem tornou-se possível (servidores paravam o código quando não conseguiam achar o *driver* de vídeo para renderizar os objetos).

3.2 Estrutura do sistema

3.2.1 Estados

Os estados de nosso sistema são as informações do campo de futebol em determinado instante de tempo. Ele é representado por vetor unidimensional, em que cada item corresponde a alguma informação específica, como:

- Posições no eixo X e Y da bola
- Velocidades no eixo X e Y da bola
- Distância entre a bola e o gol
- Posições no eixo X e Y de um jogador
- Velocidades no eixo X e Y de um jogador
- Ângulo em que um jogador está direcionado
- Distância entre o jogador e bola
- Ângulo entre a "frente" de um jogador e a bola
- Ângulo entre a bola e o centro do gol
- Velocidade angular de um jogador
- Ângulo absoluto entre o centro do jogador e o centro da bola

Podemos perceber que quanto mais variáveis o vetor tiver, mais completo e informativo é essa representação. Ao mesmo tempo, porém, mais complexo será aproximar uma função Q que determinará uma política ótima para os jogadores. Isso acontece pois a nossa rede terá que se ajustar a muito mais variáveis. Isso pode resultar em um aumento considerável no tempo de treinamento, e até em sua capacidade de gerar bons resultados. Por outro lado, quanto menos variáveis existirem no vetor, menor é a capacidade

do sistema de estimar uma política ótima. Logicamente, uma ação tomada com poucas informações disponíveis terá menos confiança do que uma que considera diversos fatores. Um dos grandes problemas da definição de estado está em encontrar um número razoável de variáveis de estados sem aumentar excessivamente a complexidade do sistema. Para isso, temos que tentar usar medições que representem as informações mais importantes.

Podemos observar na lista de variáveis, que temos variáveis absolutas (como a posição dos objetos), assim como relativas (como distâncias entre objetos). A inclusão do segundo grupo é uma forma de tentar facilitar o aprendizado da rede, resumindo informações importantes diretamente para o agente. Com a inclusão desses, foi observado uma melhora considerável no desempenho da estratégia. Antes da implementação dessas medidas, inclusive, os treinamentos resultavam em o agente executando apenas uma ação (como rodar dentro de seu próprio eixo), mesmo depois de milhões de interações com o ambiente. Após a implementação, observamos comportamentos mais interessantes como se aproximar da bola e até, dentro de circunstâncias específicas, fazer gol.

3.2.2 Ações

As ações, como discutido anteriormente em MDPs, são a forma que o agente tem de interagir com o ambiente. No nosso campo de futebol de robôs, é natural que isso seja a relacionado a movimentação do robô. Como citado anteriormente, o simulador recebe 2 valores - velocidade angular e velocidade linear - para inserir no robô. A partir dessa informação, podemos definir várias formas de utilizar a saída de nossa rede.

Podemos usar posições desejadas como a saída de nossa rede. Essa técnica, quando combinada com um sistema de planejamento de rotas e um controlador que as converte em velocidades, podem ser usadas para movimentação do robô. A vantagem de tal abordagem é a simplificação do sistema: a rede não tem que aprender as consequências de aplicação de velocidade, mas apenas as posições que oferecem melhores recompensas. Essa forma foi descartada no início, pois a implementação dos 2 controladores - o planejador de rotas o conversor de velocidades - é uma tarefa de devida complexidade que poderia tomar um bom tempo até ser corretamente executados.

Podemos, por outro lado, controlar as acelerações dos robôs. Dessa forma, não estamos trabalhando com velocidades, e sim com as variações delas. Essa foi uma das tentativas de aproximação do problema, porém ela se apresentou muito instável e seus resultados foram insatisfatórios.

Por último, podemos usar diretamente as velocidades como saídas de nossa rede. Dessa forma, precisamos apenas definir as velocidades válidas, e fazer com que a nossa rede retorne a velocidade que tem maior probabilidade de retornar a maior recompensa. Essa abordagem é ainda instável, porém foi a forma escolhida, visto que com a regulação

de parâmetros de treinamento, obtivemos resultados satisfatórios.

Além de escolher a função da rede final, podemos ainda deliberar o *espaço de ação* que será usado. Existem 2 formas de descrever o espaço de ação: contínuo ou discreto. No primeiro caso, a velocidade angular e linear são valores contínuos que variam entre $(-V_{min}, V_{max})$. Neste caso, a nossa rede funcionaria de forma similar a um problema de regressão, com 2 saídas contínuas que se traduziriam nas velocidades desejadas. No segundo caso, precisamos definir as ações possíveis e associar um dicionário finito a elas. No sistema em questão, foram usados 3 valores possíveis de velocidade angulares $[V_{ang_{min}}, 0, V_{ang_{max}}]$ e 3 valores de velocidades lineares $[V_{lin_{min}}, 0, V_{lin_{max}}]$. Esses valores são combinados em duplas, e associados em um dicionário que mapeia 9 valores a cada dupla, como obserado na figura 11.

