GOVERNO DO ESTADO DO RIO DE JANERIO

SECRETARIA DE ESTADO DE CIÊNCIA E INOVAÇÃO

FUNDAÇÃO DE APOIO À ESCOLA TÉCNICA

FACULDADE DE EDUCAÇÃO TECNOLÓGICA DO ESTADO DO

RIO DE JANEIRO FAETERJ / PETRÓPOLIS

MODELAGEM E APLICAÇÃO DE MÉTODOS DE APRENDIZADO POR REFORÇO NA CATEGORIA IEEE VERY SMALL SIZE SOCCER

LUCAS BORSATTO SIMÃO

PETRÓPOLIS

MARÇO DE 2017

APLICAÇÃO DE MÉTODOS DE APRENDIZADO POR REFORÇO NA CATEGORIA IEEE VERY SMALL SIZE SOCCER

LUCAS BORSATTO SIMÃO

Trabalho apresentado no curso de Formação em Tecnologia da Informação e Comunicação da FAETERJ - Petrópolis como requisito parcial para obtenção do grau de tecnólogo.

Orientador: Prof. D.Sc. Eduardo Krempser da Silva

PETRÓPOLIS

MARÇO DE 2017

LUCAS BORSATTO SIMÃO

CONSTRUÇÃO E UTILIZAÇÃO DE SIMULADOR PARA A CATEGORIA VERY SMALL SIZE SOCCER

Monografia de Projeto Final de Graduação sob o Título “Construção e Utilização de Simulador para a Categoria Very Small Size Soccer", defendida por Lucas Borsatto simão e aprovada em Março, em Petrópolis, Estado do Rio de Janeiro, pela banca examinadora constituída pelos professores:

_____

Prof. D.Sc. Eduardo Krempser da Silva

Orientador

FAETERJ / PETRÓPOLIS

_____

Prof.

FAETERJ / PETRÓPOLIS

_____

Prof.

FAETERJ / PETRÓPOLIS

_____

Prof.

CEFET / PETRÓPOLIS

DECLARAÇÃO DO AUTOR

Declaro, para fins de pesquisa acadêmica, didática e tecnico-científica, que o presente Trabalho de Conclusão de Curso pode ser parcial ou totalmente utilizado desde que se faça referência à fonte e aos autores.

_____

Lucas Borsatto Simão, em X de Março de 2017

AGRADECIMENTOS

Agradeço primeiramente a minha mãe e meu padrasto, pois sem sua perseverança e cuidado com meu futuro não poderia teria chegado até aqui. E meu irmão, bem, por me distrair pegando no meu pé sempre que pode.

Aos professores coordenadores, e amigos, Eduardo Krempser e Alberto Angonese, que me apoiaram durante fases importantes do meu processo de aprendizado.

Gostaria de agradecer também a meus amigos do Laboratório de Sitemas Inteligentes e Robótica, que foram um dos principais motivos do meu crescente interesse pela área que me dediquei a estudar.

Agradeço a todos os meus amigos feitos nessa jornada, por que a amizade e a companhia de vocês foram muito valiosas para mim. Eu espero mantê-las daqui em diante.

A todos da FAETERJ - Petrópolis, pois essa instituição foi de importância única na minha vida acadêmica e pessoal e, tenho certeza, fará muita diferença no meu futuro.

A todos os demais que fizeram parte e me apoiaram nessa caminhada, seja com o conhecimento compartilhado ou com a amizade.

EPÍGRAFE

"O único lugar aonde o sucesso vem antes do trabalho é no dicionário."

Albert Einstein

RESUMO

Esse documento relata a modelagem de um simulador para a categoria de futebol de robôs IEEE *Very Small Size Soccer* (VSSS) com o objetivo de ser utilizado para aplicação técnicas de aprendizado de máquina, visando melhorar a tomada de decisão nas tarefas executadas em jogos da categoria.

A utilização de aprendizagem de máquina em um simulador da categoria citada reflete a tendência atual de pesquisa na área para problemas que envolvem aprendizagem por reforço de sistemas com espaços de estados relativamente grande. Com esse objetivo, a aplicação da técnica na categoria depende de se ter uma ferramenta na qual esses algoritmos possam ser aplicados em um ambiente que realize a simulação de um jogo de forma eficiente.

Para melhor utilização de técnicas de aprendizagem, o simulador foi desenvolvido de forma a extrair todas as informações que forem relevantes a tomada de decisão de um agente, o que envolve desde a projeção de trajetória ou velocidade das rodas até extração dos pixels referentes a janela do simulador em cada instante no tempo.

Este projeto foi implementado na equipe SirSoccer da Faculdade de Educação Tecnológica do Estado do Rio de Janeiro (FAETERJ), tendo sido utilizado para criar uma base estratégica e assim esta fosse utilizada concomitante ou separadamente a plataforma de hardware do VSSS.

Para a construção do simulador foi utilizada a linguagem de programação C++ junto a biblioteca de cálculos de física Bullet Phyics. Além do uso dessa biblioteca, no projeto foi utilizada também a biblioteca Glut para executar um ambiente gráfico representante do ambiente real. E devido a necessidade de executar diversas tarefas em um curto período de tempo foi usada a classe thread, que provém da biblioteca padrão da API Posix do Linux. Para construção dos modelos de aprendizado de máquina utilizados no projeto, foi utilizada a biblioteca tiny-dnn para criação de redes neurais artificiais.

Palavras-chave: Modelagem Computacional, Machine Learning, Simulação Computacional, Redes Neurais

ABSTRACT

This document treats about the modeling of a simulator to the robots soccer category Very Small Size Soccer (VSSS) with the objective of this project be used for application with machine learning techniques, aiming increase the decision-making performance in execution of tasks on the games of this category.

For the best use of learning techniques, the simulator was developed in order to extract all the information that are relevant to decision-making of an agent. That involves from the projection of a trajectory or wheel's velocities to extraction of pixels that refers to the simulation window in each time step.

The utilization of machine learning in a simulator of the cited category reflects de current tendency of the research in this area to problems that involve reinforcement learning of systems with large state spaces. With that in mind, the application of this technique in this category depends on having a tool in which such algorithms can be applied in an environment that execute game simulation efficiently.

This project was implemented on the robotics team SirSoccer from Faculdade de Educação Tecnológica do Estado do Rio de Janeiro and have been used to create a base strategy and so this would be used together or separately with the VSSS hardware platform.

To the construction of this simulator was used the C++ programming language together with the library of physics calculus Bullet Physics. Besides the using of Bullet, was used the Glut library to execute the graphic environment that represents the real game environment. And due to the need of perform multiple tasks in the shorter period of time was used threads from the standard library API Posix from Linux.

Palavras chave: Computational modeling, Machine Learning, Computational simulation, Neural networks

LISTA DE FIGURAS

SIGLAS

VSSS	Very Small Size Soccer
MCU	Movimento Circular Uniforme
PWM	Pulse Width Modulation
PID	Proportional Integral Derivate
ODE	Open Dynamics Engine
OGRE	Object-Oriented Graphics Rendering Engine
SI	Sistema Internacional
OpenGl	Open Graphics Library
FreeGLUT	Free OpenGL Utility Toolkit
DQN	Deep Q Learning
RNA	Redes Neurais Artificiais
MLP	Multi Layer Perceptron
RNC	Redes Neurais Convolucionais
MDP	Markov Decision Process
DP	Dynamic Programming
MC	Monte Carlo
TD	Temporal Difference
RMSProp	Root Mean Square Propagation

SUMÁRIO

1. INTRODUÇÃO

A aplicação de técnicas de aprendizagem de máquina vem sendo uma tendência cada vez mais frequente em muitos sistemas de tomada de decisão, principalmente na área de robótica [36]. Isso se deve ao fato de os computadores terem adquirido uma melhora na capacidade de processamento muito grande, sendo possível melhorar algoritmos que antes demandavam um custo computacional muito grande [37].

O aprendizado é necessário partindo da premissa que, tendo um problema que envolve um número muito grande de estados que o agente pode atuar, a simples utilização de lógica booleana muitas vezes não vai cobrir todas as situações de maneira correta, abrindo assim a possibilidade para a implementação de algoritmos que aproximem as ações para cada um dos estados onde o agente atua.

No entanto, para que a aplicação de tal técnica seja viável é necessário fazer a modelagem do sistema físico envolvido, principalmente para problemas como o do VSSS que, sem esse artifício, dependem de dados provenientes de situações reais. Isso pode ser pouco prático devido ao custo computacional envolvido.

1.1. MOTIVAÇÃO

A criação de estratégias para os agentes autônomos, principalmente quando se trata de futebol de robôs, envolvem normalmente a integração de diversas técnicas diferentes que variam desde a utilização de planejamento de trajetória para se chegar a um ponto objetivo até a utilização de campos potenciais para evitar colisões. Essas técnicas são utilizadas visando fazer com que o robô realize sempre os movimentos que parecem mais promissores para um determinado instante do jogo.

Entretanto, mesmo que o desenvolvimento da estratégia seja feito de forma cuidadosa, sempre pode haver ou um estado onde as técnicas empregadas não funcionem como o esperado, ou o agente passa por um estado do jogo no qual não foi previsto anteriormente, dentre outras situações.

Para evitar a sobreposição de várias dessas técnicas para a criação de uma estratégia, pode-se utilizar algoritmos conhecidos como sendo de aprendizado por reforço para criar uma lógica de decisão mais eficiente e abrangente. Um indicativo de que essa abordagem pode ser mais efetiva é que existem categorias mais complexas, como a *Robocup 2D Soccer Simulation* [39], que já possuem equipes que implementam

aprendizado de máquina ou técnicas heurísticas para toda decisão que o agente precisa tomar [38].

Porém, o VSSS diferencia-se da simulação de futebol em 2D pelo fato de envolver a necessidade de execução em ambiente físico, o que pode impedir que as técnicas em questão encontrem soluções de maneira rápida.

O objetivo deste trabalho, então, é modelar um ambiente de simulação que seja capaz de extrair características de um jogo padrão da categoria, de forma que possa ser utilizado como plataforma para algoritmos inteligentes de tomada de decisão. Assim, uma vez que o algoritmo consiga aprender a tomar decisões pelo agente no ambiente virtual, é possível estender seu uso para também poder ser usado para agentes no ambiente real.

2. MODELAGEM COMPUTACIONAL

A modelagem computacional é uma área na qual se utiliza de forma bastante ampla conhecimentos de ciência da computação, física e da matemática. O objetivo é estudar o comportamento de sistemas complexos presentes normalmente em diferentes situações de diversas áreas da ciência através de simulações ou representações em computador.

O que torna esses sistemas complexos e de difícil resolução é o fato de eles frequentemente não seguirem um comportamento linear, e ferramentas analíticas como o cálculo diferencial são de difícil implementação em sistemas não lineares, como descrito em [41].

Um exemplo de como eventos físicos podem se tornar complicados no estudo da natureza de alguns deles esta presente na análise da termodinâmica de objetos, como visto na figura 2.1. Como descrito em [4], uma das equações mais úteis de mecânica dos fluídos, a de Navier-Stokes (2.1), pode ter uma solução analítica muitas vezes inviável devido a sua complexidade. Para esses casos, tanto a modelagem computacional quanto a utilização de técnicas de otimização para solução da equação referente a este processo podem ser uma alternativa bastante útil. Um exemplo dessa abordagem pode ser visto através de [40]. Eles são, muitas vezes, até mesmo fundamentais para a solução do problema em questão e entendimento completo do fenômeno.

$$\rho \frac{\partial \boldsymbol{v}}{\partial t} + \rho v \cdot \boldsymbol{\nabla v} + \boldsymbol{\nabla t} - \rho \boldsymbol{g} = 0 \tag{2.1}$$

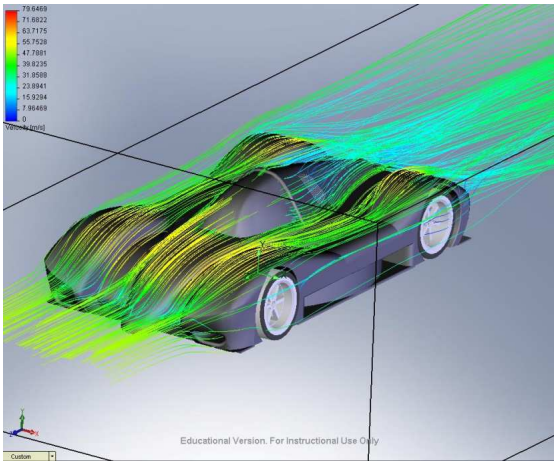


Figura 2.1 - Demonstração de simulação de um túnel de vento

O que também justifica a utilização da computação é a utilização de sistemas com muitos graus de liberdade ou muitas variáveis. Para um time de futebol de robôs pode ser relativamente fácil imaginar uma relação com este último sistema.

As variáveis que podem ser utilizadas no projeto proposto podem ir desde tamanho da roda do robô ou potência do motor até para questões de estratégia de time, como a distância requerida para evitar colisão com o adversário ou velocidade de interceptação da bola, tornando este um problema também de difícil solução sem o auxílio de um simulador em conjunto com algoritmos de otimização ou aprendizagem.

2.1 VERY SMALL SIZE SOCCER

Este projeto tem como principal objetivo a aplicação de técnicas de modelagem para a criação de uma simulação de jogos da categoria VSSS. A categoria é regulamentada pelo Instituto de Engenheiros e Eletricistas Eletrônicos (IEEE) [1], e tem características bastante similares com o futebol padrão, em que a equipe que fizer mais gols ganha a partida. Esta categoria possui regras que são baseadas na Mirosot [45], que definem as determinações que vão desde a construção do robô até os meios de captura de imagem utilizados.

A partida é composta de três robôs para cada time, cada um possuindo etiquetas coloridas de identificação em seu topo, como indicado na figura 2.2. Esses robôs são identificados por uma câmera que fica acima do campo. As imagens então são recebidas nos servidores, que se responsabiliza pelas funções de estratégia da equipe, transmissão de comandos e dados para os robôs e visualização dos dados para o usuário. O esquema descrito pode ser visto na figura 2.3.

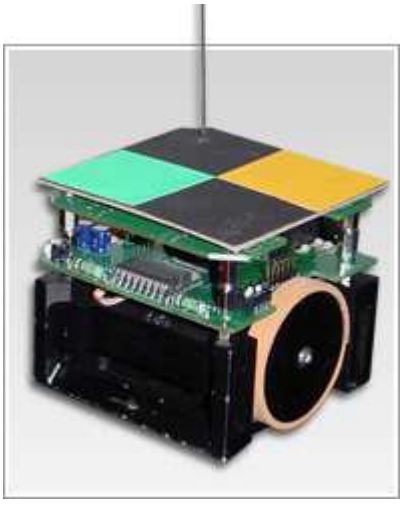


Figura 2.2 - Robô da categoria VSSS

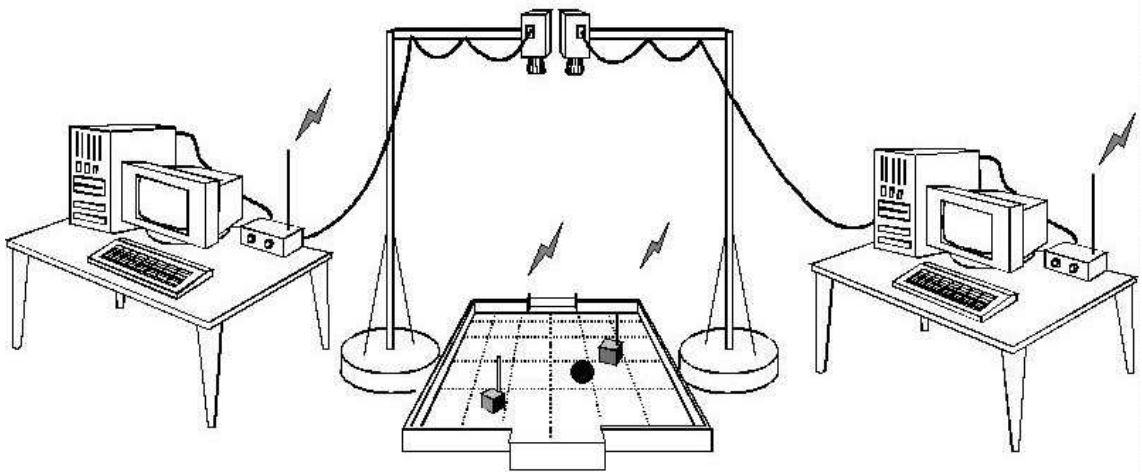


Figura 2.3 - Esquema de funcionamento da categoria VSSS

Os robôs aceitam comandos em números inteiros que determinam a potência em suas rodas. Para o motor usado pela equipe do SIRLab, esses comandos tem resolução de 1 byte, ou seja, variam entre 0 e 255, sendo estes calculados pela estratégia determinada nos servidores.

2.2 MODELOS CINEMÁTICOS E DINÂMICOS

Para a construção deste simulador foram adotados modelos cinemático e dinâmico para criação de uma física idealizada que serão explicados nesta seção. Isto ocorre porque o sistema real do VSSS não possui um sistema de referência a ser seguido, isto é, é um modelo estocástico. Assim, a preferência por um mundo físico ideal, onde não se possuem variáveis que podem mudar de ambiente para ambiente, se torna desejável para o desenvolvimento com base em um comportamento padrão para os agentes. Isso pode, de certa forma, ser visto como uma simplificação do problema, que irá servir como base sobre como o agente deveria se comportar no ambiente real.

Sendo assim, para análise do robô utilizado na categoria, podemos adotar o diagrama descrito na Figura 2.4, assim como descrito em [5].

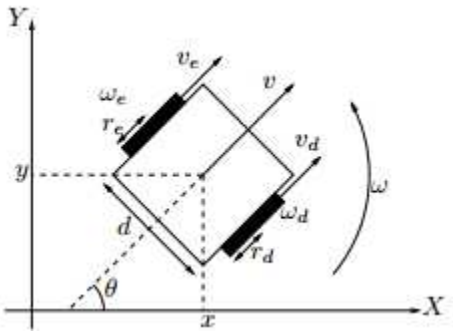


Figura 2.4 - Descrição do modelo VSSS

Assim, temos que θ é o ângulo entre o vetor de direção do robô e a abscissa, ω é a velocidade angular, v é a velocidade linear do veículo e r_e, r_d, ω_e e ω_d são os raios e as velocidades angulares das rodas respectivamente.

2.2.1 MODELO CINEMÁTICO

Definidas as variáveis descritas pela figura 2.4, podemos descrever então o modelo cinemático do robô em (2.2). Este é o modelo que define as derivadas do movimento do robô, adotando que o modelo é não holonômico já que as rodas não permitem movimentos laterais.

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}\cos\theta & 0\\ sen\,\theta & 0\\ 0 & 1\end{bmatrix}.\begin{bmatrix}v\\ \omega\end{bmatrix} \tag{2.2}$$

Porém, a descrição desta equação é apenas válida para o corpo sólido como um todo. Para que possa ser aplicado com o objetivo de controlar o robô precisamos chegar até a equação 2.5, para que possa ser usada para suas rodas. Assim pode ser feita a movimentação do corpo rígido através do centro de massa, como descrito nas equações de 2.3 a 2.5.

$$\overrightarrow{v_u} = \frac{(\overrightarrow{v_e i}+\overrightarrow{v_r i})}{2} \tag{2.3}$$

$$\omega = \overrightarrow{v_{cm}} \times \vec{R} \tag{2.4}$$

onde $\overrightarrow{v_{cm}} = \overrightarrow{v_u} + \overrightarrow{v_c}$, em que $\overrightarrow{v_c}$ é a colisão calculada pela física da biblioteca.

$$v_e = \overrightarrow{v_{cm}} - \left|\omega\frac{d}{2}\right| \;\; e \;\; v_r = \overrightarrow{v_{cm}} + \left|\omega\frac{d}{2}\right| \tag{2.5}$$

Onde os pares ($\overrightarrow{v_e'}$, $\overrightarrow{v_r'}$) e (v_e, v_r) são as velocidades das rodas esquerda e direita antes e depois da influência da colisão. O primeiro par pode ser vista como sendo uma função da velocidade das rodas, a ser computada pelo usuário e inserida através do mecanismo de impulso da biblioteca de física, a ser descrito na seção 3, enquanto o segundo é computado automaticamente através das equações em (2.5). $\overrightarrow{v_u}$ é a velocidade do corpo após a inserção dos impulsos requeridos pelo usuário através de ($\overrightarrow{v_e'}$, $\overrightarrow{v_r'}$). $\overrightarrow{v_{cm}}$ é a velocidade linear final a ser atingida no centro de massa após a colisão. Os cálculos de colisão irão ser descritos na seção 3.5. O ω é a velocidade angular a ser aplicada no mesmo ponto. $\vec{R}$ é o vetor do raio em direção ao centro instantâneo de velocidade nula, como explicado em [6]. Este último pode ser encontrado através da equação 2.6.

$$R = \frac{d}{(v_{max} - v_{min})} \cdot v_{min} + \frac{d}{2} \tag{2.6}$$

Em que d é a distância entre as rodas. v_{max} e v_{min} são as velocidades das rodas, dependendo de qual esteja em velocidade maior. Fechando assim as equações que também podem ser definidas como movimento circular uniforme (MCU).