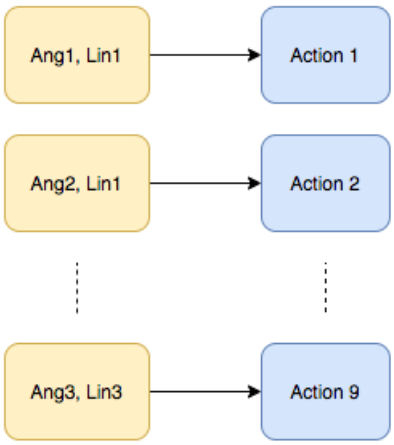

Figura 11 – Dicionário de ações

Podemos utilizar mais valores possíveis de velocidades a fim de podermos ter mais ações possíveis. Por exemplo, podemos adicionar valores lineares intermediários, como uma ação de andar para frente lentamente, normalmente ou rapidamente. Tal mudança aumentaria a complexidade do problema, e o enquadramento de uma rede para a determinação de uma boa política se torna mais difícil, como já discutido anteriormente.

O caso contínuo, como o nome já sugere, irá produzir saídas de velocidades lineares entre $V_{lin_{min}}$ e $V_{lin_{max}}$ e velocidades angulares entre $V_{ang_{min}}$ e $V_{ang_{max}}$. Nesse caso, precisaríamos trabalhar com funções de probabilidades contínuas (como uma função normal), que indicariam a região mais provável de render a melhor ação. Esse caso não foi estudado no desenvolvimento desse trabalho.

3.3 Função de recompensa

Uma das partes mais relevantes em um projeto de *reinforcement learning* é a **Função de Recompensa**. O projeto de uma função que representa bem o ambiente é vital para o sucesso de um projeto desse gênero. Essa função determina as recompensas que alimentaram a rede para cada tipo de ação, e consequentemente determina o comportamento que um agente tem. Recompensas podem ser tanto positivas quanto negativas, e sua magnitude influencia o quão desejado é chegar em tal estado.

Para o contexto de futebol de robôs, a ideia mais direta é recompensar positivamente gols a favor, e negativamente gols contra. Essa recompensa binária (ganhou ou perdeu o jogo) pode gerar bons resultados, como os resultados vistos em tentativas de agentes que jogam *Dota*, desenvolvido pela OpenAI [OpenAI (2018)]. O tempo de treinamento, porém, são ordens de grandeza maior, o que faz com que esse tipo de treinamento seja indesejável.

Podemos então definir ações intermediárias positivas que podem ajudar a chegar no objetivo final. Se aproximar da bola ou "chutá-la" (tocá-la), por exemplo, são ações desejadas que podem ser recompensadas. Ficar preso perto das paredes podem ser comportamentos "punidos". Por outro lado, é possível que essas recompensas intermediária atrapalhem o agente, fazendo que a política fique presa em um mínimo local.

Funções de recompensa contínuas e diferenciáveis são preferíveis, no sentido que o agente consegue aprender com mais facilidade. Isso é, é preferível aumentar uma recompensa, por exemplo, a medida que um robô se aproxima da bola, em vez de dar uma recompensa muito grande quando o robô encosta nela.

A fórmula básica para a recompensa R é dada pela seguinte fórmula 3.1

$$R_t = A_0(MAX_D - A_1 * D_{bg})/A_2 + A_3(\pi - A_4 * D_{ang_{bg}})/\pi + B_x * A_5 - B_y * A_6 \quad (3.1)$$

Onde:

- A_n são constantes que foram sendo ajustadas conforme o treinamento
- MAX_D é a constante que indica a distância máxima entre dois corpos no campo
- D_{bg} é a distância entre a bola e o jogador
- $D_{ang_{bg}}$ é o angulo entre a frente do jogador e a bola
- B_x e B_y são a posição da bola no eixo X e Y, respectivamente.

Também foram adicionadas algumas regras especiais, que sobrepunham a recompensa R definida anteriormente, gerando uma nova recompensa. Os seguintes acontecimentos resultavam em uma recompensa única, associada a tal evento:

- Gol a favor
- Gol contra
- Toque na bola

3.4 Bibliotecas e Frameworks

O código foi integralmente desenvolvido em *Python 3.6*. Visto que esta era a linguagem do simulador e também a existência de diversas bibliotecas que auxiliam o desenvolvimento de projetos de aprendizado de máquina, a escolha dessa linguagem era favorecida. Para compensar o tempo de execução naturalmente lento, foi usado o *numpy*, a biblioteca numérica do python, para auxiliar operações com vetores e aumentar a velocidade do código.

Para o processamento de redes neurais, foi usada a biblioteca *Keras*, que é uma API em python de alto nível para treinamento e implantação de redes profundas. O Keras roda em cima de outras plataformas de aprendizado - como *Tensorflow* - mas fornecendo uma interface de prototipagem fácil e rápida.

4 Resultados

No decorrer do projeto, foram estudadas diversas formas de treinamentos combinados com mudanças de hiperparâmetros. Os algoritmos foram comparados em relação a convergência, sensibilidade a hiperparâmetros e velocidade de treinamento.

Os experimentos foram feitos com apenas um robô em campo, e o objetivo era que ele aprendesse a fazer gols de forma consistente. O agente deveria aprender puramente pela interação com o ambiente.

O treinamento era em realizado episódios, em que uma episódio correspondia a um inicio de jogo. A bola sempre começava no meio do campo, e o robô poderia começar a partir de um local aleatório dentro do lado em que ele defende (isso é, se ele ataca no lado direito, o agente será inicializado em algum lugar no lado esquerdo do campo).

A avaliação dos algoritmos foi feita de duas formas: quantitativamente e qualitativamente. Na primeira maneira, observamos o gráfico de recompensas média por época em função das épocas. Um algoritmo que aprende deve ter um padrão de crescimento dessa recompensa média. Também foi observado a quantidade de gols que o agente fazia em 100 episódios, dado um limite máximo de 1000 quadros por episódio. A avaliação qualitativa é uma avaliação que envolve observar os comportamentos do robô, e julgar o quão ideal é esse comportamento.

Foram feitos testes com algoritmos Deep Q-learning, Double Deep Q-learning, e por último o Proximal Policy Optimization.

4.1 Deep Q-Learning

Os primeiros testes feitos foram utilizando uma rede *DQN*. A implementação de tal rede foi feita com base na implementação criada pela *Deepmind* para os jogos de *Atari*. A implementação se baseia na noção de *Experience Replay*, que é armazenar um conjunto de amostras s_t, a_t, r_t, s_{t+1} a cada passo do ambiente em um set de dados $\mathcal{D}$, chamado de *Replay Memory*. Esse conjunto de dados é amostrado aleatoriamente a cada atualização da função Q. A experiencia é obtida a partir de interação com o ambiente sob uma política ϵ-*Greedy* (isso é, ele tem um chance ϵ de escolher uma ação dita pela política π, e uma chance $1-\epsilon$ de escolher uma ação aleatória) [Mnih et al. (2013)]. Para favorecer a exploração inicial, esse ϵ começa muito baixo (próximo de 0), e vai aumentando de acordo com o quanto o agente explorou o ambiente. O algoritmo usado é descrito na Figura 12.

A função de recompensa desse ambiente inicialmente era simples: -1 para todos os estados, a não ser na ocorrência de um gol a favor (que retornaria +500 de recompensa),

Algoritmo 1: Algoritmo básico de uma DDQN

Inicializar *Replay Memory* $\mathcal{D}$;
Inicializar função ação-valor Q;
for *episódio = 1, M* **do**
  Inicializar sequencia de estados s;
  for *t=1, T* **do**
    Escolher ação aleatória a_t com chance $1 - \epsilon$ ou $a_t = \max_a Q^*(s_t, a)$;
    Executar ação a_t;
    Observar o novo estado e recompensas Adicionar o conjunto (s_t, a_t, r_t, s_{t+1}) em $\mathcal{D}$;
    if *O tamanho de amostras em* $\mathcal{D}$ *esteja acima de um limite* **then**
      Amostrar de $\mathcal{D}$ um *batch* (lote) de treinamento;
      Configurar $y_j = r_j$ caso o estado seja terminal;
      Configurar $y_J = r_j + \gamma \max_{a'} Q(s_{t+1}, a^{'})$;
      Treinar a rede sobre a função $(y_j - Q(s_j, a_j))^2$;
    end
    Aumentar o valor de ϵ Cortar valores de $\mathcal{D}$ caso tenha muitos valores
  end
end

Figura 12 – Algoritmo básico de uma DQN

ou um gol contra (retornaria -500). Os estados eram definidos pela posição absoluta dos objetos no campo, que eram por sua vez normalizados para valores entre -1 e 1).

Foram testadas várias versões desse treinamento, combinando diferentes taxas de aprendizagem para a rede neural, tamanho de rede (número de camadas e número de neurônios por camada), tamanho de *batch* e a taxa de decrescimento de ϵ e número de episódios. Em quase todas as tentativas, o algoritmo não conseguia convergir para uma política que fizesse algo interessante. Em várias situações, o agente acabava rodando em torno de seu próprio eixo, ou andando em círculos eternamente, sem nenhuma reação a mudança de posição da bola.

O primeiro resultado interessante aconteceu depois de uma combinação de 2 mudanças: uma diminuição no tamanho da rede neural que aproximava Q, e a adição de uma recompensa por tocar na bola. Nos primeiros testes, estava usando uma rede de 3 camadas ocultas, cada uma com 128 neurônios. Quando a rede foi testada com apenas 2 camadas ocultas e 40 neurônios cada, o robô começou a mostrar sinais de reação ao ambiente. Adicionalmente, colisões entre objetos começaram a ser registradas e recom-

pensadas. O robô começou a reagir à bola, mudando de posição quando muito distante da bola e tentando, mesmo que muitas vezes sem sucesso, tocar nela. Ainda sofria muito com o problema de ficar preso perto de paredes, ou entrar em uma região em que ficasse em um laço eterno.

Mesmo com essas duas mudanças, não foi possível aprimorar mais o agente, mesmo com tempos maiores de treinamento e mudanças nos parâmetros de treinamento. Até esse resultado era extremamente inconsistente: pequenas mudanças nos hiperparâmetros resultavam no comportamento repetitivo e em loop descrito anteriormente. Até a mudança da *seed*, isso é, o número que determinará a sequência de números aleatórios gerado pelo sistema, apresentava mudança considerável no treinamento.

Por último, o agente conseguiu fazer, em sua melhor versão, 14 gols em um teste de validação de 100 episódios. Esse resultado aponta que o algoritmo não é muito consistente em seu objetivo.