A principal motivação em adotar o sistema que obedece o movimento circular uniforme (MCU) é a praticidade que a categoria VSSS apresenta. Por conta de adotar-se robôs que possuem apenas duas rodas, se torna simples fazer os cálculos segundo o MCU.

Para chegarmos a uma compreensão melhor da equação (2.6), pode-se assumir que qualquer movimento que o robô realizar poderá ser considerado um MCU, pois as duas rodas, que estarão sempre apontadas na mesma direção, garantem que ele obedeça a essa regra.

Isso pode ser estabelecido por conta de uma propriedade física de corpos rígidos, chamada de centro instantâneo de velocidade nula. Esta estabelece que todo ponto em um corpo rígido que tenha como referência o centro de movimentação onde a velocidade nula possui a mesma velocidade angular, como descrito em [6]. Com essas duas definições estabelecidas, através de equações básicas da dinâmica dos corpos é possível chegar até a ideia descrita pela equação (2.6).

Estabelecida a abordagem responsável pela cinemática dos corpos presentes no sistema VSSS, agora é necessário encontrar a função a qual o par ($\overrightarrow{v_e'}$, $\overrightarrow{v_r'}$) obedece, de

forma que se possa criar regras para que o robô possa chegar ao destino estabelecido pela estratégia de jogo definida pelo desenvolvedor.

2.2.2 MODELO DINÂMICO

Para que a simulação possa ocorrer conforme o modelo ideal proposto, são necessários alguns cálculos relacionados a física a qual o sistema pretende obedecer.

$$a_{cent} = \frac{{v_{lin}}^2}{R} \tag{2.7}$$

Na equação (2.7) é feita a definição da variável de aceleração a_{cent}. Este é um ponto positivo a se enfatizar relativo a uma simulação ideal, e não real, da categoria VSSS. A equação também é conhecida como aceleração centrípeta, e é responsável por manter o robô em rotação em relação ao centro de velocidade nula [6]. Para isso, normalmente é assumida também uma força de atrito com seu respectivo coeficiente, responsável pela criação de uma reação a força centrípeta, além de definir se haverá derrapagem ou não. Porém, a adoção de cálculos para derrapagem estaria contra a intenção deste projeto de se obter um padrão de movimentação do agente, de forma a facilitar o desenvolvimento de seu comportamento em campo. Sendo assim, é assumido que sempre haverá uma força de atrito de magnitude igual a gerada pela aceleração centrípeta, de forma a não haver derrapagem.

Com isso, como será discutido de forma mais profunda na seção 2.3, o padrão estabelecido não é limitante para que a simulação se mantenha confiável.

As responsáveis por definirem as velocidades linear e angular finais do robô, como discutido anteriormente, são as rodas. Para se criar um modelo apropriado, o corpo rígido atribui a velocidade dessas rodas através de impulsos. Todos os cálculos destes são convertidos para velocidade conforme a equação 2.8.

$$\overrightarrow{v_{roda}} += \frac{\overrightarrow{I}}{m} \tag{2.8}$$

Onde $\overrightarrow{v_{roda}}$ é a velocidade da roda após a inserção do impulso, $\vec{I}$ é o impulso gerado pela aceleração média do robô, medida pelo desenvolvedor, e m é a massa do robô.

Entretanto, a definição da lógica de movimentação com base em impulsos pode ser complicado para o desenvolvedor. Por isso, para finalidades práticas, esta lógica é calculada com base na velocidade final que o par ($\overrightarrow{v_e}'$, $\overrightarrow{v_r}'$), definido em 2.3, deve atingir naquele instante. Para tanto, é preciso definir a aceleração média que o motor do

robô é capaz de estabelecer, para que os impulsos atribuídos possam alcançar a velocidade requerida através da equação 2.8 e não extrapolem as medidas físicas reais. O estabelecimento dessa aceleração ajuda também a não ser necessário que se faça a modelagem do motor de forma completa, levando-se em conta a corrente fornecida ao motor, atrito, etc.

A atribuição aditiva em 2.8 representa a adição de impulsos durante os intervalos de tempo computados pela física.

Porém, para a adoção de tal sistema, outro aspecto torna-se importante para que a física se torne hábil para uma padronização da movimentação. A amortização na mudança repentina de comandos para mudança da velocidade é requerida por conta da necessidade de se manter o agente em uma trajetória circular. A amortização é feita para que, em determinado espaço de tempo, a velocidade no par ($\overrightarrow{v_e}'$, $\overrightarrow{v_r}'$) requerida pelo usuário seja alcançada de forma uniforme nas duas rodas. Caso contrário ele pode executar um movimento não previsto por conta da aplicação de um impulso excessivo naquele espaço de tempo, estabelecendo uma discrepância no valor de mudança da velocidade das rodas, causando assim a saída de sua trajetória circular padrão.

Para tanto, é necessário calcular a força exata em cada roda para que as duas cheguem na velocidade estabelecida final no mesmo tempo, realizando desta forma um movimento coordenado para que se obtenha uma mudança de velocidade suave. O código para se alcançar tal objetivo pode ser visto abaixo.

```
1       float deltaVelocidade = MAX_ACELERACAO*tempoCiclo;
2       float percAceleracao = 1.f;
3       float tempRestante = fabs(difVelocidade[rodaMaxDifVel]) / (MAX_ACELERACAO);
4
5       if (fabs(difVelocidade[rodaMaxDifVel]) < deltaVelocidade)
6           percAceleracao = fabs(difVelocidade[rodaMaxDifVel]) / deltaVelocidade;
7
8       float forcaRodaMaxDifVel = MAX_ACELERACAO*percAceleracao*massa;
9
10      if (endWheelVel[rodaMin] < iniWheelVel[rodaMaxDifVel]) forcaRodaMaxDifVel = -forcaRodaMaxDifVel;
11
12      float acRodaMaxVel = difVelocidade[rodaMinDifVel] / tempoRestante;
13
14      deltaVelocidade = fabs(acRodaMinDifVel)*tempoCiclo;
15
16      if (fabs(difVelocidade[rodaMinDifVel]) < deltaVelocidade)
17          percAceleracao = fabs(difVelocidade[rodaMinDifVel]) / deltaVelocidade;
18
19      float forcaRodaMinDifVel = acRodaMaxDifVel*percAceleration*mass;
```

Figura 2.5 - Código de cálculo da força

No código, o vetor possui a diferença entre a velocidade inicial e final de cada roda. Assim, na linha 1 é calculada a máxima velocidade que pode ser adicionada ao corpo rígido naquele ciclo. Na linha 3 calculamos o tempo restante entre essas velocidades na roda que possui a maior diferença. Na linha 6, como a velocidade adicionada no ciclo atual não pode ultrapassar o máximo definido pela aceleração, calculamos a porcentagem da aceleração que deve ser usada caso ela seja menor. Para valor maior esta é definida como 1. Na linha 8, definimos então a força que possui a maior aceleração, que deve ser atribuída a roda que possui o maior intervalo entre as velocidades. Linha 10, define pra qual sentido a força deve ser aplicada. Linha 12, é definida a aceleração para que a roda restante chegue á sua velocidade final no mesmo tempo em que a outra roda, e essa lógica serve para a definição de qual a máxima velocidade que pode ser adicionada para esta roda. Na linha 17, define-se a porcentagem que a aceleração deve seguir e, finalmente, na linha 19, é estabelecida a força para que tal roda deve utilizar. Como operação final, as forças são multiplicadas pelo intervalo de tempo computado para se obter os impulsos.

As forças calculadas serão utilizadas, então, somente para este ciclo da simulação, sendo recalculadas no ciclo seguinte para se poder obter uma variação amortizada nas velocidades linear e angular do robô.

2.3 ATRITO DAS RODAS

Como dito antes, o atrito das rodas é desconsiderado na modelagem da física por ser uma medida empírica e de difícil obtenção para diferentes ambientes. Tem vários aspectos que influenciam nesse quesito, não se tratando somente do tipo de material do qual a arena é feita, mas também do tipo de tinta, por exemplo, que ela foi pintada.

Para tanto, o simulador se utiliza de uma física idealizada para obtenção de seus modelos cinemáticos e dinâmicos. Isso ocorre de forma que, a medida que a velocidade angular da roda se modifica, a velocidade do veículo também se alterará proporcionalmente, assim como seu deslocamento. Isso é obedecido segundo a equação 2.9.

$$\Delta d = \Delta\theta * r \tag{2.9}$$

Onde $\Delta\theta$ é a rotação em ângulos sofrida pela roda durante dos ciclos de simulação seguidos, e r é o raio da roda.

O resultado obtido para Δd é exatamente o valor do deslocamento da roda na direção de seu vetor frontal. Podemos então chegar a conclusão de que $\Delta\theta$ influencia diretamente na velocidade da roda, e é assim que são feitos os deslocamentos na simulação.

A escolha pela desconsideração do atrito pode ser justificada através da análise da natureza física do modelo em ambiente real. Isso porque, enquanto o robô executa comandos que fazem seu torque nas rodas manterem sua aceleração, nada muda contanto que a força de atrito sempre seja maior que a força empregada pelo torque, de forma que não haja a derrapagem. Esse modelo pode ser visto na figura 2.6, onde a força empregada pela roda contra o chão provoca a reação contrária de uma força no sentido oposto. Elas estão em vermelho e azul, respectivamente, como explicado em [6].

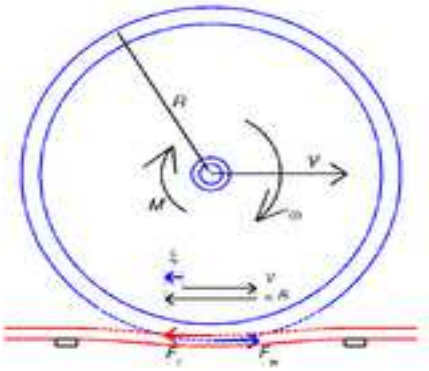


Figura 2.6 - Demonstração de atuação de atrito nas rodas

Porém, a força de atrito não causa somente a reação ao deslocamento, ela também causa um torque contrário ao movimento vigente no centro do corpo e, portanto, contrário ao M identificado na figura 2.6. Isso ocorre devido ao atrito interno dos mecanismos que fazem a transferência do torque para as rodas. Durante o processo de execução dos comandos, então, o torque gerado por esse atrito impedirá a progressão uniforme da movimentação do agente.

Esse processo é difícil de ser modelado em uma simulação devido aos diferentes componentes de um robô. Por isso a escolha por um ambiente físico idealizado se mostra acertada. Dessa forma, o responsável por manter o modelo estável e de acordo com o que é computado pela simulação é o controle, como será explicado mais adiante na seção 2.5.

2.4 FUNÇÃO DA VELOCIDADE

A função que define a função da velocidade nas rodas, presente na equação 2.5, é esta demonstrada em 2.10. Esta funcionará como lógica para estabelecer a velocidade requerida para uma das rodas, enquanto a outra receberá a velocidade máxima permitida. Isso irá permitir ao agente girar na direção desejada. Apesar de existirem outras formas que possam representar a função da velocidade, esta equação foi elaborada com o objetivo de o robô alcançar o ponto objetivo o mais rápido que seus comandos permitissem, sendo essa a função do argumento v_{max}.

$$v = v_{max}\left(1 - \frac{\theta}{\beta}\right) \tag{2.10}$$

Onde v_{max} representa o comando máximo que os motores do robô podem assumir, θ é o ângulo entre o vetor de direção do robô e o vetor unitário para o ponto objetivo e β é o ângulo a partir do qual o robô passa a utilizar esta função ao invés de rodar em seu eixo para entrar na faixa de ângulos menor que β. A representação desta função pode ser vista na figura 2.7.

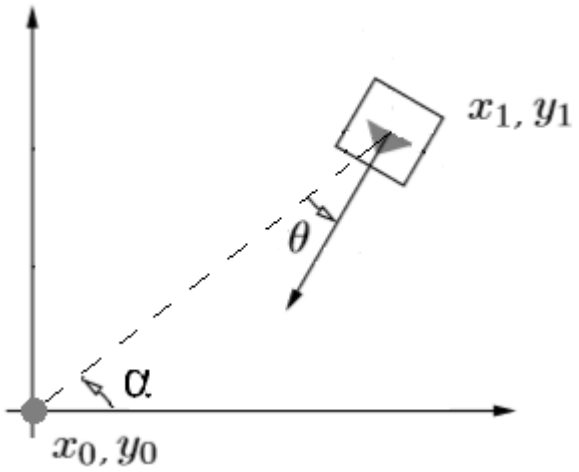


Figura 2.7 - Parâmetros para a função da velocidade

Esta função, como dito anteriormente, é então utilizada para determinar a velocidade de uma das rodas enquanto a outra assume o valor da velocidade máxima do robô, fazendo assim com que este percorra a circunferência necessária para seguir o ponto objetivo. Esta é uma das formas pelas quais o par ($\overrightarrow{v_e}'$, $\overrightarrow{v_r}'$) em 2.3 pode ser estabelecido.

2.5 A TEORIA DO CONTROLE

Para que possamos calcular eventos físicos no mundo real em modelos como os cinemático e dinâmico descritos, apenas precisamos da velocidade das duas rodas para que a física possa agir de acordo com o a natureza física inerente ao corpo rígido do agente em simulação.

Mas para um planejamento de trajetória em ambiente físico precisamos ir na contra mão desta ideia, pois precisamos partir de uma posição ou pose e calcular quais as velocidades que as rodas devem assumir para atingir o ponto objetivo. O problema é que, no mundo real, essas duas características, velocidade e distância, são medidas diferentes, ao contrário do ambiente simulado. Se levarmos em conta que a captura do estado do campo é feita por câmera, por exemplo, então medidas de pose serão dadas em pixels e graus, enquanto que as velocidades para o robô são dadas em PWM (Pulse Width Modulation), que não possui medida exata a qual possa ser relacionada de forma unidade padrão de velocidade, metros por segundo. Além disso, o PWM pode variar de motor para motor. Para resolver esse problema e conseguirmos definir a velocidade das rodas de forma que a estratégia desenvolvida em simulação se aproxime o máximo possível ao apresentado em ambiente real, precisamos recorrer á teoria de controle.

De acordo com [9], o controle é uma área da mecatrônica que trata da análise da dinâmica de todo sistema que pode ser controlado, reproduzindo os efeitos desejados.

Esta é uma parte muito importante para que o que ocorra na simulação e no mundo real aconteça de forma semelhante. Isso por que se um bom controle não for implementado, o comportamento desenvolvido para o agente na simulação pode não ser seguido em quase nada pelo que é calculado no mundo real.

O fluxograma básico para o uso de um sistema de controle pode ser visto na figura 2.8, o que é chamado de sistema de malha fechada. Podemos ver que a primeira etapa é estabelecer um valor objetivo, ou *set point*. Para o caso do futebol de robôs, podemos dizer que este seja o ângulo de zero graus do robô em relação a bola. Isso então é passado para a etapa do controlador, que decidirá quais correções no comando final devem ser feitas para se alcançar este valor objetivo. Feito isso chegamos a etapa de processo, que significa que será feita a execução do comando definido pelo controlador. O sensor serve para verificar qual foi o efeito da saída do processo no ambiente analisado. No caso do VSSS, o sensor seria a câmera, que avaliará qual foi a

efetividade do comando PWM executada pelo robô, e retornará uma imagem a ser processada pelo servidor para que o comando seja corrigido novamente.

Como o objetivo principal deste trabalho foi a implementação de técnicas de aprendizado no ambiente VSSS, o controlador escolhido para o sistema de simulação foi o discutido na seção 2.4. Entretanto, após testes realizados na plataforma, foi concluído que o controle PID teria melhor adaptação para o caso de interesse de aproveitamento do código desenvolvido na simulação para o ambiente real. Isto se dá pelo fato de que a função discutida na seção 2.4 não envolve correção de erro. Em ambiente simulado esse requisito não é necessário, pois se trata de um comportamento ideal. Porém, no ambiente real há diversas variáveis que não o permitem agir da mesma forma.

Neste trabalho, no entanto, este tema não é discutido profundamente, sendo necessária a discussão futura de implementação de um controlador que se adapte aos dois ambientes.

O controlador PID é preferencial pela simplicidade de aplicação e eficiência no tratamento de erros, este tipo de controle é indicado para a maior parte dos problemas. Este não é um controlador ideal, pois não é variável com o tempo, o que seria útil devido ao sistema apresentar erros diferentes segundo o decaimento da bateria do robô, por exemplo. Isso pode prejudicar de forma significativa a performance desenvolvida em simulação. Outros tipos de controle podem ser vistos através de [42].

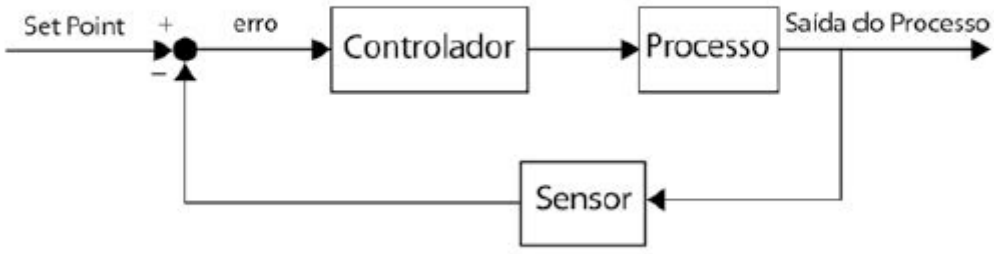


Figura 2.8 - Fluxograma de atuação do controle em um processo

3. BIBLIOTECAS E SOFTWARES DE FÍSICA

Há hoje disponíveis para os desenvolvedores diversos motores de física, os quais foram criados com objetivos principais distintos, mas que podem ainda assim serem utilizados para finalidades diversas. Em [43] é feita uma comparação entre os motores MuJoCo, Havok, Bullet, Physx e ODE, podendo se chegar a conclusão semelhante sobre sua eficiência.

MuJoCo é um motor de física feito para simular robôs com articulações, sendo baseado não em plano cartesiano, mas em um espaço articulado. Apesar de conseguir realizar muito bem os testes definidos no trabalho referenciado, ele falha quando se tem múltiplos corpos rígidos para gerenciar, além de não ser de código aberto.

Physx foi desenvolvido pela NVIDIA e tem opção de maior performance com suas placas de vídeo, devido a empresa ser hoje uma das líderes nesse setor. Apesar de vários jogos terem sido desenvolvidos com esse motor, podemos ver que apesar de sua performance e estabilidade, ela pode muitas vezes deixar a desejar na exatidão da física criada. Geralmente deixa de obedecer até mesmo leis básicas de forma muito acentuada, como a manutenção da quantidade de energia.

O Havok, assim como Physx, é destinada quase que exclusivamente para o desenvolvimento de jogos. Este motor possui um desempenho melhor para leis da física e tem desempenho regular quando se aumenta seu intervalo de execução da física entre uma atualização e outra do cenário.

O ODE (Open Dynamics Engine) e Bullet são ambos de código aberto, mas com algumas diferenças em suas abordagens mais promissoras. O ODE tem uma performance maior quando se aumenta sua velocidade em relação ao tempo real , que é o intervalo de execução da física. Já a Bullet, apesar de possuir menor eficiência neste quesito, possui uma qualidade muito boa em seus cálculos de colisão.

Há também softwares de simulação de física ou desenvolvimento de jogos que incluem esse requisito, como Maya, Blender ou Unity. Porém todos dependem de suas respectivas plataformas para que possam executar as mecânicas, e o objetivo do simulador deste projeto é fazer um processamento mais rápido, o que impede a utilização destas.

3.1 UTILIZAÇÃO DA BULLET

A Bullet Physics [14] é uma biblioteca programada em C++ amplamente utilizada principalmente, por conta de seu alto desempenho e precisão. Criada em 2003 por Erwin Coumans, a Bullet há anos vem ganhando reconhecimento, inclusive tendo sido premiada [44], por conta de sua alta performance para simulação de grandes colisões. Há diversas plataformas de desenvolvimento de ambientes físicos que se utilizam da Bullet para ampliar sua performance, como os conhecidos Maya, OGRE e Blender, além de já ter sido usada em diversos filmes e jogos, como 2012, Sherlock Holmes e Gran Theft Auto V.