4.2 Double Deep Q-Learning

As diferenças entre o uso de redes *DDQN* e as redes *DQN* são bem pequenas, porém os resultados são visíveis. Após poucas interações, podemos ver que o robô reage muito melhor ao ambiente: Ele procura tocar na bola com mais frequência, e até consegue fazer gols dependendo de sua posição e/ou posição da bola.

Um dos testes observados foi feito modificando a função de recompensa, mudando os pesos A_k associado às variáveis. Enquanto as recompensas de gols se mantiveram, foi adicionado uma recompensa por estar virado em direção a bola e estar perto dela. Esta recompensa foi projetada de forma contínua, isso é, o quanto mais perto da bola (e quanto mais alinhado estiver em relação a bola), maior a recompensa. O resultado foi o agente achar uma política que se resume a correr atrás da bola. Apesar dele não tentar fazer gol, ele sempre movia atrás da bola de forma, independente de onde a bola estivesse e para onde se movesse. Esse resultado é interessante pois é o que demonstrou uma política mais consistente, em que os o comportamento do agente não é imprevisível.

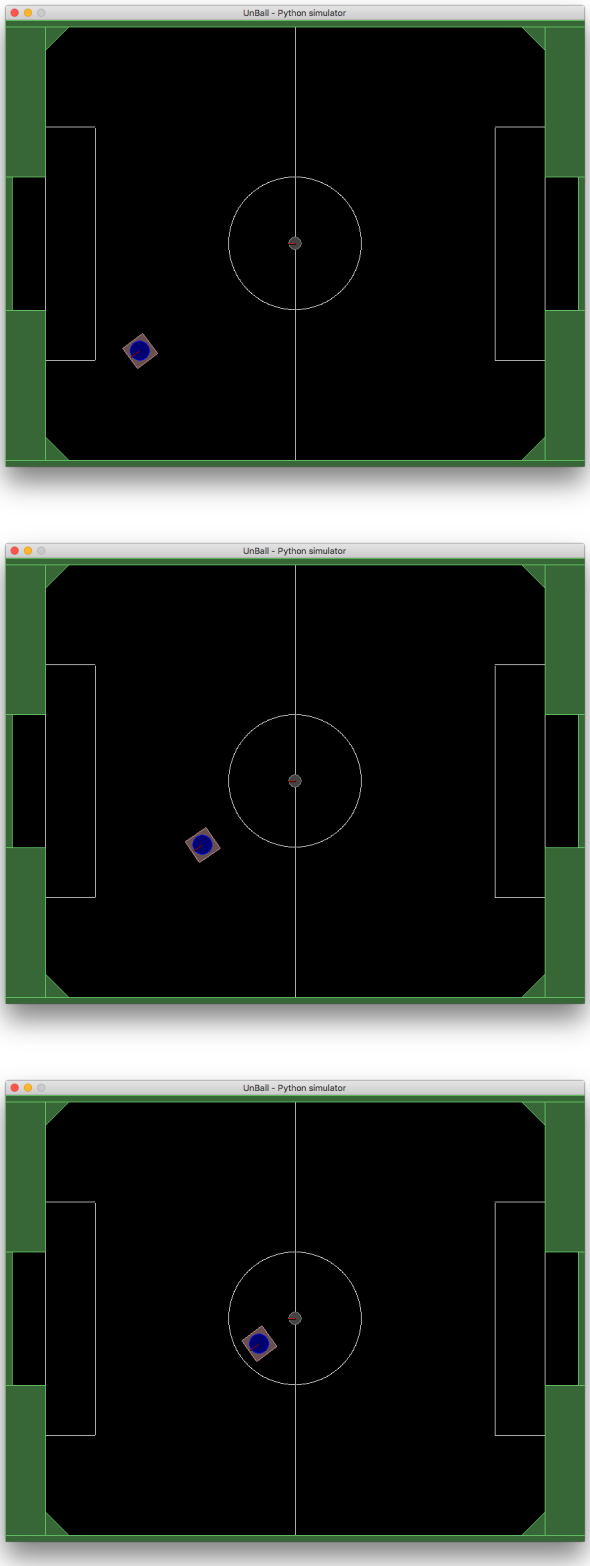

Figura 13 – Sequência do agente seguindo a bola.

Um fato curioso é a forma que o agente aprendeu a seguir a bola: se "movendo"de costas, como visto nas Figuras 13 e 14. A função de recompensa premiava, além de fazer gol e tocar na bola, a direção em que o robô está apontado (se estiver com sua parte frontal

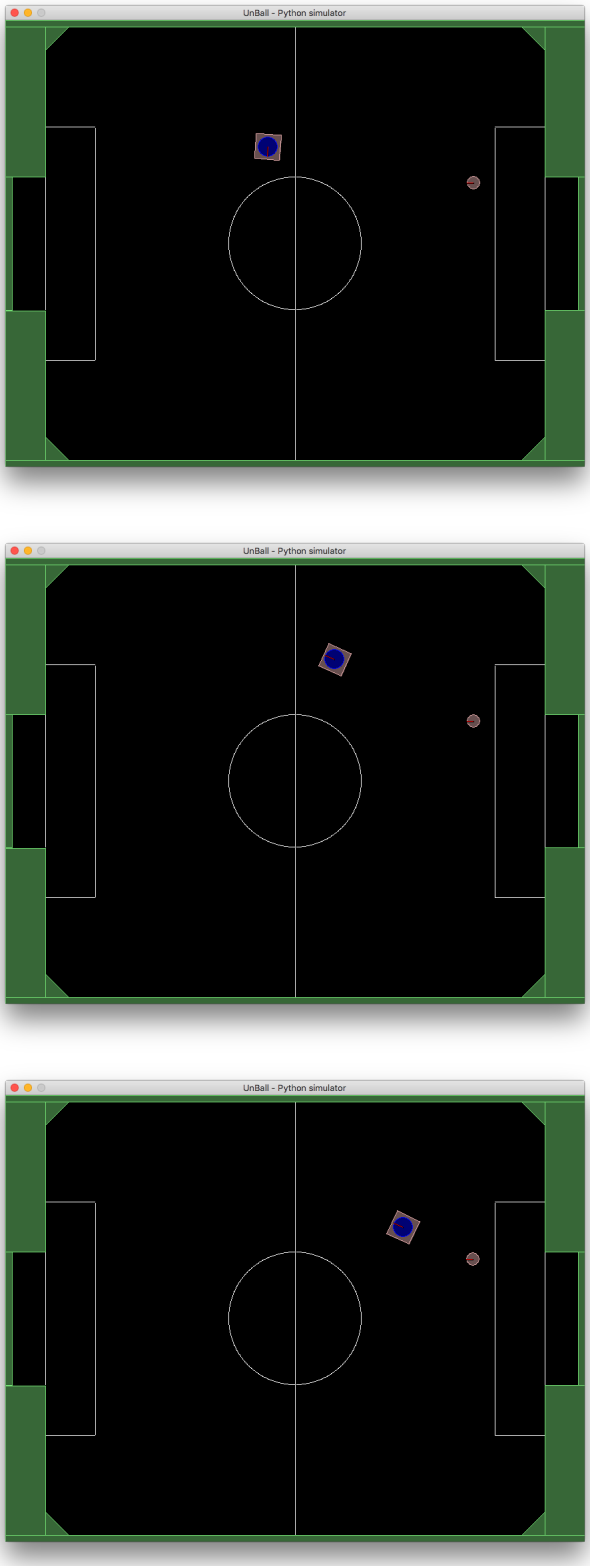

Figura 14 – Sequência do agente seguindo a bola. Podemos ver que o agente consegue realizar curvas suaves com o fim de se alinhar em relação a bola e chuta-lá.

alinhada com a bola, a recompensa é aumentada). Algumas diferenças existiam entre o movimento para frente e o para trás: o movimento para trás tem menor velocidade. Foram

feitos várias combinações de testes, mudando a magnitude das velocidades, invertendo os valores na função de recompensa, aumentando tempo de treinamento e mudando a taxa de aprendizado, porém nenhuma resultou em políticas interessantes.

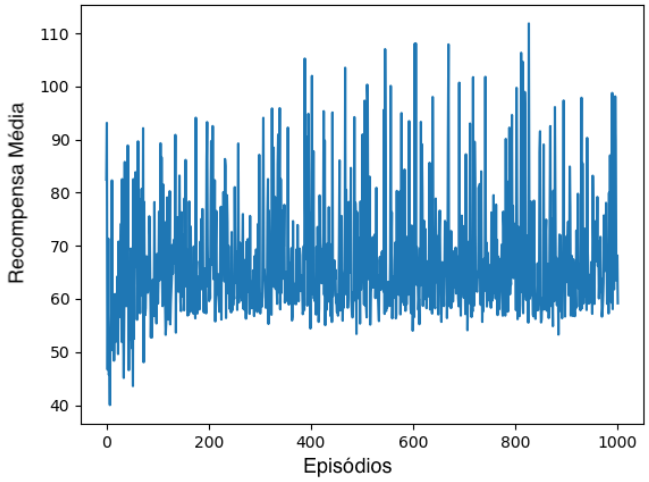

Figura 15 – Recompensa média da rede DQN. Podemos observar que o treinamento, por mais ruidoso que seja, se estabiliza em torno de uma recompensa média entre 60 e 70.

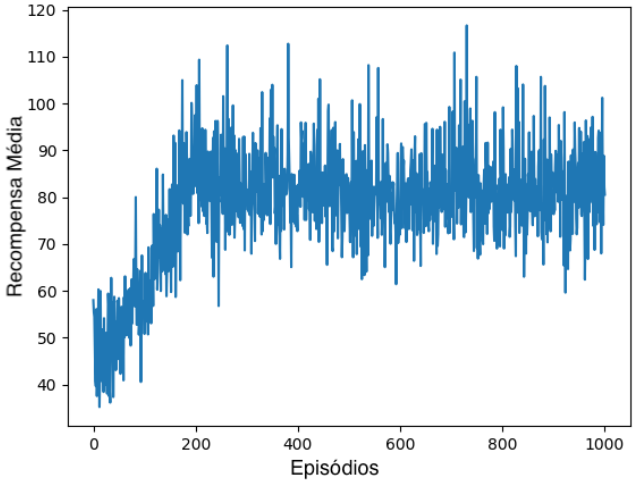

Figura 16 – Recompensa média da rede DDQN. Podemos observar que o treinamento, mesmo ruidoso, alcança uma recompensa média maior no decorrer dos episódios, em comparação com a rede DQN.