A física deste trabalho foi implementada com base nesta biblioteca, principalmente por conta de sua reconhecida confiabilidade em ambientes físicos e facilidade de implementação, principalmente relacionada a parte de veículos. Além disso, ela possui uma comunidade de desenvolvedores grande, o que facilita o suporte e a ajuda no desenvolvimento.

Outras opções de desenvolvimento também poderiam ter sido adotadas, como os já citados Blender ou Maya, mas a opção pela utilização da Bullet também se justifica pela oportunidade de aprendizado e implementação matemática, que possibilitou maior possibilidade customização do código fonte. Isso facilitou a implementação da física desejada para o simulador, uma vez que a Bullet já trabalha com uma classe de criação de veículos pronta.

Quanto ao funcionamento interno da biblioteca, primeiramente é preciso entender que a Bullet trabalha com impulsos em sua maior parte para realizar a sua dinâmica. O impulso tem a unidade de Newton segundo, o que possibilita tanto cálculos tanto de ação e reação através da força quanto a transformação desta para a velocidade final do corpo rígido. A execução dos impulsos no mundo físico da biblioteca é muito simples, e é feito apenas fornecendo a biblioteca o intervalo de tempo que se deseja que a física avance. Esses intervalos definem o instante físico a ser calculado pela biblioteca. A física desta é, então, realizada através da execução em sequência desses instantes. Essa parte e a de criação de corpos rígidos podem ser vistos em [7].

Quanto a descrição das classes que envolvem a Bullet, algumas de maior importância dentro da biblioteca merecem destaque. Existe uma classe básica para criação de objetos, chamada *btRigidBody*, que é herdada por qualquer corpo rígido que seja criado, inclusive os veículos utilizados como robôs na simulação. Todas as

propriedades da classe de corpos rígidos pode ser vista no Anexo A. Cada um destes corpos pode possuir uma forma diferente, e todas as formas possuem como classe pai a *btCollisionShape*. Os veículos são feitos através da criação de formas compostas, chamadas de *btCompoundShape*, permitindo que se unam todas as partes do robô, sendo estas chassi e rodas. A classe utilizada para criação deles chama-se *btRaycastVehicle*.

A biblioteca trabalha através de ciclos que determinam os cálculos para diversos eventos físicos que envolvem toda a parte de corpos rígidos, incluindo sua classe de veículos. Como exemplo de tais eventos temos atrito, impulsos, gravidade e colisões. Para tanto, este ciclo obedece a uma ordem que permite que a Bullet mantenha sua precisão e eficiência, o que é descrito na figura 3.1.

No primeiro módulo, há a verificação de colisões entre corpos rígidos. Isso influenciará diretamente em seu vetor de velocidade resultante, já que a Bullet se utiliza de incrementos na variável de velocidade presente dentro desta classe. Este é o primeiro dos dois momentos em que algoritmos da biblioteca incrementam a velocidade diretamente no objeto.

Na segunda etapa do diagrama há o cálculo de suspensão, reservado somente á parte de veículos da Bullet. A suspensão servirá para criar uma força de reação do veículo contra a força da gravidade, principalmente. Além disso, choques podem influenciar nas forças de reação também, visto que criam velocidades angulares em X e Z.

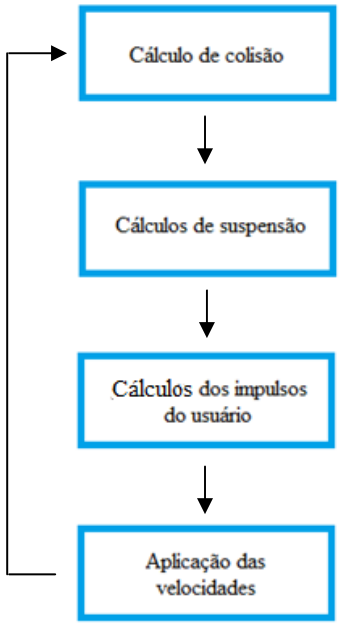


Figura 3.1 - Fluxograma do ciclo da Bullet para um veículo

A Bullet permite ao usuário inserir no corpo rígido a aplicação de impulsos, o que é calculado no terceiro bloco em 3.1. Isto na classe de veículos é feita através da aplicação de força nas rodas, que são implementadas através de impulsos e que, posteriormente, são convertidos em velocidade.

A última etapa refere-se ao processo da aplicação das velocidades através dos impulsos fornecidos pelo usuários e calculados em forma de velocidades angular e linear no centro de massa do corpo rígido do veículo através da terceira etapa.

A dificuldade maior para a implementação deste motor de física foi justamente com a dinâmica dos robôs - implementada através da citada classe de veículos. Por conta de algumas opções de desenvolvimento e modelos cinemáticos e dinâmicos desejados, a modificação desta classe foi necessária. Isso fez com que o simulador pudesse alcançar um patamar de física simples e exata para comparação com o ambiente real. Essas mudanças serão relatadas a seguir.

3.2 MUDANÇA DE ESCALA

Um problema que a Bullet apresentou durante o desenvolvimento foi a pouca confiabilidade em ambientes físicos menores. A biblioteca, para que possa fazer o cálculo de suas colisões, se utiliza de uma margem de 0.05 na unidade utilizada pelo usuário. Isto é, se o usuário esta usando o Sistema Internacional de Unidades (SI), a Bullet precisará de uma margem de 0.05 metros para fazer os cálculos de física das colisões. Isto quer dizer que ela usará esta como a distância mínima que pode haver entre dois objetos.

O problema com essa margem esta no fato de que os robôs da categoria medem, em média, cerca de 8 centímetros. Isso impede que uma física apurada seja calculada para o simulador, visto que essa margem precisa ser utilizada e, com isso, cria-se uma distância mínima entre a bola e o robô de 5 centímetros.

Para solucionar esse problema, basta aumentar a escala e a Bullet tratará de manter a física funcionando corretamente. Assim, ao invés de ser utilizado o sistema em metros, foi utilizado o sistema em centímetros, sendo cada unidade de distância equivalente a um centímetro na simulação. Para ilustrar esses resultados foram utilizadas medidas arbitrárias para demonstração. O resultado é visto através da comparação da figura 3.2.

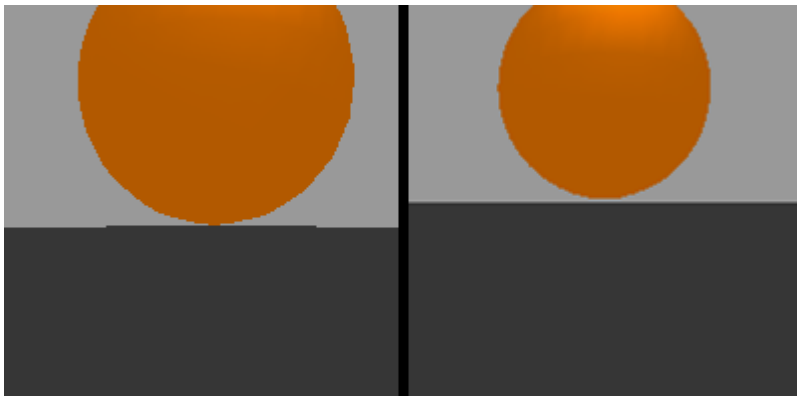


Figura 3.2 - Esquema mostrando a diferença entre adotar escala em centímetros, a esquerda, e em metros a direita

3.3 MIGRAÇÃO PARA O SISTEMA MCU

A adaptação da biblioteca, para que envolvesse os cálculos dos modelos mencionados na seção 2, teve como principal motivação a falta de suporte por conta da Bullet em executar a física requerida pelo MCU. Ela usa outros métodos que envolvem cálculos de matrizes jacobianas, que somente são adaptáveis, na forma como foram implementadas na biblioteca, para simulação de veículos que possuam duas rodas dianteiras e duas traseiras, como descrito em [8], mas não são tão eficientes quando executados em um robô do VSSS.

Essa afirmação se torna verdadeira verificando testes feitos simulando um movimento que deveria apresentar a movimentação feita em MCU, mas que não apresentou o comportamento adequado. Primeiramente, na simulação foram introduzidos impulsos constantes nas rodas do robô, de 0 N/s na esquerda e 2 N/s na direita. A manutenção da adição desses valores influenciando diretamente na velocidade do veículo a cada ciclo da Bullet deveria fazer com que o mesmo realizasse um movimento circular. Mas, como podemos ver na figura 3.3, não é isso o que acontece. Assim, de acordo com o resultado, a matriz jacobiana se mostra ineficaz para este caso.

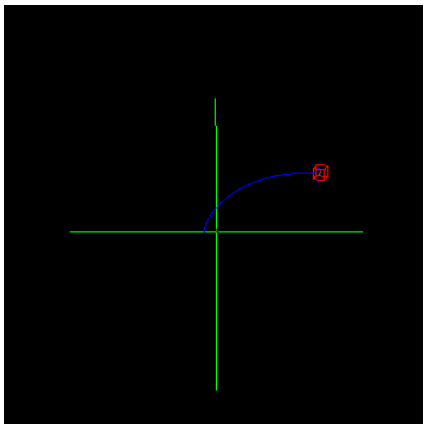


Figura 3.3 - Movimentação com utilização de matriz jacobiana

Já substituindo a matriz jacobiana por um algoritmo que represente os modelos cinemáticos e dinâmicos mostrados na seção 2, o resultado obtido é mostrado pela figura 3.4. Em ambos os casos, a linha azul representa a trajetória do veículo e as linha vermelhas representam sua forma. Já as linhas verdes representam o ponto de origem das coordenadas.

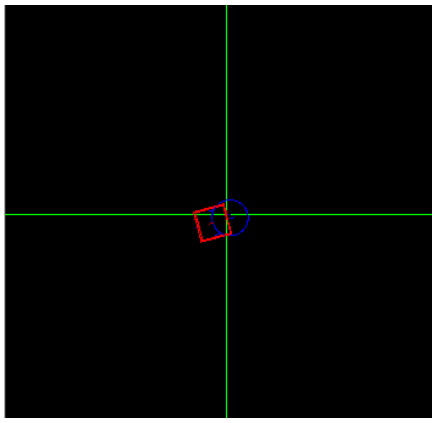


Figura 3.4 - Movimentação com utilização de MCU

Para que tal mudança fosse possível, foi necessário desenvolver um novo sistema de coordenadas para os robôs. Isto foi necessário por que o padrão dentro da Bullet é que as coordenadas globais sejam atribuídas ao corpo rígido. Desta forma todo impulso imposto pelo desenvolvedor é aplicado de forma incremental á velocidade já existente naquele corpo. Porém esse não é um modelo satisfatório para que seja usado para simular a ação do MCU.

Devido a forma da equação 2.7, que não é linear, a adição de vetores de velocidade a cada ciclo se torna impossível de ser mantida. A aceleração centrípeta atua

em um corpo de forma integral, utilizando o valor total da velocidade linear naquele momento. Se a adição incremental fosse mantida, a força centrípeta prevaleceria na interação física e puxaria o robô para o centro do centro instantâneo de velocidade nula. Ou, de outra forma, se somente o dv referente àquele instante fosse adicionado ao somatório a equação 2.7 não estaria satisfeita, uma vez que o somatório dos componentes quadráticos de um somatório seria diferente do módulo da velocidade ao quadrado.

Em outras palavras, a função da força centrípeta é alterar momentaneamente o vetor de velocidade linear do robô. Porém como a Bullet somente incrementa o impulso na velocidade em parcelas, de acordo com os instante físico que esta sendo computado, a componente do vetor de aceleração referente a força centrípeta atuaria de forma desproporcional no corpo rígido, já que em todo ciclo seria aplicado um impulso equivalente ao seu valor total, que é diretamente proporcional a $v_{lin}{}^2$.

Para corrigir este erro foi necessária a criação de coordenadas locais para o robô, de forma que os impulsos implementados pelo desenvolvedor primeiro sejam acrescentados á velocidade da roda, ou seja, serão sempre unidirecionais por conta do formato do corpo rígido. Depois o valor da velocidade linear do veículo seja substituída pela nova causada pela ação das rodas.

Assim, a cada ciclo a Bullet passa a incrementar um valor real para a velocidade da roda. Para o cálculo de velocidades linear e angular e aceleração centrípeta são utilizadas as equações 2.4, 2.5 e 2.7. A aplicação desses cálculos pode ser visto no algoritmo a seguir.

Algoritmo de aplicação de velocidade em coordenadas locais

```
dvVelocidadeLinear[0] = impulso[0] * inversoMassa
dvVelocidadeLinear[1] = impulso[1] * inversoMassa

velRodaEsquerda += dvVelocidadeLinear[0]
velRodaDireita += dvVelocidadeLinear[1]

velLinearCentroMassa = vetorUnitFrontal * (velRodaEsquerda + velRodaDireita)/2

velAngularCentroMassa = velLinearCentroMassa x raioCentroVelNula

aceleracaoCentripeta = exp(velLinearCentroMassa, 2) / raioCentroVelNula

velLinearCentroMassa += dt*aceleracaoCentripeta*vecDirecaoCentroVelNula
```

As duas primeiras linhas são responsáveis por converter a unidade de impulso para velocidade, alocando cada uma em posições de vetores referentes ás rodas. As duas subsequentes irão acrescentar a velocidade calculada dv ao valor da velocidade já existentes nas rodas. Então é atribuída a velocidade linear de acordo com o $\overrightarrow{v_{cm}}$ da equação 2.5 e, assim, é multiplicado o vetor unitário frontal, para que se possa converter o valor para coordenadas globais referentes ao mundo físico. Feito isso, é calculado também a velocidade angular de acordo com a equação 2.4, a aceleração centrípeta segundo a equação 2.7 e, finalmente, esta é acrescida a tangente do vetor da velocidade linear do robô.

Com a utilização do algoritmo acima, podemos garantir que toda a física será respeitada de acordo com o que foi descrito na seção de Modelos Cinemáticos e Dinâmicos.

3.4 ROTAÇÕES SOBRE OS EIXOS X E Z

Como descrito anteriormente nesta seção, foram identificados alguns erros que impediram a aplicação dos cálculos presentes na biblioteca para o modelo cinemático proposto. O problema de definição de valores para a constante elástica e comprimento adequado da suspensão do veículo se encontra entre eles.

A classe de veículos da Bullet possui implementações cálculos automáticos para definição de impulsos para as suspensões devido a gravidade e a colisões. Porém, as suspensões possuem um limite de comprimento em que podem agir. O que ocorre é que muitas vezes esse comprimento é superado, podendo ou ficar negativo ou acima do comprimento máximo permitido dependendo do sentido que a colisão impõe a velocidade angular do corpo rígido. Isso ocorre devido ao fato de esses eventos com colisões em velocidade alta resultarem em um valor alto nas componentes X e Z da velocidade angular. Porém, a biblioteca não trata esses erros. Isso ocasiona em erros que fazem a simulação não respeitar leis básicas da física como a da conservação de energia, como a aplicação de velocidades no corpo rígido desproporcionais ás forças de reação em colisões.

Além disso, o robô usado para a categoria VSSS é muito robusto e não possui suspensão alguma. Porém, a classe de veículos da Bullet necessita de valores para a coeficiente elástica da suspensão, pois não permite entrada de valor zero. Definições de

valores muito baixos ou muito altos para o coeficiente costumam ocasionar em comportamentos imprevistos na física do agente.

Para sanar o problema de forma geral e evitar que outras partes da física apresentassem o mesmo erro houve uma tentativa de evitar que a física realizasse seu ciclo para o veículo se a lei da conservação da energia fosse violada, da forma apresentada na equação 3.1. Essa não é a forma mais adequada de se atacar o problema, porém a Bullet possui um código extenso e de complexo entendimento, e é necessário que se garanta que nenhum algoritmo que seja referente aos cálculos da física do agente viole a lei.

$$Ec_i + Ep_i = Ec_f + Ep_f \tag{3.1}$$

Onde Ec_i e Ec_f são as energias cinéticas inicial e final do sistema e Ep_i e Ep_f são as energias potenciais inicial e final do sistema, respectivamente.

Apesar da tentativa de se tratar o problema através da própria lei da conservação de energia, os erros ainda permanecem. Apesar de o principal causador do problema ser basicamente o comprimento da suspensão, não há meios de retirá-la por completo sem a alteração da física que mantém o veículo suspenso.

Por conta disso, a retirada das componentes X e Z da velocidade angular dentro do ambiente de simulação se mostrou uma decisão certa, pois impede que essas parcelas influenciem na suspensão do agente de forma desproporcional e não causa tantas influências na simulação por conta das leis estabelecidas na seção 2. Essa solução, apesar de ainda não ser a melhor, não retira a confiabilidade da simulação. Isso por que os robôs, como dito, dificilmente se chocam com força suficiente para se virarem, e estas situações são difíceis de serem previstas.

3.5 MODELO DE COLISÃO

No mundo real, o futebol de robôs tendo como o modelo os agentes do VSSS, em forma de cubo, possuem uma facilidade em sua física de colisão, além de serem bastante estáveis e dificilmente um robô é derrubado após um choque. O modelo de colisão empregado no simulador foi feito de forma que todo choque identificado pela biblioteca Bullet é calculado de forma a obedecer a lei física da ação e reação.

O que ocorre é que, pelo fato de praticamente toda a física envolvendo os robôs ter sido refeita para adaptar-se ao seu modelo cinemático, adaptações da física da Bullet

tiveram de ser feitas de forma a considerar as velocidades empregadas diretamente da classe de veículo da biblioteca.

Isso ocorre devido ao fato de a Bullet possuir dois algoritmos que são responsáveis pela aplicação da dinâmica dos impulsos para os veículos, e estes acionados em momentos diferentes dentro de seu código. Primeiramente, ocorre o cálculo de colisão, que definirá os impulsos a serem aplicados aos corpos rígidos para reagirem ao impacto proveniente de um impulso aplicado no último ciclo. Em um segundo momento são aplicados os impulsos calculados pelo usuário, que novamente afetarão na dinâmica do robô, como já descrito no fluxograma da figura 3.1.

Acontece que o modelo descrito na seção 2 não são os obedecidos pela Bullet, como dito anteriormente nesta seção. Dessa forma, ocorre a necessidade de se calcular toda a física que envolve o corpo rígido do veículo na própria classe. Isso faz necessário a manutenção do efeito causado pelos algoritmos de dinâmica da Bullet descritos, de forma que seus efeitos possam ser usados no cálculo final do impulso do robô, tornando a física mais realista.

Para que possamos entender o modelo de colisão, seguiremos o modelo mostrado na figura 3.5. As forças aplicadas por cada corpo servem também como forças de ação e reação no corpo oposto, e são determinados pelas setas vermelha e azul. Neste caso, tiraremos como exemplo o robô de etiqueta azul, e a demonstração das forças aplicadas neste corpo são vistas na figura 3.5. O objetivo então se torna o deslocamento das forças para o centro de massa do robô A, o que pode ser visto na figura 3.6, em que possuímos então um Fr representando a força resultante. Posteriormente esses impulsos irão influenciar na velocidade da roda do robô.