Quando comparado com o primeiro algoritmo, podemos contrastar a recompensa média por episódio, como forma de avaliar o aprendizado do agente. Podemos observar nas Figuras 15 e 16 que o segundo algoritmo tem melhor desempenho que o primeiro,

convergindo mais rápido e tendo uma recompensa média maior que a sua versão anterior ao decorrer dos episódios.

Neste caso, o agente fez 21 gols em um teste de 100 episódios, um pequeno aumento em comparação ao método antecessor.

4.3 Resultados com Proximal Policy Optimization

A implementação desse método não é tão complexa, porém são necessárias alguns cuidados com seu uso. A nossa rede não divide parâmetros entre política e valor de função, logo devemos considerar $c1 = 0$. Diversas adequações tiveram que ser feitas ao sistema, pois agora não estamos mais guardando valores de experiência, isso é, a *replay memory*, como nos métodos Deep Q-Learning.

As primeiras tentativas não renderam resultados interessantes, pois o agente, depois de vários episódios aprendendo, acabava ficava preso em um laço de repetição de ação (andando em círculos). Isso levou a mudanças importantes no sistema. A função de recompensa foi modificada, penalizando mais frequentemente estados ruins, e bonificando mais estados bons (deixando, em última análise, a função mais robusta). De forma interessante, a mesma função de recompensa não gerou resultados bons para vários treinamentos usando redes DQN e DDQN.

O treino, com a função de recompensa nova, ficou menos sensível a mudança de hiper parâmetros, obtendo resultados pouco diferentes, e ainda não ficando em regiões em que ele ficava 'preso'. Algumas políticas foram bem sucedidas, e o robô conseguia, com certa regularidade, chutar a bola pro gol, e realizar manobras para posicionar-se de forma inteligente, como observado nas Figura 18 e 19.

Apesar da Figura 17 mostrar um comportamento ruidoso, podemos ver que a recompensa média por episódio ainda aumenta.

Neste caso, o agente alcançou 71 gols em 100 episódios, a melhor marca registrada. Isso indica uma política mais consistente em relação ao objetivo final de treinamento.

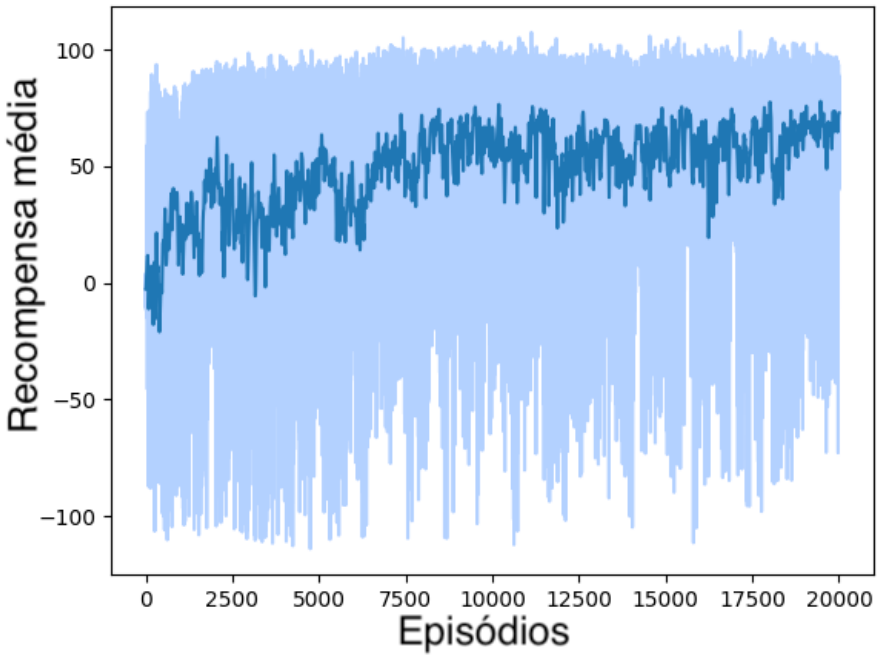

Figura 17 – Recompensas média do treinamento em função dos episódios. A função destacada (em azul escuro) representa a função de recompensa suavizada, enquanto a função em azul claro representa todos os pontos de entrada (todas as recompensas por episódio)

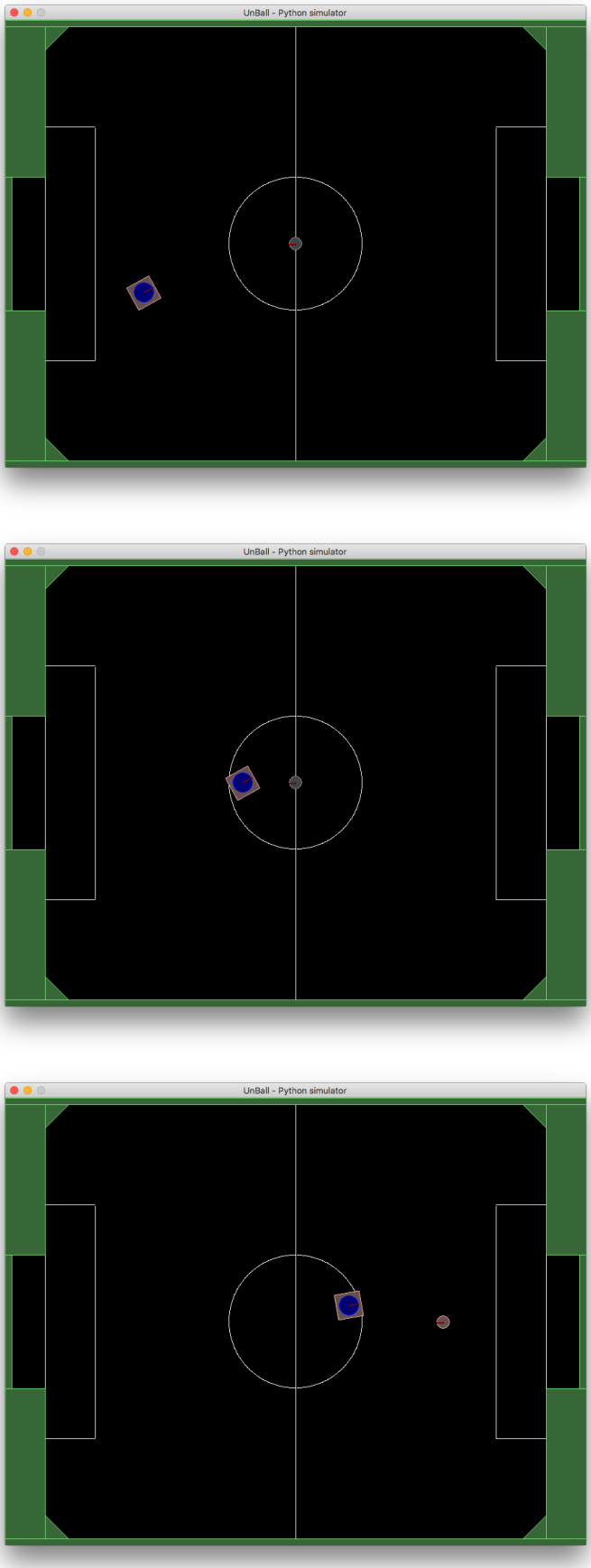

Figura 18 – Sequência do jogador chutando a bola para o gol. Observamos aqui que o agente tenta se aproximar do centro do campo, para tentar empurrar a bola no centro do gol.

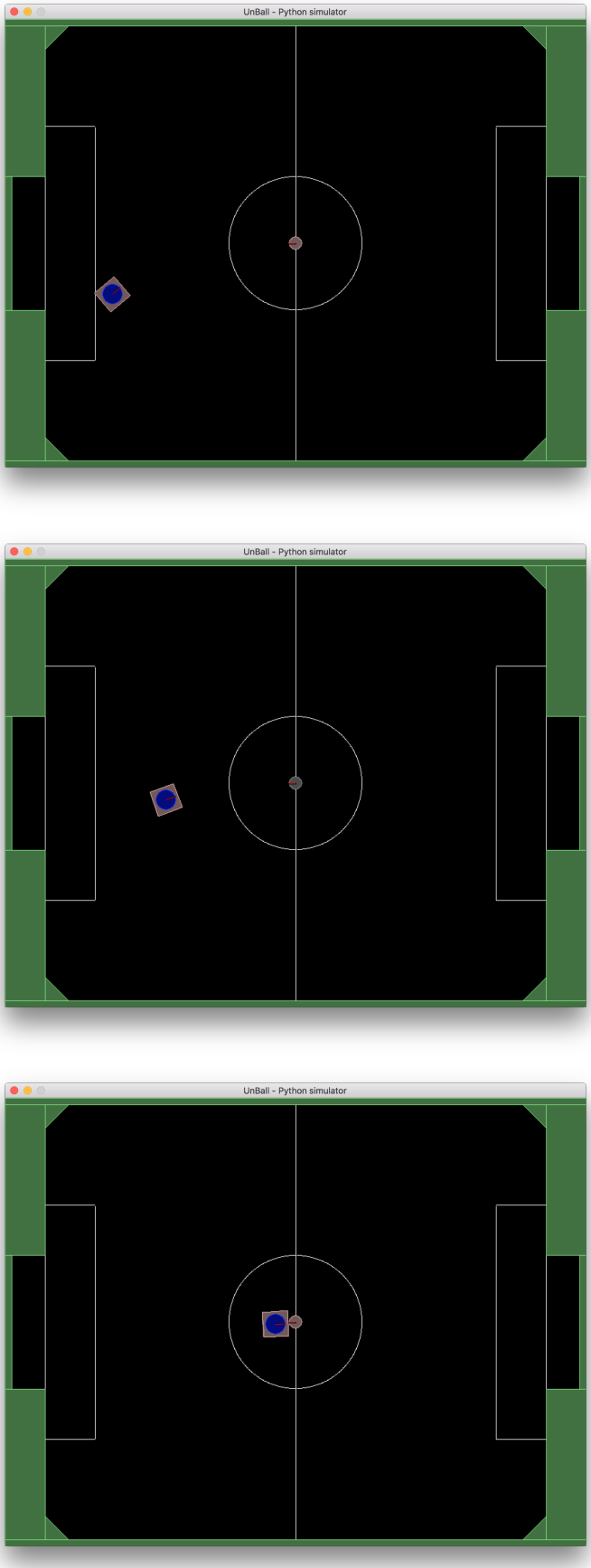

Figura 19 – Sequência do jogador chutando a bola para o gol.