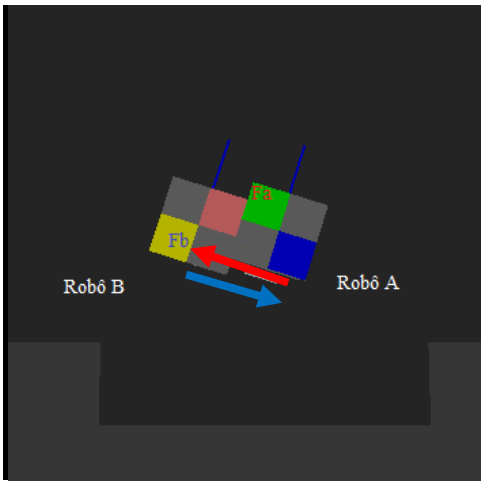


Figura 3.5 - Forças de ação e reação atuando em dois robôs

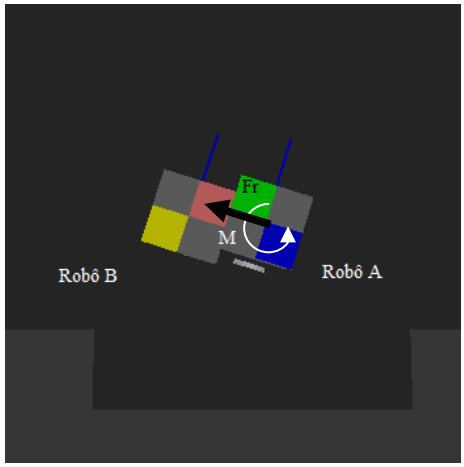


Figura 3.6 - Forças de ação e reação atuando no centro de massa do robô A

Assim, podemos calcular o vetor de direção das forças, que pode ser adquirido através da própria Bullet. Esse processo irá envolver o uso das forças calculadas pela biblioteca para convertê-las ao novo modelo e, após seu uso, a realização da anulação destas para que não influencie na dinâmica do corpo. Para fazermos essa conversão, podemos transferir as forças para o centro de massa, podemos utilizar a equação 3.2 para gira-las de forma que fiquem alinhadas com as coordenadas locais do robô. Feito isso, precisamos apenas transferir seu módulo para o centro e criarmos a rotação a partir da distância do ponto de colisão até o centro de massa do robô, o que já é feito pela Bullet.

$$\overrightarrow{v_c} = |v| * cos\,\theta = |v_i| * \overrightarrow{v_i} \cdot \overrightarrow{f_r} \tag{3.2}$$

Onde $\overrightarrow{v_c}$ é o vetor de velocidade da colisão adicionado ao centro de massa. $|v|$ é o módulo da velocidade aplicada no centro de massa, que é essencialmente igual á $|v_i|$. Já $|v_i|$ é o módulo da velocidade produzido pela Bullet na colisão. $\overrightarrow{v_i}$ é o vetor unitário da velocidade produzida pela colisão, cujo módulo é $|v_i|$. E $\overrightarrow{f_r}$ é o vetor unitário que descreve a frente do robô.

Concluídos esses cálculos, é preciso aplicar $\vec{v}$ na velocidade das rodas, de forma que estas tenham uma desaceleração devido aos impactos. Caso isso não seja feito, as rodas continuariam fazendo com que o robô mantivesse sua rota, por que somente elas irão influenciar na velocidade linear e angular finais do corpo rígido de acordo com o modelo implementado. A partir deste momento, qualquer velocidade aplicada através de impulsos no robô será adicionada á velocidade total, sendo esse processo repetido a cada ciclo. A equação 3.3 que define esse processo.

$$\overrightarrow{v_n} = \overrightarrow{v_a} + \overrightarrow{v_c} + \overrightarrow{v_u} \quad (3.3)$$

Onde $\overrightarrow{v_n}$ é a nova velocidade atribuída ao corpo rígido e $\overrightarrow{v_a}$ é a velocidade anterior do corpo rígido. $\overrightarrow{v_c}$ é a velocidade causada pelo impulso da colisão calculada na equação 3.3 e $\overrightarrow{v_u}$ é a velocidade proveniente do impulso calculado e implementado pelo usuário calculado na equação 2.3. Essa equação teve desenvolvimento prévio através da demonstrada em 2.4.

A equação vai permitir então que a velocidade nas rodas possa ser calculada, e assim é atribuída ao final do processo a velocidade ao corpo rígido, fazendo com que os modelos cinemáticos e dinâmicos descritos sejam respeitados e a física seja calculada de maneira eficiente.

3.6 COORDENADAS E DIREÇÕES RELATIVAS

O ambiente modelado de acordo com a física implementada pode apresentar um problema quando necessário fazer, por exemplo, a comparação entre duas implementações de estratégia diferentes. Isso acontece por que, como as coordenadas retiradas da biblioteca de física são todas absolutas em relação a um único ponto de origem, os robôs, a princípio, possuem pontos de referência diferentes em campo.

Por exemplo, levando-se em consideração que o campo mede 640 x 480 pixels. A referência de gol de um time seria quando x é 0 e a do outro quando x é 640. Isso afeta negativamente a lógica que envolve o desenvolvimento de estratégias por que, quando em etapa de desenvolvimento, seria necessário criar condições para quando o time estivesse em um lado do campo ou de outro. Caso contrário, se considerados os dois lados da mesma forma, cálculos de reta, por exemplo, teriam resultados diferentes para ambos.

Para resolver esse problema, foi implementada uma função de inversão de referências para cada lado do campo. Isto é, supondo que um dos times ficasse no lado oposto a direção de positiva das coordenadas globais, ele teria todas as suas referências invertidas. É fácil criar essa condição, uma vez que sua direção de ataque seria da coordenada 640 para a 0, "apontando" para a direção negativa do campo.

Com o objetivo de resolver esta inversão, é necessário fazer tanto a inversão de coordenadas absolutas de posição quanto a inversão dos vetores direcionais dos próprios agentes. Então, a nova coordenada x', por exemplo, seria atribuída como $x' = 640 - x$, pois 640 é o comprimento máximo do campo. Assim também ocorre com $y' = 480 -$

y. Para a inversão dos vetores direcionais referentes a cada agente individual, é necessário somente multiplicar seus vetores unidirecionais frontal e do lado direito por -1, desta forma serão invertidos e as direções estarão de acordo com as novas coordenadas atribuídas.

Com este processo, as duas estratégias podem ser desenvolvidas de forma igual, não sendo necessário se preocupar com o tratamento de coordenadas absolutas e, desta forma, só sendo necessário uma nova conversão para estas caso as coordenadas sejam utilizadas com o propósito de desenhar parâmetros gráficos, por exemplo.

4. UTILIZAÇÃO DO OPENGL

O OpenGl [17] é uma biblioteca gráfica de código aberto e baixo nível que tem como objetivo desenhar cenas criadas por código pelo desenvolvedor. Criada em 1992, o esta biblioteca é a mais usada mundialmente para criação de aplicações de gráficos tanto 2D quanto 3D. Além disso, sua alta performance permite sua confiabilidade para utilização até mesmo computação de alto desempenho.

Por ser uma das bibliotecas de uso mais próximo do hardware que existem hoje, este projeto se utilizará de uma outra biblioteca baseada no OpenGl, a FreeGLUT. Ela simplifica e encapsula de maneira razoável o código descrito pela sua biblioteca de origem, trabalhando com funções de retorno. Isso ajuda a criar os gráficos desejados e medir os efeitos da estratégia de forma eficiente.

O único limitante da FreeGLUT é que, com sua utilização, o código fica dependente desta para diversas funções, como entrada de dados pelo teclado ou movimento do mouse. Isso impede sua retirada para aceleração do código em ambiente físico exclusivamente, sem uso de gráficos. Ela também tem funções de flexibilidade limitada, pois não é oferecido ao desenvolvedor a opção de alterar o modo como funciona. Versões posteriores do projeto podem se dedicar a essa parte para que, inclusive, o uso de paralelização seja viável.

5. MÉTODO DE UTILIZAÇÃO

Para se entender como utilizar o simulador VSSS, é necessário visualizar sua implementação para que seja entendido corretamente. As partes descritas através das seções de 2 a 4 podem ter sua estrutura melhor visualizada dentro do ambiente simulado através da figura 5.1.

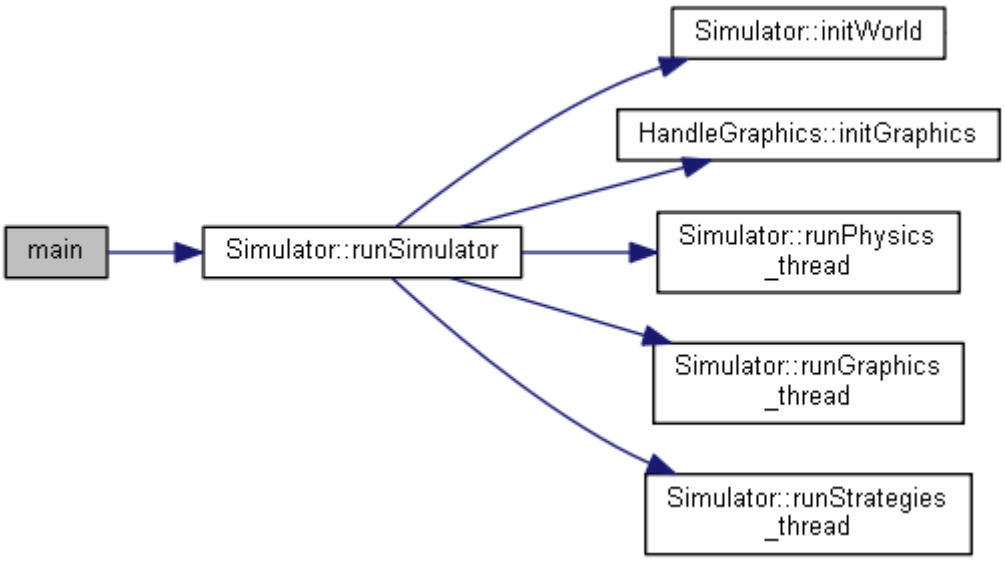


Figura 5.1 - Implementação do paralelismo dentro do simulador

Na figura, é demonstrada a implementação das seções citadas através da utilização do paralelismo das chamadas *threads*. Cada uma, como seu próprio nome diz, é responsável pela execução de uma parte do simulador e podem trocar informações entre si.

Dentro dos métodos *initWorld* e *initGraphics* há somente algumas diretivas para o sistema iniciar corretamente. As *threads* identificadas como *runPhysics* e *runGraphics* são responsáveis pelo que foi descrito na seção 2, 3 e 4 respectivamente. Já a *runStrategies* será discutida mais profundamente nesta seção.

As *threads* trocam informações entre si. Por exemplo, a captura de teclado realizada pela *runGraphics* é compartilhada para o restante do sistema. Já unidades físicas como pontos cartesianos e velocidades são calculados pela *runPhysics* e compartilhados com *runStrategies*, de onde a estratégia e a movimentação dos agentes será feita para retorno e compartilhamento novamente com a *runPhysics*. Para visualização da relação entre as classes de forma completa, ver Anexo D.

5.1 ATALHOS PARA AVALIAÇÃO DO JOGO

Com o objetivo de se ter melhor visualização do estado de jogo, alguns atalhos importantes foram implementados:

- Barra de espaço - pausa o jogo no último estado calculado pela física.
- Tecla P - funciona quando o jogo esta parado. É utilizada para visualização estática, avançando o jogo estado por estado.
- Tecla D - tecla de debug da física do jogo. Retira as imagens gráficas do jogo, deixando somente as bordas.

5.2 DESENVOLVIMENTO DE ESTRATÉGIAS

Como o foco deste projeto é a aplicação de técnicas de aprendizado de máquina na categoria VSSS através da simulação, um método prático para o desenvolvimento de estratégias e comportamento dos agentes é necessário para a execução destes.

Desta forma, as seções a seguir tem o objetivo de esclarecer o funcionamento interno da programação da lógica dos agentes.

5.2.1 DESCRIÇÃO DAS CLASSES E MÉTODOS

- *ModelStrategy.h*

Classe pai que é responsável pela definição básica de todas as classes que envolvam a lógica do agente, como por exemplo a *Strategy.h* descrita no Anexo B. Seus principais métodos são:

- *runStrategy* - método virtual puro. Estabelece os parâmetros obrigatórios para entrada e execução da lógica do robô.
- *set/getAtackDir* - responsável pela direção atacante. Como as coordenadas são relativas e não globais para cálculo da lógica, a variável é responsável por fazer a conversão de volta para coordenadas globais para os gráficos, por exemplo.
- *getStrategyHistory* - responsável por armazenar estados passados do estado como um todo. O tamanho desse vetor depende do tipo de estratégia, com tamanho mínimo de um.

- *getRobotStrategies* - guarda as variáveis e métodos estratégicos de um agente, como ponto de objetivo, distância relativa para qualquer ponto, etc. Sendo limitada a variáveis físicas em relação ao time adversário, uma vez que não há conhecimento prévio da lógica adotada.
- *updates* - responsáveis pela atualização de variáveis inerentes a dinâmica dos agentes, como velocidades linear e angular, aceleração, etc.
- *getDrawComponents* - tem o objetivo de fornecer os pontos requisitados para serem desenhados pela classe gráfica. Os pontos são calculados segundo a estratégia implementada pelo próprio desenvolvedor.

- *Strategy.h*

Classe fornecida para exemplificação de uma estratégia. Seus métodos são implementados de acordo com o requerido pela classe herdada *ModelStrategy.h*. A herança é obrigatória para funcionamento correto da simulação, que relata erros caso implementada de outra forma.

- *ArtificialIntelligence.h*

Esta classe não possui métodos específicos. A implementação destes irá depender do tipo de algoritmo de aprendizado de máquina ou otimização que o desenvolvedor desejará implementar. Descartável se for preferente o uso de lógica booleana. Neste caso somente a classe *Strategy.h* é suficiente.

Os métodos mostrados através do Anexo B são de uso do algoritmo de aprendizado de máquina que será descrito mais a frente.

- *RobotStrategy.h*

Responsável por armazenar variáveis importantes a estratégia do agente. Pode ser vista através do anexo C. Seus métodos principais são:

- *getPosition* - retorna a posição do agente naquele estado.
- *getMaxCommand* - equivalente ao máximo de PWM convertido para metros por segundo. Medido empiricamente.
- *getLocalFront/Right* - funções responsáveis por retornarem o vetor unitário frontal e direito do agente, respectivamente.
- *getId* - retorna a identificação do robô.

- *getCommand/updateCommand* - responsáveis por retornar e atualizar o comando do agente definido pela estratégia em *Strategy.h*.
- *get/setTargetPosition* - retornam e atribuem a posição final para a qual o agente deve direcionar sua movimentação. Valor definido também em *Strategy.h*.
- *getFieldAngle* - retorna o ângulo em relação ás coordenadas globais em campo em relação ao seu vetor unitário frontal.
- *invertLocalFront* - responsável por inverter o vetor unitário frontal. Utilizado para cálculos de coordenadas relativas.

5.2.2 IMPLEMENTAÇÃO

Como visto através do diagrama de classes do anexo B, a criação e execução desses códigos é feita através da herança da classe *StrategyModel.h*. Ali estão todos os cálculos básicos a serem feitos para que uma estratégia possa ser usada, como métodos de inserção de comandos para o robô e inserção de dados para a estratégia, *updateCommand* e *runStrategy*, respectivamente. Assim, as classes filhas dessas, como por exemplo a estratégia criada na classe *SampleStrategy.cpp*, tem de ser herdadas para que o simulador VSSS possa funcionar corretamente.

Assim, a lógica pode ser feita da forma como o desenvolvedor preferir, seja com técnicas de aprendizagem ou lógicas pré definidas, bastando que seja usado o método *updateCommand* para que o robô tenha uma ação a executar. Na classe ainda estão presentes a posição e velocidade de outros agentes no campo, o que possibilita uma ação conjunta entre todos.

Um requisito importante, porém, é a definição de uma aceleração máxima e uma máxima velocidade, definidas através dos *defines* *MAX_ACCELERATION* e *MAX_VELOCITY* em *RobotPhysics.h* e *RobotStrategy.h* respectivamente. Essa definição é necessária devido á escolha do ambiente físico perfeito, explicado anteriormente. Isso por que sem um valor para a velocidade máxima os impulsos vão se acumulando ás velocidades lineares, causando resultados finais desproporcionais.

Quanto a aceleração, a modelagem de um motor real é muito complexa, como discutido anteriormente na seção 2.3. Por isso a taxa deve ser fixa para simplificar os cálculos, que foram já demonstrados. Esses valores devem ser atribuídos de acordo com

a preferência do desenvolvedor, lembrando que as unidades físicas utilizadas para cada um são $\frac{cm}{s}$ e $\frac{cm}{s^2}$.

Para execução da estratégia criada, basta criar um objeto da classe representada por esta e passar por parâmetro para o método *runSimulator*.

5.3 EXECUÇÃO DO SIMULADOR

Quanto a execução do código, é necessário somente o sistema Linux com um compilador C++ com a biblioteca gráfica. Neste projeto foi utilizado o *g++* e a biblioteca *freeglut*, os quais podem ser instalados através de comandos *apt-get*. É necessário entrar na pasta do simulador presente no computador através do Terminal e executar o comando *make* para compilar e *make run* para fazer a execução da simulação.

5.4 CONSIDERAÇÕES SOBRE A UTILIZAÇÃO

Com as características descritas anteriormente, o simulador VSSS adquire a capacidade de simular situações reais que podem estar presentes em uma partida. Apesar de poderem haver certas limitações na simulação que são provenientes da abordagem adotada para extração das características da física real, a modelagem do modelo VSSS através do simulador se mostra muito eficiente para testes entre modelos determinísticos e a avaliação da qualidade da mesma.

6. APRENDIZADO DE MÁQUINA

Atualmente, a área que vem mais ganhando notoriedade dentro da computação é o aprendizado de máquina. Empresas estão focando em áreas que misturam a técnica com o que se chama de *Big Data*, que é definido como uma grande massa de conteúdo criada principalmente pelos diversos serviços disponíveis na internet e que capturam informações de seus usuários. O aprendizado de máquina vem sendo bastante usado para se analisar o reconhecimento de padrões dentro desses volumes de dados que inicialmente não possuem ligação alguma.

Um exemplo muito comum onde estes algoritmos são utilizados é na sugestão de vídeos do site *Youtube*. O *Google* vem investindo neste campo há alguns anos, e vem apostando na área para o crescimento da qualidade e eficiência de seus produtos. Muito desse poderio vem principalmente do uso massivo pelo seu site por usuários. O algoritmo do *Youtube*, no caso, possui mecanismos que permitem que o site sugira vídeos que estejam afinados com o que vem sendo assistido pelo usuário.

Mas além de ser utilizado para esse objetivo, a área de aprendizado de máquina também vem desenvolvendo outras características interessantes, principalmente no que é referente a tomadas de decisão. Várias são as técnicas que se utilizam de leitura de sensores dentro de um robô ou agente presente em uma simulação para calcular as melhores ações a serem tomadas naquele instante de tempo.

Como exemplo pode ser citado o algoritmo AlphaGo [10], que por muito tempo foi considerado um tabu nesta área devido ao grande número de estados possíveis que podem ocorrer dentro do jogo Go, muito popular principalmente no oriente. São cerca de 2.08×10^{170}, o que, como parâmetro de comparação, é mais do que o número de partículas estimado no universo.

Nesta parte do trabalho, o objetivo é apresentar as vantagens de se possuir um ambiente de simulação em que algumas técnicas de aprendizado de máquina podem ser utilizadas, com foco na área de tomadas de decisão, também conhecida como aprendizado por reforço. Como descrito anteriormente, o VSSS é uma categoria simples de futebol de robô, que pode ser usada para se testar algoritmos que precisem de um ambiente pouco complexo para provar sua eficiência.

Este objetivo é mais fácil de ser alcançado por que temos um simulador. Agora o desenvolvimento desses algoritmos ficam independentes do mundo físico. Além disso temos a possibilidade da aceleração na simulação, o que pode facilitar bastante a criação

de métodos de aprendizado. O objetivo desta seção é mostrar a utilização, então, do ambiente de simulação para desenvolvimento de um algoritmo chamado *Deep Q-learning* (DQN), utilizado para aprendizado de jogos de Atari [21]. A técnica se utiliza de diversos conceitos e, por isso, estes serão mostrados antes de mostrar como aplicar a técnica.

6.1 REDES NEURAIS

As redes neurais artificiais (RNA) foram criadas com o objetivo de se imitar o cérebro humano e seu processo de aprendizagem. Sua estrutura imita o uso dos neurônios, possuindo algo similar com as conexões entre estes através de sinapses. Neste sentido, elas podem se adaptar para representar funções muito bem.

As RNAs são muito utilizadas para fazer a aproximação de funções a partir de uma amostra de dados. Elas podem, por exemplo, fazer a classificação e o reconhecimento de padrão em imagens. Elas possuem outras outras finalidades, como pode ser visto em [10, 24], mas que não fazem parte do escopo deste trabalho.

Com o objetivo representar esta arquitetura artificialmente, as primeiras redes neurais foram criadas com base nestes mesmos princípios. Chamada de *Perceptron*, uma destas primeiras redes possui ativação a partir da soma dos valores das entradas ponderadas através de seus respectivos pesos, que pode ser verificado na figura 6.1. Em toda rede neural, são os pesos que guardam as informações e, portanto, devem ser calculados para que se possa obter a saída desejada.