5 Conclusões e Trabalhos futuros

O uso de Redes profundas são ferramentas muito poderosas para problemas de aprendizado por reforço, e já se provaram capazes de resolver problemas que até poucos anos eram considerados intratáveis.

Neste projeto, foi implementado com sucesso um ambiente adequado para o treinamento de um agente jogador de futebol. O sistema simula um ambiente definido por uma MDP, e consegue mapear recompensas associadas a estados. Diversas otimizações foram feitas nesse ambiente, como a exclusão de componentes do ROS e também a retirada da necessidade renderização do simulador. Essas mudanças resultaram em ganhos significativos de velocidade, que foram essenciais para a continuação do projeto.

Também foram implementados três algoritmos de aprendizado por reforço, e comparados os resultados entre eles. Inicialmente, com a *DQN*, observamos que o agente aprendia a responder ao ambiente em que se encontrava, porém não conseguia aprender a realizar a tarefa mais simples de tocar na bola. O segundo algoritmo, a *DDQN*, se mostrou mais eficiente, convergindo rapidamente para políticas que tinham uma recompensa média maior do que o seu antecessor. O comportamento dessa também era melhor, pois sempre se movimentava em direção a bola. O último algoritmo implementado, o *PPO*, se mostrou o mais eficaz de todos. Com esse método, o agente aprendeu a levar a bola para o gol de forma rápida e consistente, mesmo que sua posição inicial seja diferente a cada episódio.

Durante o desenvolvimento do projeto, algumas dificuldades foram encontradas. Dos desafios identificados, os tempos longos de treinamento foram o maior problema durante o desenvolvimento do trabalho. Isso fazia com que modificações no sistema, mesmo que pequenas, demorassem horas, ou até dias, para serem validadas.

Apesar dos bons resultados, existem diversas áreas que devem ser melhoradas para que o uso de técnicas de *Deep Reinforcement Learning* esteja em um nível competitivo. Primeiramente, é preciso melhorar a estrutura do sistema, de modo que ele possa processar várias instâncias de treino paralelamente. Isso aumentaria consideravelmente a velocidade de treino, e justificaria o uso extensivo de computadores de alto desempenho. Outro investimento seria em um estudo de funções de recompensa que representassem bem o ambiente. Durante o projeto, isso se mostrou como algo vital no funcionamento dos algoritmos. Por último, a extensão do ambiente - e adequação dos métodos de aprendizagem – para acomodar mais jogadores é fundamental para a continuação do trabalho. Com a inclusão dessas mudanças no sistema da UnBall, a equipe estará pronta para competir usando estratégias baseadas em aprendizado por reforço.

Referências

BORGES, R. et al. Sincronização de disparos em redes neuronais com plasticidade sináptica. *Revista Brasileira de Ensino de Física*, v. 37, p. 2310–1, 06 2015. Citado 2 vezes nas páginas 11 e 33.

FACURE, M. *Introdução às Redes Neurais Artificiais*. 2017. <https://matheusfacure.github.io/2017/03/05/ann-intro/>. Accessed: 2019-11-25. Citado 2 vezes nas páginas 11 e 34.

HASSELT, H. v.; GUEZ, A.; SILVER, D. Deep reinforcement learning with double q-learning. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI Press, 2016. (AAAI'16), p. 2094–2100. Disponível em: <http://dl.acm.org/citation.cfm?id=3016100.3016191>. Citado na página 36.

MC.AI. *Perceptron: The Artificial Neuron*. 2018. <https://mc.ai/perceptron-the-artificial-neuron/>. Accessed: 2019-11-25. Citado 2 vezes nas páginas 11 e 34.

MNIH, V. et al. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. Disponível em: <http://arxiv.org/abs/1312.5602>. Citado 2 vezes nas páginas 21 e 45.

OPENAI. *OpenAI Five*. 2018. <https://blog.openai.com/openai-five/>. Citado 2 vezes nas páginas 21 e 43.

SCHULMAN, J. et al. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. Disponível em: <http://arxiv.org/abs/1707.06347>. Citado 3 vezes nas páginas 11, 37 e 38.

SILVER, D. et al. Mastering the game of go with deep neural networks and tree search. *Nature*, v. 529, p. 484–503, 2016. Disponível em: <http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html>. Citado na página 21.

SIRLAB. *Very Small Size*. <https://sirlab.github.io/vss.html>. Accessed: 2019-11-25. Citado 2 vezes nas páginas 11 e 22.

SUTTON, R. S.; BARTO, A. G. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. Disponível em: <http://incompleteideas.net/book/the-book-2nd.html>. Citado 4 vezes nas páginas 11, 25, 26 e 36.

TESAURO, G. Temporal difference learning and td-gammon. *Commun. ACM*, ACM, New York, NY, USA, v. 38, n. 3, p. 58–68, mar. 1995. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/203330.203343>. Citado na página 31.