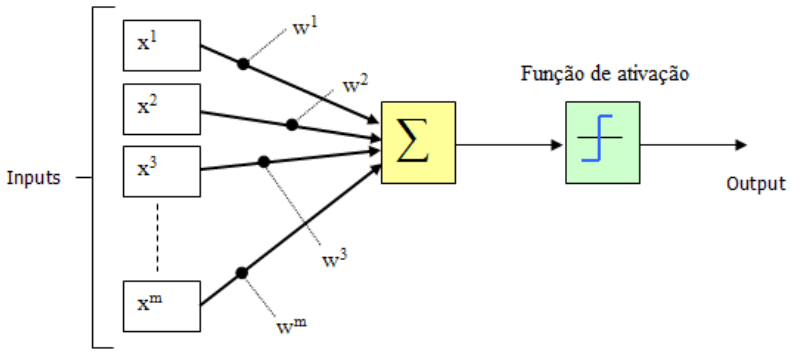


Figura 6.1 - Representação de uma rede *Perceptron*

Como mencionado, a rede demonstrada na figura 6.1 tem recepção de sinais de entrada, conhecidos como *inputs*, e são representados pelo domínio fornecido através dos dados para a rede. Assim, esta possui um valor limite ao qual, se ultrapassado, ela

ativará o nó de saída, ou *output*. A função de ativação correspondente pode ser encontrada em 6.1, com w_j representando os pesos e x_j representando os *inputs*.

$$output = \begin{cases} 0\ se\ \sum_j w_j x_j\ \leq valor\ limite \\ 1\ se\ \sum_j w_j x_j > valor\ limite \end{cases} \tag{6.1}$$

Porém, esta equação ainda pode ser simplificada. Podemos passar o valor que é definido como limite de ativação para o outro lado da equação. Para adotarmos uma notação, o chamaremos de *bias*, e desta forma a equação ficará na forma de 6.2 [25].

$$output = \begin{cases} 0\ se\ \sum_j w_j x_j - bias\ \leq 0 \\ 1\ se\ \sum_j w_j x_j - bias > 0 \end{cases} \tag{6.2}$$

O *bias* representa a tendência a da rede de se comportar de uma determinada maneira. Por isso ele sempre terá um *input* equivalente a -1, e seu peso definirá a tendência da rede neural, como é mostrado na figura 6.2.

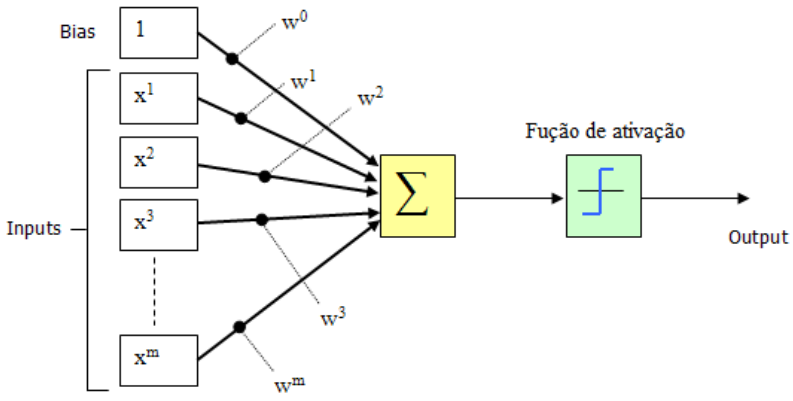


Figura 6.2 - Demonstração de uma rede *Perceptron* com presença de Bias

Um problema relacionado a equação 6.2 é que, como mencionado, são os pesos que guardam toda a informação adquirida pela rede neural. Então é de nosso interesse fazer com que pequenas mudanças nestes pesos aconteçam para que pequenas mudanças na saída da rede também ocorram. Isso não é feito pela equação 6.2, pois em

um momento a resposta pode ser 0, mas com uma pequena mudança nos pesos podemos obter uma resposta igual a 1.

Para corrigir isso, a adoção de uma função diferenciável que varie entre 0 e 1 é necessária, sendo geralmente adotada a função de sigmoide [18], dada pela equação 6.3.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{6.3}$$

Onde $\sigma(z)$ representa a saída da rede e z é definido na equação 6.4.

$$z = \sum_j w_j x_j - bias \tag{6.4}$$

Desta forma, podemos obter uma função que nos dará uma resposta mais amortizada do que a função degrau apresentada pela rede *Perceptron*. E a diferença entre as duas técnicas pode ser vista na figura 6.3.

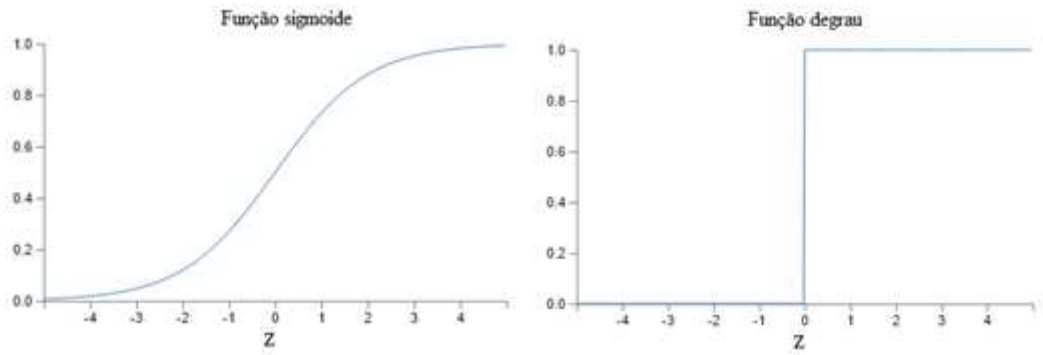


Figura 6.3 - **Esquerda:** Representação de uma função sigmoide**. Direita:** resposta da rede *Perceptron*

Com a mudança para a nova função de ativação, mais do que a utilidade de podermos mudar os pesos com a adição de pequenos valores, agora podemos contar com uma função derivável para realizar o processo de aprendizagem do algoritmo.

6.1.1 ARQUITETURA DAS REDES NEURAIS

A rede aprendida até agora é capaz apenas de classificar funções lineares, como o operador AND, por exemplo. Mas a rede apresenta problemas para fazer a aproximação de funções que classifiquem um domínio não linear, como por exemplo a função XOR, pois por mais que se variem os pesos, nunca irá se conseguir chegar em uma solução satisfatória. Mas um resultado diferente pode ser obtido se uma topologia

multi camadas for construída. Este tipo de rede é demonstrada pela figura 6.4. Com esta rede será possível classificar uma função XOR corretamente.

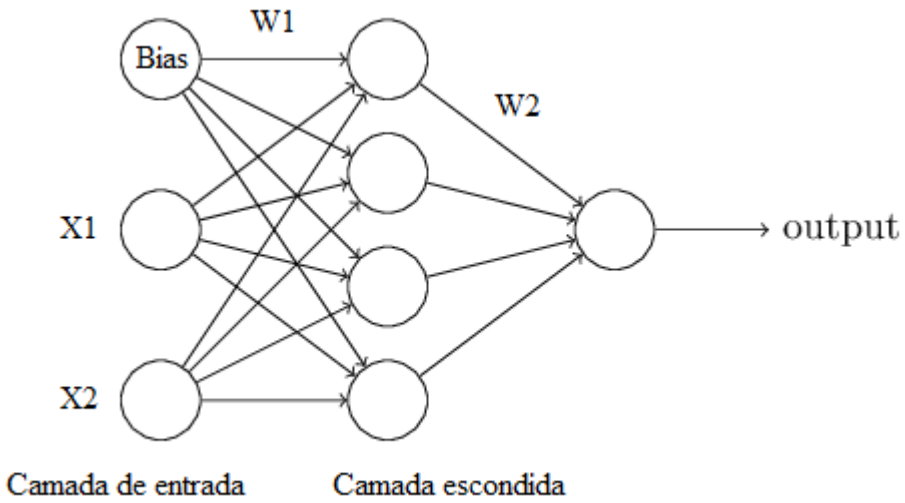


Figura 6.4 - Exemplo de rede neural com múltiplas camadas

A rede da figura 6.4 é chamada de *Multi Layer Percpeptron* (MLP). Redes de mais de uma camada são muito úteis para os mais diversos propósitos. Elas são capazes de cobrir absolutamente qualquer função, seja ela linear ou não linear, porém isso depende da topologia que assumirem [26].

A alimentação e o processamento dos cálculos pela rede é feito por uma operação chamada de *feedforward*, que tem como objetivo fazer com que a saída dos nós de uma camada seja a entrada para os nós da próxima. Um algoritmo de exemplo de uso desta operação para a rede da figura 6.4 segue abaixo:

Algoritmo de aplicação de velocidade em coordenadas locais

```
bias = 1
x = {1 , 2, bias}
resProdutoMatriz1 = produtoMatriz( x, W1 )
outCamadaEscondida = funcaoAtivacao( resProdutoMatriz )
resProdutoMatriz2 = produtoMatriz( outCamadaEscondida, W2 )
output = funcaoAtivacao( resProdutoMatriz2 )
```

Neste algoritmo, o método chamado produtoMatriz fará a operação de produto matricial entre os *inputs* dos nós e os respectivos pesos, que é o mesmo resultado obtido pela equação 6.4. Isto produzirá um vetor de *inputs* para os nós da camada em questão.

O método funcaoAtivacao receberá este vetor e substituirá cada elemento na função 6.3.

Sabendo como executar a operação de *feedforward*, agora podemos partir para a próxima seção, que vai tratar da aprendizagem do valor dos pesos.

6.1.2 GRADIENTE DESCENDENTE E O BACKPROPAGATION

Na literatura de redes neurais artificiais, o um dos métodos de aprendizagem mais amplamente utilizados é sem duvida o gradiente descendente estocástico. Este método serve para que se possa mudar os valores dos pesos de forma se obtenham as saídas desejadas. Antes de chegarmos ao entendimento sobre gradiente descendente, porém, é importante passarmos antes pela definição do que é função de custo para a rede neural.

A função de custo é uma função que definirá quão perto a atual disposição dos valores dos pesos esta para que a função determinada pela entrada x e pela saída y seja aproximada pela rede. Para tanto, utilizaremos inicialmente uma função de custo de fácil entendimento, que é representada pela equação 6.5. Esta função no entanto servirá somente para o escopo de algoritmos de aprendizagem supervisionado, e terá que sofrer modificações para ser adaptada ao aprendizado por reforço, que é o objeto principal deste trabalho.

$$C = \frac{1}{2}\sum_j (y_j - \sigma(z)_j)^2 \tag{6.5}$$

Onde C representa o custo da função. Portanto quanto maior C, maior a distância da solução ideal. y_j representa cada valor que foi previamente definido por um professor, que pode ser um humano, como a resposta certa para a entrada do vetor x_j. E, por fim, $\sigma(z)_j$ é a resposta prevista pela rede.

Podemos perceber que a função de custo é essencialmente uma diferença entre o valor correto a ser obtido pela rede neural e o valor que a rede esta realmente calculando. Em uma interpretação simples com uma RNA com dois pesos somente, a equação é quadrática para que ocorra uma concavidade de acordo com o que é mostrado na figura 6.5. Com isso, fica claro que o ponto onde devemos chegar é o vale da concavidade, que é onde a função de custo alcança seu menor valor, e o dali podemos tirar também o valor ideal dos pesos para que isso ocorra.

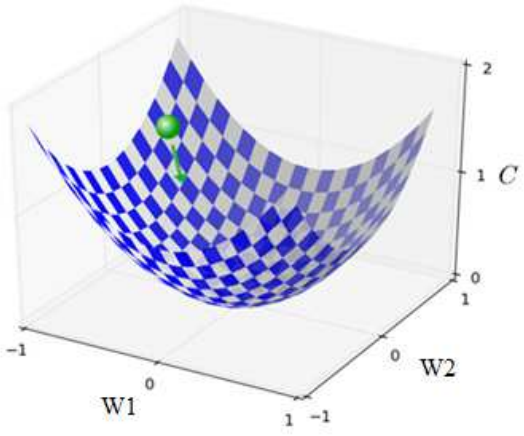


Figura 6.5 - Gráfico que representa o método de gradiente descendente

Para uma rede de muitas camadas como demonstrado na seção anterior, porém, a demonstração gráfica é impossível, devido ao maior número de pesos e, assim, ao maior número de dimensões que o problema trata. Além disso, a função de custo demonstrada pela equação 6.5 é apenas uma das muitas utilizadas na literatura de redes neurais. Em outras etapas deste projeto serão usadas outras funções, mas estas terão o mesmo efeito causado pela equação demonstrada. A bola verde representada na figura seria a inicialização dos pesos e a seta o gradiente.

Para que os pesos cheguem até o ponto mais baixo do vale, devemos nos utilizar das derivadas parciais para calcularmos a influência de cada um em cima da função de custo [18]. A equação responsável para que este processo ocorra é mostrada em 6.6. Para simplificar este exemplo, utilizaremos ainda a rede *Perceptron*, para que haja somente duas camadas e um vetor de pesos a ser corrigido.

$$\Delta w_j = \alpha \ \frac{\partial C}{\partial w_j} \tag{6.6}$$

Onde Δw_j é o resultado do gradiente, α é um valor real, que define a intensidade do passo a ser dado em direção ao vale. Este último é o que na literatura se chama de hiperparâmetro, e é arbitrado pelo desenvolvedor, que deve escolher o valor com o qual a rede faça melhor aprendizagem.

Por fim, para que o peso convirja para o ponto desejado, é necessário somente atualiza-los com base no que foi adquirido através do Δw_j. Com isso, pode-se obter a equação mostrada em 6.7.

$$w_j = \ w_j - \Delta w_j \tag{6.7}$$

Assim, terminamos a definição do método de gradiente descendente. Este processo irá se repetir todos os ciclos até que a função de custo da rede neural alcance um valor pequeno, sendo $C < \varepsilon$, onde ε é arbitrado pelo desenvolvedor. Um algoritmo de exemplo mostrando a junção do gradiente descendente com o *backpropagation* será mostrado mais adiante.

A função deste algoritmo é exatamente a descrita e, como o próprio nome diz, o objetivo é "propagar para trás" o valor da função de custo. Com esse objetivo, devemos antes chegar há algumas definições que serão necessárias para se executar o algoritmo. Uma destas é a derivada da nossa função de custo. Aqui, será utilizada como exemplo a demonstrada em 6.5, pois como dito . Primeiro será demonstrada a derivada da função de ativação, necessária devido ao uso do método da cadeia que será usado para realizar a derivada parcial definida por $\frac{\partial C}{\partial w_j}$ em 6.6, visto em [18].

$$\begin{aligned}
\frac{d}{dz}\sigma(z) &= \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{(1+e^{-z})}{(1+e^{-z})^2} - \left(\frac{1}{1+e^{-z}}\right)^2 \\
&= \sigma(z) - \sigma(z)^2 \\
= \frac{d}{dz}\sigma(z) &= \sigma(z)(1-\sigma(z))
\end{aligned} \tag{6.8}$$

Para entrarmos na próxima parte do algoritmo *backpropagation*, será necessário que se faça a definição de algumas variáveis, que podem ser vistas abaixo:

- x_j^l : onde x é o *input* do nó j referente a camada l, representado por z na equação 6.4.
- W_{ij}^l : onde W é o peso que liga o nó i da camada $l-1$ ao nó j da camada l.
- $\sigma(z)$: continua sendo a função de ativação.
- b^l : é o *bias* referente a camada l.
- O_j^l : é o *output* do nó j referente a camada l, que é o valor obtido na função de ativação.
- y_k : é o valor objetivo do nó k da camada de saída.

Definidas essas notações, agora podemos seguir com a definição de $\frac{\partial C}{\partial W_{ij}^{l}}$. Note que substituímos aqui o w_j por W_{ij}^{l} para que o *backpropagation* funcione para redes neurais de múltiplas camadas. Começaremos primeiro com a definição da derivada para a camada de saída, e a solução para esta é mostrada em 6.7. Para chegarmos lá, nos utilizaremos da regra da cadeia em $\frac{\partial C}{\partial W_{ik}}$, assim:

$$\frac{\partial C}{\partial W_{ik}} = \frac{\partial}{\partial W_{ik}} \frac{1}{2} \sum_k (y_k - \sigma(x_k))^2$$

$$= (y_k - \sigma(x_k)) \frac{\partial}{\partial W_{ik}} \sigma(x_k)$$

Note que k é uma notação que representa um nó da camada de saída. Aqui, o somatório irá sumir pois a derivada do nó k em relação aos outros pesos dos outros nós é zero. Substituindo a derivada da função de ativação encontrada em 6.9:

$$= (y_k - \sigma(x_k))\sigma(x_k)(1 - \sigma(x_k)) \frac{\partial}{\partial W_{ik}} x_k$$

x_k, como mencionado, representa o somatório mostrado na equação 6.4. Deste modo, sua derivada em função de W_{ik} é o *output* do nó referente a ligação com o nó k da camada de saída. Também sabendo que $\sigma(x_k)$ é o mesmo que O_k, obtemos:

$$\frac{\partial C}{\partial W_{ik}} = (y_k - O_k) O_k (1 - O_k) O_j^l \tag{6.9}$$

O resultado encontrado em 6.9 é, portanto, o componente que deve ser substituído para atualização dos peso na equação 6.7. Para melhor representação, vamos alterá-la para que somente atualize os pesos presentes entre a penúltima e a última camadas, e chamaremos o resultado encontrado em 6.9 de ΔW_{ik}. Esta modificação é demonstrada em 6.11.

$$W_{ik} = W_{ik} - \alpha \Delta W_{ik} \tag{6.10}$$

Essa definição, porém, servirá somente para a camada de saída. Para avaliarmos a atualização do vetor de pesos W1 presente na figura 6.4, porém, precisaremos propagar a função de custo também para eles. Isto pode ser obtido, novamente, realizando a regra da cadeia para $\frac{\partial C}{\partial W_{ij}^{l}}$. Deste modo:

$$\frac{\partial C}{\partial W_{ij}^{l-1}} = \frac{\partial}{\partial W_{ij}^{l-1}} \frac{1}{2} \sum_k (y_k - \sigma(x_k))^2$$

$$= \sum_k (y_k - O_k) \frac{\partial}{\partial W_{ij}^{l-1}} O_k$$

$$= \sum_k (y_k - O_k) O_k (1 - O_k) \frac{\partial}{\partial W_{ij}^{l-1}} x_k$$

Até aqui tudo foi calculado de forma bastante parecida com como encontramos a equação 6.9, exceto pelo somatório, que desta vez permanece por que os pesos em W1 influenciam diretamente em cada um dos nós na camada de saída. Agora, para podermos prosseguir, vamos transformar o termo $\frac{\partial}{\partial W_{ij}^{l-1}} x_k$ multiplicando-o em cima e embaixo por ∂O_j^l. Obtemos desta forma $\frac{\partial x_k}{\partial O_j^l} \cdot \frac{\partial O_j^l}{\partial W_{ij}^{l-1}}$, para que conseguirmos realizar a derivada. Acontece que, utilizando a definição da equação, o termo $\frac{\partial x_k}{\partial O_j^l}$ resulta simplesmente no peso W_{ik}. Substituindo estes termos na equação encontrada, temos:

$$= \sum_k (y_k - O_k) O_k (1 - O_k) W_{ik} \frac{\partial O_j^l}{\partial W_{ij}^{l-1}}$$

Agora a derivada não possui mais nenhum termo k, o que nos possibilita tirá-la do somatório e seguir com os cálculos. Assim:

$$= \frac{\partial O_j^l}{\partial W_{ij}^{l-1}} \sum_k (y_k - O_k) O_k (1 - O_k) W_{ik}$$

$$= O_j^l (y_k - O_j^l) \frac{\partial x_j^l}{\partial W_{ij}^{l-1}} \sum_k (y_k - O_k) O_k (1 - O_k) W_{ik}$$

$$\frac{\partial C}{\partial W_{ij}^{l-1}} = O_j^l (y_k - O_j^l) O_j^{l-1} \sum_k (y_k - O_k) O_k (1 - O_k) W_{ik}$$

Para melhor visualização, chamaremos a parte da equação $(y_k - O_k) O_k (1 - O_k)$ encontrada em 6.9 de γ_k. Desta forma obtemos:

$$\frac{\partial C}{\partial W_{ij}^{l-1}} = O_j^l (y_k - O_j^l) O_j^{l-1} \sum_k \gamma_k W_{ik} \qquad (6.11)$$

E, por fim, é só substituirmos 6.11 na equação na 6.7. Agora podemos utilizar as equações 6.9 e 6.11 para fazermos a atualização dos pesos na rede de topologia apresentada na figura 6.4 a cada ciclo. Apesar de a maioria das funções não lineares poderem ser tratadas com uma topologia envolvendo apenas estas três camadas, é possível encontrar na literatura de redes neurais vários casos em que mais camadas são necessárias. Para estes, precisamos desenvolver um algoritmo de *backpropagation* universal, que funcione para todas as topologias que podem ser criadas para uma MPL.

Para que isso seja feito, nos utilizaremos das equações já desenvolvidas. Podemos perceber que, quando o erro da função de custo é propagado dos peso em W2 para os pesos em W1, há uma dependência da derivada dos pesos em W1 por variáveis da próxima camada. E isso irá acontecer sempre, de modo que a derivadas dos pesos na camada atual sempre vão depender da próxima e assim sucessivamente.

Como exemplo, vamos adotar a MPL descrita na figura 6.4, mas com uma camada escondida a mais. Por intuição, podemos imaginar que o erro do nó k na camada de saída seja equivalente a $\delta_k = y_k - O_k$ na equação 6.9. Para a equação 6.11, podemos sugerir que o erro do nó j na camada l seja o equivalente a $\delta_l = \sum_k (y_k - O_k)\, O_k(1 - O_k)W_{ik}$. Portanto, podemos substituir δ_k aqui, e ficamos com a equação $\delta_l = \sum_k \delta_k\, O_k(1 - O_k)W_{ik}$. A notação δ pode ser utilizada para também para a primeira camada da nova rede. Por fim, temos que o erro de um nó pode ser representado através do erro dos nós da camada posterior. Desta forma, temos que:

$$\delta_k = y_k - O_k$$
$$\delta_i = \sum_k \delta_k\, O_i(1 - O_i)W_{ik}$$
$$\delta_i = \sum_j \delta_j\, O_i^l(1 - O_i^l)W_{ij}^l$$

Isso possibilita que, para qualquer camada l, para qualquer peso ligando dois nós, as derivadas dos pesos possam ser reescritas da forma apresentada pela equação 6.12.

$$\frac{\partial C}{\partial W_{ij}^l} = O_j^l \cdot O_j^{l+1}(1 - O_j^{l+1}) \cdot \delta_{l+1} \tag{6.12}$$

Onde O_j^l representa o *output* dos nós da camada calculada naquele instante e δ_{l+1} representa o erro dos nós da camada posterior. Para o *bias*, a única mudança é que, devido ao seu *output* sempre ser 1, ele será atualizado com base no somatório dos erros dos nós da camada posterior, resultando na equação 6.13.

$$\frac{\partial C}{\partial b^l} = \sum_j \delta_j \tag{6.13}$$

Por fim, um algoritmo que se aplica a uma rede neural como a mostrada na figura 6.4 e que se utiliza de todas as partes demonstradas sobre redes neurais é mostrado a seguir:

Algoritmo de aplicação de uma rede neural

```
for i : m do
        bias = 1
        x = carregaDadoBase( i )
        y = carregaRespostaBase( i )
        resProdutoMatriz1 = produtoMatriz( x, W1 )
        outCamadaEscondida = funcaoAtivacao( resProdutoMatriz )
        resProdutoMatriz2 = produtoMatriz( outCamadaEscondida, W2 )
        output = funcaoAtivacao( resProdutoMatriz2 )

        realizaBackpropagation( output, y )
end
```

Onde m é o total de amostras presentes na base de dados.

Um ponto importante relacionarmos agora é que todos os pesos presentes em qualquer tipo de rede neural deve ser inicializado de forma aleatória, preferencialmente entre os valores zero e um. Isso será muito importante para que os pesos não convirjam para uma uniformemente, por exemplo. Essa inicialização irá contribuir para que a RNA aproxime uma função generalizada para os dados que lhe servem de entrada.

6.1.3 TÉCNICAS DE APERFEIÇOAMENTO

Como foi visto, as redes neurais são ferramentas muito utilizadas como um tipo de aproximador de funções de qualquer tipo. Há porém certas características que fazem as redes neurais as vezes se comportarem de forma indesejada. Há muitos métodos que podem ser adotados e que aperfeiçoam, cada um, um aspecto especifico da rede. Apesar de muitos deles serem usados unicamente em aprendizado supervisionado, é útil que se saiba a utilidade de alguns deles. Os métodos para melhora de desempenho da rede são os de regularização e normalização, mais presentes em métodos com supervisão, e o uso de uma função de custo alternativa para a rede, como a de *log-likelyhood*, por exemplo. Estes e outros podem ser vistos em [12].

Há técnicas também de otimização para formação do gradiente presente no *backpropagation*, e que são os usados também neste projeto. Conhecido como RMSProp, o algoritmo otimizador tem a função de incluir e balancear a importância de gradientes calculados em ciclos passados [46].

Porém, os mais úteis para métodos de aprendizado de reforço, além da adaptação da função de custo, são as técnicas de *experience replay* [27] e *batch normalization* [28].

6.2. REDES NEURAIS CONVOLUCIONAIS

Em redes neurais, vimos que o objetivo da técnica é principalmente o reconhecimento de padrões e aproximação de função para que estes padrões sejam definidos. O problema com as RNAs, porém é que elas possuem limitações quando tratam de problemas mais complexos, como imagens que tenham dimensões muito grandes, por exemplo.

Para se classificar imagens de números de 100x100 pixels, por exemplo, seriam necessários dez mil *inputs* para a RNA. Além disso, para que a classificação alcance um padrão satisfatório é necessário que se criem mais camadas escondidas, método de aprendizado conhecido também como *deep learning.* Vamos supor que se obtenha algoritmos que achassem estratégias satisfatórias para os agentes do VSSS, sejam necessárias duas camadas escondidas de seis mil nós cada uma. Supondo que a imagem seja em preto e branco, isso resultaria em um total de quase de 100 x 100 x 6000 + 6000 x 6000 + 6000 x 10 = 96 milhões de pesos diferentes presentes na rede.

As consequências para se ter uma RNA desse tamanho atuando em um problema complexo como esse são grandes. Existem diversos estudos que apontam que o uso de uma RNA com tal quantidade de pesos deve ser feita com muita cautela. Tais estudos podem ser encontrados em [18,29].

Um dos fatores que mais pesam, porém, é o uso dessa quantidade de pesos para processamento do algoritmo de *backpropagation*, pois isso influencia diretamente na demora para realizar a tarefa [30].

As chamadas redes neurais convolucionais (RNC), são tidas como o estado da arte atual quando se trata de processamento de imagens. A estrutura de uma RNC é feita para que haja diminuição na quantidade de pesos e, além disso, imitar de forma muito mais eficiente o que acontece em nosso cérebro.

Ao contrário de uma RNA, que aplica métodos de aprendizado utilizando todos os pixels de uma imagem, a RNC divide a imagem em pequenos quadros, com os quais procura reconhecer características presentes bordas ou sombras em uma imagem. Com esse processo, a RNC diminui consideravelmente o tamanho da imagem, até que, no final de sua estrutura ela faz a classificação desta com base nas características aprendidas. Por exemplo, para caracterizar um ser humano, a rede aprende o que são olhos, nariz e boca. Então, toda imagem que serve de *input* para a rede será verificada para que possam se identificar esses três requisitos. Caso sejam identificados, então a rede classifica a imagem como sendo um ser humano, se não, ela aprende que aquela imagem não pertence ao grupo relacionado [31]. A estrutura padrão de uma RNC pode ser vista na figura 6.6.

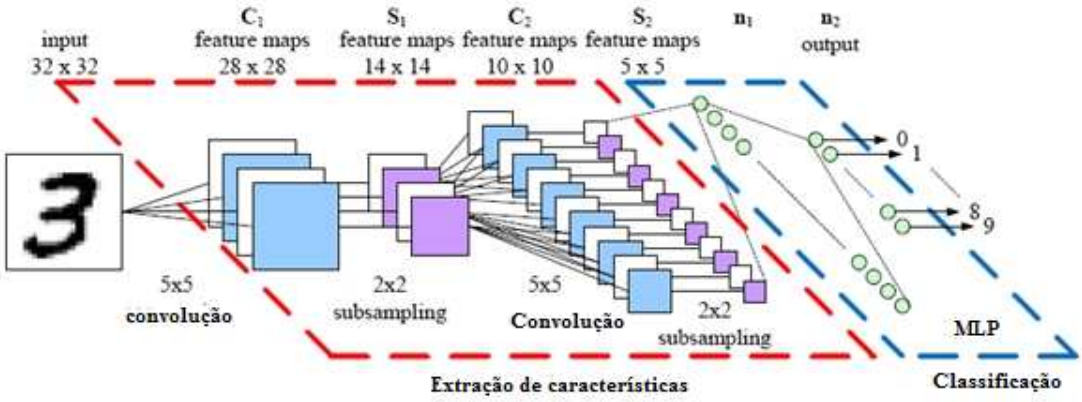


Figura 6.6 - Estrutura de uma rede neural convolucional.

Podemos ver que as redes convolucionais tem em sua estrutura básica camadas diferentes das presentes em uma RNA. São elas a camada convolucional, a camada de *pooling* e, por fim, a camada MLP já conhecida para realizar a classificação da imagem. Abordaremos cada uma delas nas seções a seguir.

6.2.1 CAMADA CONVOLUCIONAL

Para descrever a melhora na eficiência causada pela RNC, vamos adotar que um frame de vídeo do simulador da categoria VSSS tenha 100 x 100. Os *inputs* são de 100 x 100 = 10000. Diferente da RNA, logo após a camada de entrada tem-se uma camada de convolução. Esta camada se utiliza de pequenos mapas, chamados *receptive fields*, que irão ajudar na construção de um segundo volume de nós, chamado de *feature map* [19]. Este tem como objetivo fazer a extração das características mencionadas anteriormente através do exemplo do corpo humano.

Os *receptive fields* são de tamanho arbitrário, e no modelo demonstrado na imagem 6.13 pode se ver que a primeira camada eles possuem tamanho de 5 x 5. Eles varrem toda a imagem procurando a característica com a qual foram treinados, pulando um determinado valor de pixels de cada vez. Este valor é chamado de *stride*, que na figura tem o valor de 1. Há a opção também de contornar a figura com pixels de valor zero, técnica que é conhecida como *zero padding*, e é responsável por permitir que os *receptive fields* hajam também nas bordas das figuras.

Os *repective fields* funcionam como pequenas redes MPL completamente conectadas, e eles fazem sua varredura utilizando os mesmos pesos em toda a imagem, já que esses guardam as informações da característica aprendida, gerando o que é chamado de *feature map*. Um exemplo mais simplificado do funcionamento de uma camada convolucional pode ser visto na figura 6.7. Embora o modelo seja unidimensional, seus efeitos podem ser da mesma forma nas imagens, que representam um modelo bidimensional.

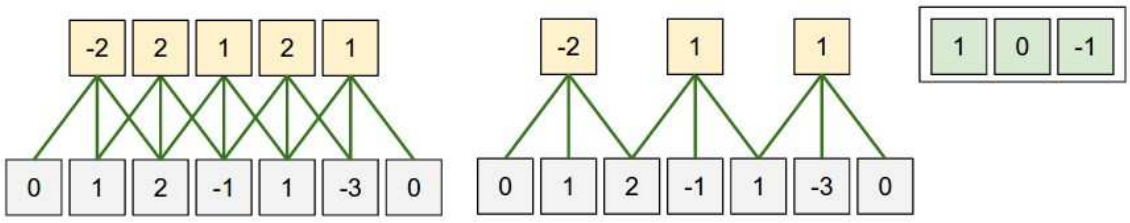


Figura 6.7 - Demonstração do arranjo espacial em uma camada convolucional.

Aqui, os *inputs* para a camada são vistos em branco, e os *feature maps* são mostrados em laranja. Podemos ver duas redes convolucionais atuando com dois valores de *stride* diferentes, e os pesos identificados como as linhas verdes tem os valores indicados na caixa a direita. Para a estrutura mais a esquerda se tem um valor de *stride* de 1. Já para a segunda estrutura o valor usado é 2. Em ambas o valor de *zero padding* é de 1. Para obtermos o tamanho do *feature map* a ser criado na próxima camada é necessário aplicarmos a fórmula descrita em 6.14.

$$\mathrm{N} = \frac{(W - F + 2P)}{S} + 1 \tag{6.14}$$

Onde W é o tamanho lateral do volume de entrada. Para o caso da figura 6.7 esse valor seria 5. F é o tamanho lateral do *receptive field*, que é de 3 no caso da figura. P é

o tamanho do *zero padding* implementado na figura, que no caso é de 1. *S* é o valor do *stride* assumido.

Substituindo os valores na equação 6.14 temos que, para a estrutura mais a esquerda, o tamanho do *feature map* precisa ser de (5 - 3 + 2) / 1 + 1 = 5. E para a segunda estrutura temos um *feature map* de tamanho (5 - 3 + 2) / 2 + 1 = 3. Estas são exatamente as quantidades encontradas para cada uma das estruturas. É importante mencionar que a camada de convolução pode produzir mais de um *feature map*. A quantidade é definida pelo desenvolvedor, que deve verificar qual a quantidade de *feature maps* que melhora a eficiência da rede.

Já para a obtenção dos resultados que definirão o *output* da rede convolucional seguem a mesma lógica das RNAs, com a diferença de que desta vês a multiplicação normal entre *input* e pesos que correspondem ao mesmo índice, mostrada na equação 6.4, será substituída pela convolução, definida pela equação 6.15.

$$z = \sum_j w_{ij} * x_i - bias \tag{6.15}$$

Nota-se que a única diferença para a equação 6.4 é o operador de convolução $*$. Por questão de nomenclatura, um conjunto de pesos também podem ser chamados de *kernel* ou filtro.

A função do operador de convolução é fazer com que ocorra a inversão dos índices no vetor, como no caso da figura 6.7, ou da matriz, conforme mostra o esquema da figura 6.8. Neste caso, a matriz é rotacionada em 180° e então é feito o produto escalar, multiplicando elemento por elemento. Os pesos representam uma determinada função que busca algum tipo de característica e os *inputs* são os dados provenientes da imagem.

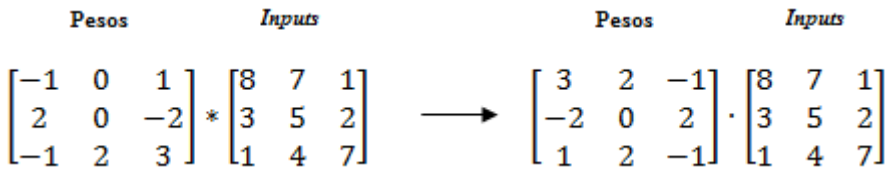


Figura 6.8 - Demonstração do cálculo de convolução

A convolução é necessária devido a sua propriedade associativa, isto é, $a * (b * c) = (a * b) * c$, ao contrário da correlação, que seria a operação realizada caso a matriz não fosse rotacionada. Esta propriedade é muito útil devido aos múltiplos filtros

que podem existir na camada de convolução. Como cada filtro representa uma função de característica diferente, a propriedade melhora a eficiência da rede quando o *feedforward* é aplicado.

Para visualizarmos o que acontece nesta camada ao longo do aprendizado, devemos olhar para a figura 6.9. A figura mostra o que os pesos de um determinado *feature map* aprenderam. Pode-se ver claramente que a característica relacionada a este mapa esta ligada aos contornos presentes na imagem. Esta é uma característica recorrente no aprendizado da primeira camada convolucional.

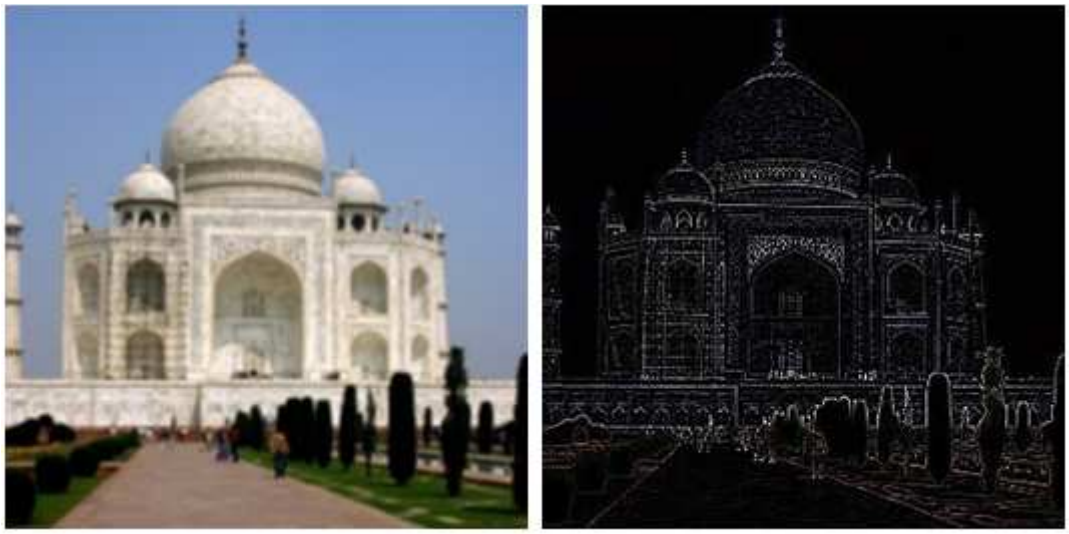


Figura 6.9 - Visualização de um *feature map* a direita após a convolução da imagem a esquerda.

Para deixar clara a execução do algoritmo de convolução em uma imagem, na figura 6.10 esta representada a convolução com *stride* igual a 1. O *receptive field* esta sendo mostrado em amarelo. Os pesos são os números multiplicadores em vermelho, lembrando que o *kernel* em questão já esta rotacionado, como vimos através da imagem 6.8. Em rosa estão os valores presentes no *feature map* até aquele momento, sendo a figura da direita onde ele se encontra completo.

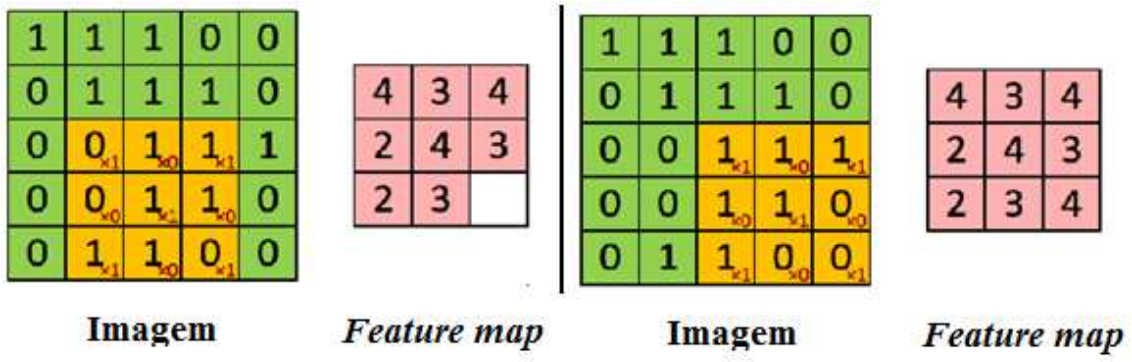


Figura 6.10 - Exemplo de convolução aplicada a uma imagem

Com isso, fechamos a parte relacionada ao processo de *feedforward* para a camada de convolução. Agora passaremos para a camada seguinte, que na literatura de redes convolucionais é chamada de *pooling*.

6.2.2 CAMADA DE *POOLING*

Após a execução da convolução, as redes convolucionais, por padrão, se utilizam da camada de *pooling* para selecionar os *outputs* de maior importância e, portanto, maior influência quando for feita a classificação [19]. Esta camada é bastante simples de ser entendida, e isso é refletido através de seu processo.

Da mesma forma que a camada convolucional, a camada de *pooling* também tem seu *kernel* e seu *stride*, mas não possui pesos. Sua função é apenas realçar as características mais relevantes presentes nos *feature maps* da camada de convolução. Para tanto, há algumas técnicas que podem ser empregadas para que seja feito este processo, porém a mais conhecida é a *max-pooling*.

A *max-pooling* consiste simplesmente da seleção do *output* que possui o valor mais alto dentro do *kernel*. Na figura 6.11 podemos ver a técnica em ação. Neste caso, o *kernel* possui dimensão de 2 x 2 e o *stride* possui valor 2. O processo também é conhecido como *sub sampling*.

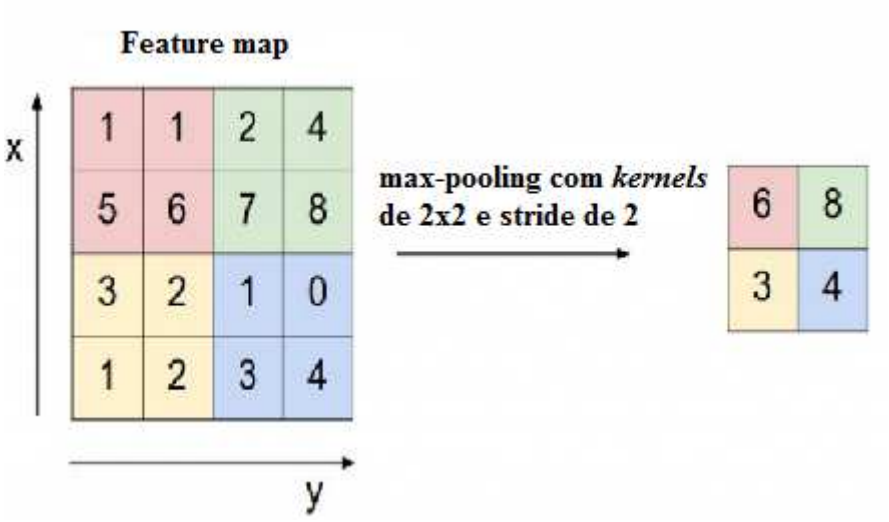


Figura 6.11 - Representação da técnica de *max-pooling*

As camadas convolucional e de *pooling* normalmente trabalham em pares, que podem ser criados seguidamente um após o outro dentro da estrutura de uma rede convolucional. Independente do número de pares usados pelo desenvolvedor, ao final deverá ser usada uma rede MLP para que possa ser feita a classificação correta da imagem. E assim finalizamos a etapa de *feedforward*.

6.2.3 BACKPROPAGATION

O algoritmo de *backpropagation* para as redes convolucionais realiza cálculos que são bastante semelhantes com os que foram aprendidos para as redes MLP. Isso se deve ao fato de que os *kernels* da camada convolucional apresentarem funções muito semelhantes a essas redes. Além disso, a camada de *pooling* não apresenta variância durante o processo do algoritmo, tendo a função apenas de reproduzir o erro calculado pela camada que é construída posteriormente.

A camada de *output* desta rede é calculada como em uma RNA [31]. Sendo assim, seguindo o esquema mostrado pela figura 6.6, logo após essa camada temos uma camada de *pooling* com uma convolucional. Supondo que a técnica utilizada para esse tipo de camada seja a de *max-pooling*, a propagação do erro pode ser demonstrada através da modificação da definição de δ definida na seção de *backpropagation* para as MLPs, de forma que o cálculo possa realizar a operação chamada de *upsample*, chegando a equação 6.16.

$$\delta_i = \sum_j \delta_j \, O_i^l (1 - O_i^l) W_{ij}^l$$

$$\delta_l = \sigma'(z)_l \cdot (W^l \times \delta^{l+1})$$

$$\delta_k = upsample(W_k^l \times \delta_k^{l+1}) \cdot \sigma'(z)_k^l = 0 \quad (6.16)$$

Onde *upsample* é simplesmente a atribuição de erro ao nó que foi selecionado durante a operação de *max-pooling* e, para o restante dos nós presentes na operação, é atribuído erro igual a 0. No caso de haver o cálculo da média, os erros são atribuídos igualmente entre os nós que fizeram parte do *kernel* do *max-pooling*. k representa o índice do *kernel* avaliado naquele momento. Na equação os somatórios foram vetorizados para melhor visualização.

Com a definição de δ_k agora podemos fazer o cálculo dos gradientes referentes a cada um dos filtros presentes na camada convolucional. Nesta etapa, assim como fizemos a rotação do filtro de convolução para a operação de *feedforward*, temo que rotaciona-los novamente para a realização do *backpropagation*. Desta forma, obtemos a equação 6.17 que realizará a atualização dos pesos. A explicação matemática para a rotação do *kernel* pode ser vista em [20].

$$\frac{\partial C}{\partial W_{ij}} = \sum_i O_i^l * rot180(\delta_k^{l+1}) \quad (6.17)$$

E, para *bias* do *kernel*, cuja saída é sempre igual a 1, temos a equação 6.18.

$$\frac{\partial C}{\partial b_k} = \sum_{a,b} (\delta_k^{l+1})_{a,b} \quad (6.18)$$

Com isso, agora só precisamos atualizar cada um dos pesos como descrito anteriormente, na equação 6.10. E assim finalizamos o algoritmo de *backpropagation* para as redes convolucionais e também a parte reservada ao aprendizado supervisionado.

6.3 APRENDIZADO POR REFORÇO

No aprendizado por reforço se tem o objetivo de fazer um agente interagir com o ambiente e calcular a melhor ação dentre várias dentro do espectro disponível para aquele agente.

A definição do que é um agente e o que pode ser considerado um ambiente pode mudar de várias formas. Vamos considerar como exemplo a própria categoria VSSS. Aqui, o agente é definido como sendo um robô e o ambiente é o campo de jogo, o que inclui a presença dos outros robôs, da bola e dos dois gols. Importante afirmar que, independente de qual o ambiente em que o agente esteja inserido (seja o virtual ou seja

o real), essas definições ainda se mantém atribuídas da mesma forma. Uma interpretação gráfica para esta análise pode ser encontrada através da figura 16.12.

Figura 6.12 - Representação de um ciclo de aprendizagem

Para fins de objetividade, vamos continuar usando a categoria VSSS para fazer a interpretação desta parte teórica. O ambiente e o agente já foram exemplificados e definidos. O estado seria toda a informação proveniente do ambiente que fosse relevante para a tomada de decisão do agente. Para o VSSS, estas informações poderiam ser, por exemplo, a posição da bola e do restante dos robôs presentes em campo, de forma que sabemos que devemos perseguir a bola e evitar a colisão com os outros agentes dentro do campo, ou outra estratégia que o próprio algoritmo calcule que seja melhor para o agente.

Já a definição de recompensa é todo valor que o agente obtém após a execução de determinada ação. Desta forma, assim que uma ação é tomada é feita a leitura do ambiente para avaliar qual a recompensa que o agente irá receber. Um exemplo de atribuição de recompensas seria fazer com que a colisão com outros robôs podem contribuir negativamente para este valor, já a proximidade com a bola poderá contribuir de forma positiva.

Por fim, a ação a ser tomada pelo agente, na categoria VSSS, tem relação com os comandos PWM possíveis para cada roda a serem executados pelo robô. Isto é, o robô pode tem 255 x 255 = 65025 ações que ele pode realizar e, por consequência, esse é o número de ações disponíveis para o algoritmo de aprendizado.

Agora, precisamos de uma forma de juntar todos os aspectos descritos de forma que o robô aprenda a tomar decisões com base nas premiações que recebe da leitura do ambiente.

6.3.1 PROPRIEDADES DO AMBIENTE

Para começarmos a descobrir como os processos de decisão são feitos, é importante definirmos algumas características presentes nos problemas nos quais estes processos de decisão são criados. Normalmente um problema possui um objetivo principal a ser alcançado, o qual precisa de uma sequência de tomadas de decisão, que em aprendizado por reforço é chamada de π, para ser alcançado. É a partir desta sequência de decisões que o algoritmo pode começar seu processo de aprendizado. Como já mencionado, para a plataforma VSSS o agente possui 65025 opções de decisão.

Cada uma das sequências π possíveis para a solução de um determinado problema funciona como uma função. Elas são dependentes de um estado s para calcular a ação a a ser tomada. Assim como uma função, cada sequência π possui um determinado valor de retorno, chamado de G, que lhe é atribuído através da soma das premiações que ela adquiriu enquanto realizava sua sequência de ações. Como dito, cada ação tomada em determinado estado possui uma recompensa própria.

Assim, o processo responsável por calcular o retorno de uma sequência π é, como definido em [23], demonstrado através da equação 6.19.

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T \tag{6.19}$$

Onde R_t é a recompensa a ser recebida no instante de tempo t, referente ao estado s. R_T define a recompensa que será adquirida no instante final da sequência π. Essa divisão das sequências em instantes de tempo é o que se chama de episódio.

O aprendizado de um agente consiste, então, em maximizar o valor ganho até chegar no instante final, que definido como sendo onde ele alcança seu objetivo principal. Ou seja, a melhor sequência a ser tomada pelo agente, então, é aquela que possuir o maior retorno G.

O problema é que muitos problemas modelados podem possuir sequências muito longas, ou que por vezes podem até mesmo nunca terminar. Isso acaba fazendo com que o somatório tenda a números muito grandes ou infinitos. Para evitar esse tipo de problema, implementa-se então um tipo de desconto no somatório de forma que isso faça com que ele acabe convergindo para algum número real. A equação 6.20 representa este procedimento.

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^T R_T = \sum_k \gamma^k R_{t+k+1} \tag{6.20}$$

Onde γ é um parâmetro que deve sempre estar no intervalo $0 \leq \gamma \leq 1$, é denominado taxa de desconto, que deve ser determinado pelo desenvolvedor. t é o índice de um determinado instante de tempo em que o problema se encontra. Podemos identificar que, obedecido o intervalo estabelecido, o somatório converge por obedecer a lei de convergência de séries geométricas.

A escolha da taxa de desconto certa deve ser feita com cautela, pois se escolhida muito próxima de 0 o somatório tem uma convergência mais rápida, mas o agente acaba não levando em conta premiações futuras. Isto o leva a perseguir somente o maior valor para R_{t+1}. Em compensação, se escolhido muito próxima de 1, a taxa de desconto faz com que o agente faça previsões melhores de suas futuras premiações, mas a convergência do somatório se torna mais lenta.

Em geral, um agente se utiliza da regra definida para obter o valor dos estados pelos quais passa. Assim será também para os algoritmos implementados neste trabalho no VSSS. Um exemplo de problema no qual a solução descrita pode ser aplicada é na *pole balancing*, visto em [23].

6.3.2 PROCESSOS DE DECISÃO DE MARKOV

Em aprendizado por reforço, os agentes tomam suas decisões segundo sinais que recebem do ambiente que são denominados estados do ambiente, como já definimos anteriormente. Acontece que, para que uma decisão seja tomada com uma precisão aceitável é necessário que o sinal ofereça informações suficientes que caracterizem aquele estado específico.

Se todas as informações referentes aquele estado puderem ser obtidas, então é dito que aquele é um estado de Markov, ou que possui a propriedade de Markov. E em aprendizado de máquina, a tarefa que obedece a propriedade de Markov é denominada de processo de decisão de Markov (MDP, do iglês: *Markov Decision Processes*). Apesar disso, se apenas as informações relevantes para a execução da tarefa puderem ser colhidas, é dito que se alcançou um MDP aproximado, o que muitas vezes já é suficiente para que se possa ter uma lógica para tomada de decisões confiável e é o que geralmente é feito.

Esses tipos de processo são de muita importância dentro do campo de aprendizado por reforço. Isso por que ele determina que, se o processo obedece a propriedade de Markov, não é necessário que informações passadas sejam obtidas para se prever a ação a se tomar no presente.

Levando-se em conta a plataforma do VSSS este processo seria caracterizado como um MDP se, por exemplo, em todos os episódios fossem fornecidas informações como velocidades linear e angular, grau em relação ao campo, posição dos jogadores, etc. Enfim, todas as informações físicas e coletáveis presentes no processo. Com essas informações poderiam ser determinados todos os estados s que definem um MDP.

Podemos dizer que um MDP é definido pela dinâmica de um ciclo por vez, isto é, a cada estado s alcançado calcula-se a previsão de premiações para cada próximo estado s' que possa ser alcançado através da execução de uma ação a.

Assumindo que o agente pode ter sua movimentação prejudicada por algum ruído de natureza desconhecida, dizemos que $p(s'|s,a)$ é a probabilidade de um agente executar uma ação a no estado s e acabar parando no estado s'. Igualmente, para cada estado s' alcançado se estabelece uma recompensa a ser conquistada pelo agente, denominada $r(s,a,s')$.

Para o agente poder avaliar qual é a melhor ação a ser tomada quando esta em um determinado estado, utilizaremos a equação descrita em 6.25. Porém, vamos adaptá-la para que possa representar o valor de cada estado dentro de um MDP para que assim, a partir deste valor, o agente saiba a real recompensa que ele ganhará por estar naquele estado e continuar a seguir uma estratégia, ou sequência, π que foi definida para ele.

Para tanto, podemos avaliar até dois métodos diferentes para a aquisição deste valor, que são demonstrados através das equações 6.21 e 6.22, como visto em [23].

$$v_\pi(\mathrm{s}) = \mathrm{E}_\pi[\mathrm{G_t}|\mathrm{S_t} = \mathrm{s}] = \mathrm{E}_\pi[\textstyle\sum_k \gamma^k R_{t+k+1}|\mathrm{S_t} = \mathrm{s}] \quad (6.21)$$

$$q_\pi(\mathrm{s,a}) = \mathrm{E}_\pi[\mathrm{G_t}|\mathrm{S_t} = \mathrm{s},\ \mathrm{A_t} = a] = \mathrm{E}_\pi[\textstyle\sum_k \gamma^k R_{t+k+1}|\ \mathrm{S_t} = \mathrm{s},\ \mathrm{A_t} = a] \quad (6.22)$$

Apesar de 6.25 e 6.26 serem duas equações aparentemente similares, há uma diferença bastante importante entre as duas. Para a equação 6.25, v_π(s) representa, em um instante t, o valor do estado s, e $\mathrm{E}_\pi[.]$ representa o valor esperado por se seguir a sequência π a partir daquele estado. Em contrapartida, na equação 6.26 q_π(s,a) representa, em um instante t, o valor da ação a em um estado s e apenas depois é calculado o valor por seguir a sequência π. Portanto, enquanto v_π(s) é o valor de um determinado estado s, q_π(s,a) é o valor que tem a decisão de se executar uma ação a em um estado s.

Uma propriedade interessante de ambas as equações 6.21 e 6.22 é que, levando-se em conta que elas são calculadas a cada novo episódio, temos que ela cria um ciclo

de recursividade para ambas as variáveis v_π(s) e q_π(s,a). Com isso, podemos guardar os valores e atualiza-los com o tempo.

6.3.3 PROGRAMAÇÃO DINÂMICA E O *VALUE ITERATION*

Como dito anteriormente, muitos problemas podem possuir uma sequência π muito longa até que alcancem um estado terminal. Isso ocorre basicamente para todo processo inerente a plataforma VSSS também, e é fácil de se imaginar o porque. Se o número de frames por segundo, sendo cada frame caracterizando um estado independente, for levado em conta dentro dos cálculos do algoritmo, logicamente irá se ter um desempenho muito fraco do mesmo. Isso porque, levando-se em consideração que os métodos demonstrados pela equação 6.21 é calculado para todo estado s, não importando o episódio em que se encontre o problema. Isso traz um processamento extra indesejado. Isso por que, a partir do momento em que é passado o episódio relativo ao instante t, temos que calcular novamente todos os valores referentes aos episódios seguintes. Isto pode ser evitado através da aplicação do método chamado de programação dinâmica (DP, do inglês: *Dynamic Programming*).

O método consiste simplesmente em guardar as informações calculadas em episódios anteriores referentes aos valores dos estados, para que possam ser usadas novamente em episódios futuros. A equação 6.23 demonstra com clareza como prosseguir com este método:

$$v_{k+1}(\mathrm{s}) = max_a \sum_{\mathrm{s}'} p(\mathrm{s}'|\mathrm{s}, a)[r(s, a, s') + \gamma v_k(s')] \quad (6.23)$$

Onde s' é o estado futuro avaliado quando o agente esta no estado s e toma a ação a. $p(\mathrm{s}'|\mathrm{s}, a)$ é a probabilidade de se sair de um estado s tomando a ação a e parar no estado s'. r é a recompensa que o agente recebe imediatamente pela ação tomada e $v_k(s')$ é o valor do estado futuro naquele instante, antes da atualização de seu valor. A operação max garante que, a cada ciclo, será escolhida sempre a melhor opção. A equação pode ser lida como sendo uma derivação da famosa equação de Bellman [23], portanto a convergência de sua tabela de valores é garantida.

Como a atualização dos valores normalmente não influencia o comportamento do agente depois de somente alguns ciclos, o algoritmo pode ser truncado para que sua recursividade seja parada assim que a diferença na atualização de um estado para o outro seja menor que algum número real arbitrário. Dessa forma, é evitado que o custo computacional seja amplificado em excesso. Isso na prática impede o algoritmo de

chegar ao valor ótimo dos estados, ou v^*, mas é o suficiente para que o comportamento seja o mesmo.

O algoritmo para o procedimento é visto a seguir:

Algoritmo de aplicação do *Value Iteration*

Iniciar o array de valores de v arbitrariamente (exemplo: 0 para todos os estados s)
do
delta = 0
for each s **in** S
temp = $v(s)$
$v(s) = max_a \sum_{s'} p(\mathrm{s}'|\mathrm{s}, a)[r(s, a, s') + \gamma v(s')]$
$\pi(s) = \ argmax_a \sum_{s'} p(\mathrm{s}'|\mathrm{s}, a)[r(s, a, s') + \gamma v(s')]$

delta = $max\ (delta, |temp - \ v(s)|)$
while delta < error (valor real positivo)

O algoritmo envolve, como visto em [23], as técnicas de *policy iteration* e *policy improvement*, realizando um passo de cada etapa a cada ciclo. Isso garante que o algoritmo faça suas iterações corretamente.

A propriedade de Markov, neste caso, é garantida pela parcela $p(\mathrm{s}'|\mathrm{s}, a)$. Por isso o algoritmo depende que a dinâmica da transação entre os seus estados seja calculada previamente.

6.3.4 Q-LEARNING

O *Q-Learing*, diferente do *value iteration*, é um algoritmo baseado em modelo livre. Isso quer dizer que ele não depende de uma definição prévia da dinâmica das transições entre os estados para que consiga calcular o valor destes. Ao contrário, ele se utiliza da técnica que envolve o método de Monte Carlo (MC) [33] para fazer os cálculos dos valores de cada ação enquanto ele as executa, o que é conhecido como aprendizado *online*. É também um algoritmo entendido como *off-policy*, ou seja, ele calcula π de forma paralela ao cálculo do valor ótimo π^*, como visto no algoritmo mostrado mais a frente.

O algoritmo realiza seu aprendizado tentando aprender através da história do agente, sendo uma história uma sequência de ciclos determinadas por $\{s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3 \ldots\}$. Isso significa que o agente estava no estado s_0, realizou a ação a_0 e foi para o estado s_1; neste estado ele realizou a ação a_1 e recebeu a recompensa r_1, parando no estado s_2; e assim sucessivamente.

Para realizar os cálculos, aqui empregamos a técnica chamada de diferenciação temporal (TD). Essa técnica é onde é encaixado o MC neste algoritmo. Porém, apesar de aprender através da experiência do agente como no MC, esta técnica permite a atualização dos valores de $V(s)$ a cada ciclo, ao contrário do MC que precisa esperar por um estado terminal. O TD segue como regra a equação 6.24.

$$V(S_t) = V(S_t) + \alpha[r_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \quad (6.24)$$

Onde $V(S_t)$ é a estimação do valor atual do estado, α é uma constante menor que 1, r_{t+1} é a recompensa imediata inerente a ação tomada e $V(S_{t+1})$ é a estimação do estado no futuro.

A equação mostra que o TD é uma estimação, visto que se utiliza do valor de estado atual $V(S_t)$ e não de uma política V^{π} como em MC, sendo sua convergência ainda garantida [23]. De uma forma simples, podemos interpretar que o TD se utiliza da estimação do estado futuro através de $r_{t+1} + \gamma V(S_{t+1})$ para fazer a estimação do estado atual, fazendo com que a técnica garanta vantagens inerentes tanto de DP como de MC.

De forma similar, o Q-Learning utiliza a mesma técnica de TD mas para aplicar uma estimação de $Q^*(s, a)$, o que significa que cada ação recebe um valor, e não cada estado. Sua equação pode ser vista através de 6.25.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + max_a(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad (6.25)$$

Onde $Q(S_t, A_t)$ é o valor do estado S_t segundo a ação A_t. R_{t+1} é o valor da recompensa recebida pelo agente durante a execução da ação A_t. $max_a(S_{t+1}, A_{t+1})$ é a ação de maior valor que se tem no próximo estado.

A aplicação da técnica é vista através do algoritmo abaixo.

Algoritmo de aplicação do *Q-Learing*

Iniciar $Q(s, a)$, $A(s)$ e $Q(estado\ terminal, \cdot) = 0$

para i **:** número de episódios **faça**
    Iniciar S
    Escolher A de S usando a política escolhida
    para i **:** número de iterações **faça**
        Escolher A' de S' usando a política derivada de Q
        $Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R + \gamma max_a Q(S_{t+1}, A_{t+1}) - Q(S, A)]$
        $S = S'$
    enquanto S != estado terminal
fim

6.3.6 DEEP Q-LEARNING

Deep Q-Learning, ou DQN, é uma das técnicas consideradas mais atuais dentro da área de aprendizado por reforço. Como dito anteriormente, tratar um problema sempre como sendo tabular ou por outro método discreto leva a um custo computacional muito alto. Isso ocorre pelo número de estados sempre aumentarem de maneira exponencial a medida que aumenta-se a complexidade do problema. Nesta implementação, é necessário que o agente passe por cada um dos estados um número determinado de vezes para que os valores das ações possam convergir. Isso naturalmente requer muito tempo.

O *DQN* trata os problemas de forma diferenciada. Sua principal característica é a junção das propriedades vistas em *Q-learning* com as vistas em redes neurais artificiais. Isso faz com que o algoritmo deixe de ser discreto para ser um aproximador de função, ou seja, ele não terá necessariamente que visitar todos os estados para que possa achar valores satisfatórios para cada um deles. Com a utilização de redes neurais e a extração das características do sistema, ele naturalmente já irá aproximar o valor do estado que esta visitando naquele instante.

A diferença principal para o método usual de aprendizado supervisionado esta em que a atualização dos pesos com a utilização do *backpropagation* é feita justamente através de uma parcela da equação descrita pelo *Q-learning*, em 6.29. Interpretando a equação como sendo a diferença entre o valor máximo encontrado entre as ações no próximo estado estado S_{t+1}, $R_{t+1} + max_a Q(S_{t+1}, A_{t+1})$, e o valor pertencente ao estado atual $Q(S_t, A_t)$, pode-se ter como equação de perda a vista em 6.27.

$$E = \left(R_{t+1} + max_a Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right)^2 \tag{6.26}$$

Essa diferença esta no centro de como é feita a aprendizagem da rede através do *Q-Learning*. Ela é propagada somente através da ação selecionado pela política respectiva no estado S, sendo os outros erros colocados como zero, para facilitar a convergência da rede. Há mais técnicas que também podem ser utilizadas para este fim facilitador, como a já mencionada *experience replay*, ou a técnica de *priorityzed experience replay* [27].

Em um problema caracterizado como *end-to-end*, que é como será abordado neste projeto o VSSS, as características são extraídas através dos pixels ou través da representação dos elementos presentes no estado. Estes irão servir de entrada para a rede

neural e ao final do processo de *feedforward* será utilizado, como dito, a equação 6.27 para realizar o *backpropagation*.

Podem ser utilizadas tanto a rede convolucional como a *MLP*, e a representação gráfica para cada uma das implementações pode ser vista através da figura 6.

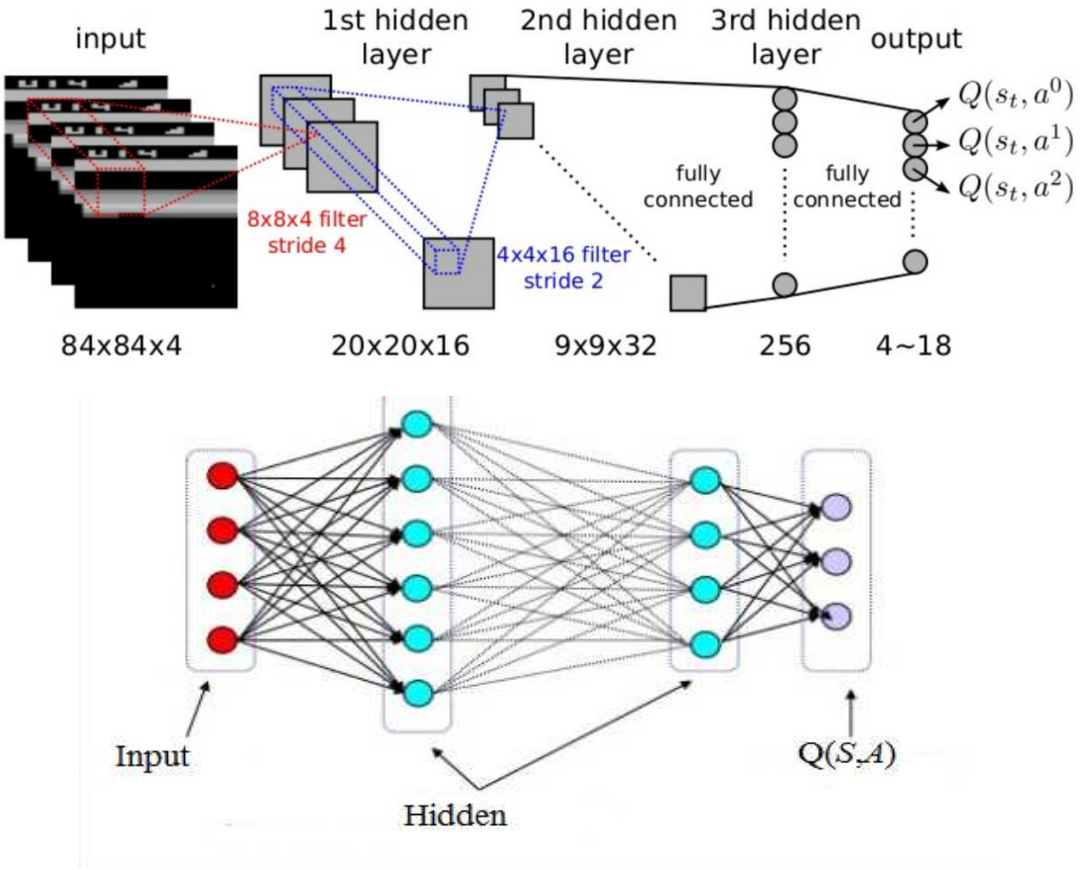


Figura 6.13 - Acima esta a representação da DQN através de redes convolucionais e abaixo através de MLP.

Mesmo a estrutura sendo mostrada de forma diferente, pode-se assumir o mesmo algoritmo para ambos, visto que os dois tem o mesmo princípio. A única diferença na implementação fica por conta de que, quando se trabalha com input por pixels, utiliza-se a rede convolucional. Porém, quando o input é uma tabela, por exemplo, utiliza-se a MLP.

O algoritmo para a técnica [21] pode ser visto abaixo.

Algoritmo de aplicação do *DQN* com *experience replay*

Iniciar a função de Q com pesos randômicos
Iniciar memória de replay D para uma capacidade N

para i **:** número de episódios **faça**
Iniciar $s_1 = \{x_1\}$ e pré processar em $\phi_1 = \phi(s_1)$
para i **:** número de iterações **faça**
Com probabilidade ε selecionar uma ação randômica a_t
senão selecionar $a_t = max_a Q^*(\phi(s_1), a, \theta)$

Realizar a ação a_t no simulador e observar R e a imagem x_{t+1}
Atribuir $s_{t+1} = s_t, a_t, x_{t+1}$ e pré processar $\phi_{t+1} = \phi(s_{t+1})$
Guardar a transição $(\phi_t, a_t, r_t, \phi_{t+1})$ em D
Retirar amostra *mini-batch* de D com as propriedades $(\phi_j, a_j, r_j, \phi_{j+1})$

Atribuir y_j $\begin{cases} r_j & para\ um\ estado\ terminal \\ r_j + \gamma max_a Q(\phi_{j+1}, a'; \theta) & para\ um\ estado\ não\ terminal \end{cases}$

Executar o gradiente descendente com
$(y_j - Q(\phi_j, a_j, \theta))^2$ de acordo com a equação 6.31
fim
fim

Apesar de o *experience replay* ter sido colocado como opção de melhora de performance, a realidade é que ele é uma parte necessária do algoritmo, e tem como objetivo retirar a tendência da rede de decorar, e não aprender, a sequência de passos correntes, sendo muitos raros os casos onde este não é necessário.

Uma de suas principais desvantagens é sua convergência de valores muito lenta. As redes neurais aplicadas para esta finalidade tem um processo de aprendizado muito lento, o que impede a aplicação descrita como *end-to-end* até mesmo não sendo em tempo real. Na verdade, para o algoritmo original foram necessários 38 dias para se aprender os jogos de Atari propostos [21]. Assim, por opção de projeto, foi escolhida uma implementação de MLP para a comparação proposta entre *Deep Q-Learning* e *SARSA*.

7. DEEP Q-LEARNING APLICADO NO SIMULADOR

Agora que foram introduzidas as técnicas utilizadas neste projeto, podemos agora ver de fato sua implantação no simulador VSSS. O objetivo para esta proposta é, como dito anteriormente, aplicar as técnicas para que o agente tome suas decisões do forma autônoma e consiga aprender a executar sua função de maneira otimizada e generalizada para qualquer que seja o estado descrito em jogo.

A seguir será explicada a aplicação das técnicas implantadas mais profundamente.

7.1 DESCRIÇÃO DA ABORDAGEM

Na categoria VSSS, a principal função do agente é chegar até a bola. A implementação dessa função se faz necessária pois a partir dela é que se consegue executar ações mais complexas durante o jogo.

A categoria permite implementação de qualquer lógica dentro para que o agente tomar suas decisões, mas não é difícil de perceber que a categoria envolve um imenso número de estados para se prever durante um jogo. Dessa forma, é possível se concluir que é difícil de se desenvolver um algoritmo que envolva uma aproximação da tomada ótima de decisão durante um jogo da categoria.

Com o objetivo de realizar essa aproximação, esta sessão irá discutir a implementação do algoritmo *Deep Q Learning* como algoritmo de inteligência para o agente executar a tarefa de chegar até a bola, qualquer que seja o estado que ele esteja no momento do jogo.

Como descrito anteriormente, o artigo original que descrevia o algoritmo implantado aqui tratava do problema de aprendizagem de jogos de Atari, e levou 38 dias para executar essa tarefa utilizando-se alguns processos de paralelização [21]. Assume-se, aqui, que um jogo de VSSS tem um número de estados e número de ações possíveis maiores que o artigo original. Devido ao custo computacional apresentado e ao *hardware* disponível para a realização dos processos que serão discutidos nessa sessão, simplificações são vistas como necessárias, levando em consideração também que esta é uma abordagem inicial ao problema.

Para limitação do espaço de estados, foi utilizado um formato de tabela para o campo, no qual cada uma das células da tabela é um estado onde o robô pode estar.

Cada um dos obstáculos, que representam os outros agentes, as paredes do campo e a bola também ocupam uma célula nesta abordagem. Um exemplo é visto através da figura 7.1.

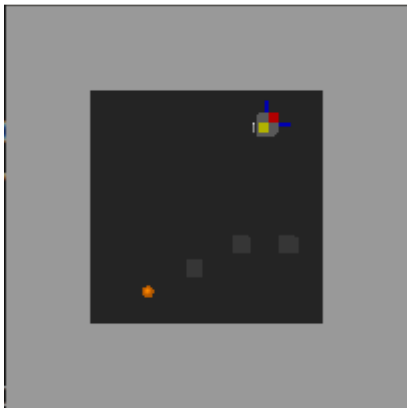


Figura 7.1 - Mapa de exemplo para solução do DQN

A tabela foi limitada para a dimensão de 12x12, sendo ela estática para averiguar a execução da tarefa de forma satisfatória em um modelo mais simples. O agente foi programado para tomar ações de forma determinística, não estocástica como ocorre durante um jogo normal devido aos diversos fatores influentes.

É importante mencionar que a abordagem tabular para em conjunto com o *DQN* não é o método tradicional. Porém a apresentação dos estado é feita de forma semelhante ao do artigo original, onde os pixels envolvidos na descrição do problema eram utilizados como *inputs* para a rede. Assim, cada um dos objetos em campo possuem uma identificação, sendo a bola representada pelo par real, a parede pelo 0.8, os obstáculos por 0.6, o agente é o 0.4 e os espaços restantes por 0.2. Isso se assemelha com uma imagem em preto e branco com *inputs* normalizados.

Essa abordagem foi implementada com o objetivo é fazer com que a rede neural atinja um ponto de ótima local para o estado apresentado para o agente, de forma a facilitar a convergência e diminuir o tempo de processamento do algoritmo.

A função de recompensa utilizada apresenta os valores de +1 quando o agente atinge a abola, -1 quando atinge a parede e -0.3 quando atinge os obstáculos em campo. Cada ação tomada pelo agente tem recompensa de 0. Esta análise foi feita segundo a lógica de que é pior ficar encostado contra a parede do que contra um agente adversário, pois se estará obstruindo as ações deste também. Além disso, quando o agente realiza 150 ações e não chega até a bola, é atribuído àquela política um valor acumulativo de -200, o qual não é propagado para a rede, visto que esta não tem noção da quantidade de tempo passada até aquele instante.

As ações possíveis para o agente em campo são norte, sul, leste e oeste, sendo estas determinísticas, como dito. Isso significa que, uma vez que o agente escolha ir para norte, por exemplo, ele terá probabilidade 1 de ir naquela direção.

O algoritmo utilizado envolve a implementação apresentada da mesma forma como no algoritmo *DQN* apresentado anteriormente. Apesar das dificuldades em encontrar os parâmetros ideais para a convergência da rede o algoritmo apresentado retornou o comportamento desejado.

A origem dessa dificuldade regrata a diferença entre o VSSS dos jogos de Atari pode ser citada como que, para os jogos, o agente recebe as recompensas que lhe cabem de forma mais contínua. Para o problema abordado, muito dificilmente o agente consegue chegar até a bola, que é onde ganha a maior das recompensas, e é onde efetivamente aprende o caminho que deve tomar. Com isso, foi implementada junto ao algoritmo a técnica conhecida como *priorityzed experience replay* [27].

Outro motivos para a dificuldade encontrada se deve ao fato de que, conforme o decaimento da *policy* ϵ-greedy, a rede vai especificando qual a ação de máximo valor a ser tomada. Isso faz com que, quando já no final do processo de aprendizagem, a rede neural entenda que é melhor para o agente ficar parado contra a parede, por exemplo. E como a probabilidade de não executar a ação de retorno maior é muito pequena, ele repete o estado por intervalos muito longos, enchendo a memória do *experience replay* com esses estados e piorando sua *policy*.

Para solucionar esse problema foi implementada uma condição de que o estado atual não pode ser igual ao anterior para que o algoritmo armazenasse estados a sua memória de *experience replay*. Além disso, para evitar que estados repetitivos preenchessem a memória do algoritmo, foi utilizada uma lógica para identificar esses estados consecutivamente. Assim, não são armazenados.

A próxima sessão mostrará os resultados que o algoritmo com as adaptações feitas apresentou.

7.2 RESULTADOS OBTIDOS

Para a obtenção dos resultados foram utilizadas diversas topologias, hiper parâmetros e funções de recompensa para o algoritmo. Porém nesta sessão somente irão ser demonstradas as configurações que tiveram melhor performance entre os dois formatos de ANN. Estes apresentados serão apresentados a seguir.

Como dito foram utilizadas tanto redes MLP quanto convolucionais. Para a MLP a topologia utilizada foi de [100,200,4], sendo suas funções de ativação de *Leaky Relu* na segunda camada e função identidade para a camada final. Já a convolucional utilizou filtros de 3x3 e com 9 *feature maps*. A segunda camada utilizou filtros de 2x2 e com 10 *feature maps*. Já a terceira utilizou filtros de 2x2 e com 14 *feature maps*. Todas utilizaram função de ativação *Leaky Relu*. A MLP final utilizou função de ativação como identidade, com 4 *outputs*.

Para ambas as redes foi utilizada a técnica de otimização de RMSProp, já discutida em sessão anterior, com η de $1 \cdot 10^{-7}$ e ε de $1 \cdot 10^{-8}$, sendo o coeficiente α de 0.95.

Outras variáveis para a execução da aprendizagem envolvem também o número de épocas total, de 30000, o número de *batch samples*, de 55, e o total de memória de algoritmo, que é de 10000 estados.

Adotados os valores mencionados, o algoritmo já pode ser rodado para obtenção dos resultados finais, retirando para esta análise um gráfico de curva de erro e um de total obtido pela política conforme cada época para ambas topologias. A metodologia utilizada para medir a eficiência e diminuir a variância dos resultados foi a de simular 15 sequências de aprendizado independentes com cada modelo e tirar sua média para a criação de cada um dos gráficos.

7.2.1 MULT LAYER PERCEPTRON

Apesar de, originalmente, a técnica ser utilizada apenas com redes convolucionais, a utilização deste tipo de rede se justifica por conta do método tabular apresentado e pela maior facilidade de se visualizar a convergência em *MLPs.*

A análise do gráfico da curva de erro mostra, em 7.3, uma convergência para um erro mínimo. Porém, apesar de a quantidade de épocas percorridas ter sido compreendida como suficiente para a realização do aprendizado, ele atinge sempre uma ótima local se os valores de *mini-batch* não apresentarem um gradiente o mais determinístico possível, de outro modo o agente acaba não chegando a atingir seu objetivo. Por isso a decisão na adoção de um *mini-batch* de 55. Isso é visto através da figura que demonstra o trajeto do agente em 7.4, que foi feito utilizando uma das 5 redes que passou pelo aprendizado.

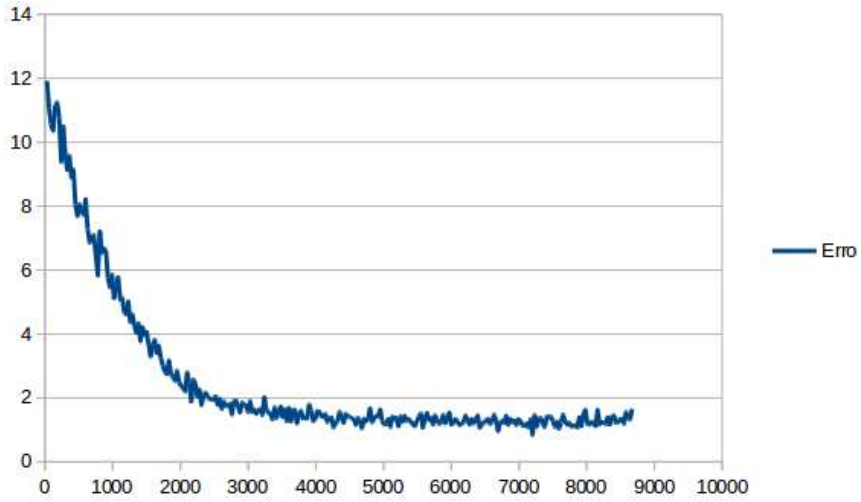
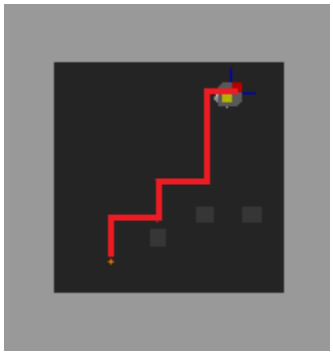


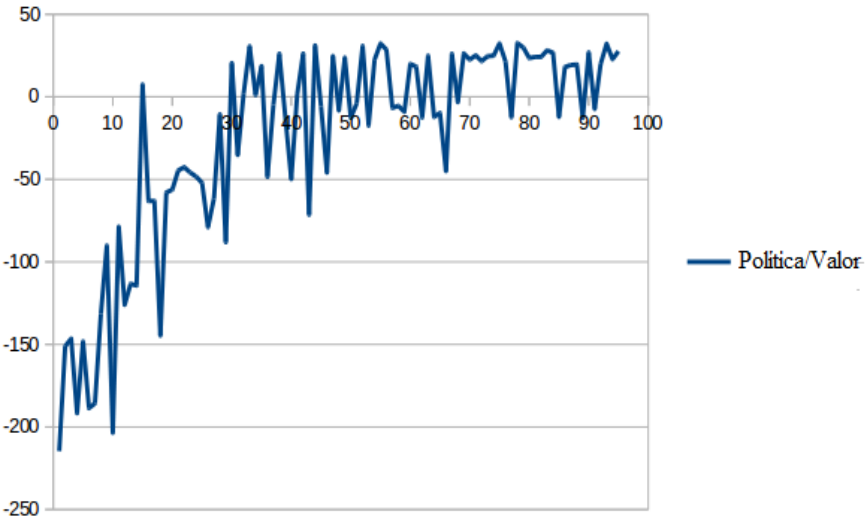
Figura 7.2 - Gráfico de erro mostrado da MLP



7.3 - Figura demonstrando o trajeto apresentado pela ANN

Em 7.2 é visto que a rede chega a um momento de estabilização nos seus erros. Apesar disso, esse é apenas o momento em que a política ε-greedy apresenta uma probabilidade de 0.7. Os outputs da média das redes nesse momento não apresenta ainda valores próximos ao ótimo, mas como visto através da sessão 6.3, isso não é necessário para que este apresente uma política ótima. Entretanto, podemos ver através de 7.3 que este já é um momento suficiente para que o agente alcance seu objetivo.

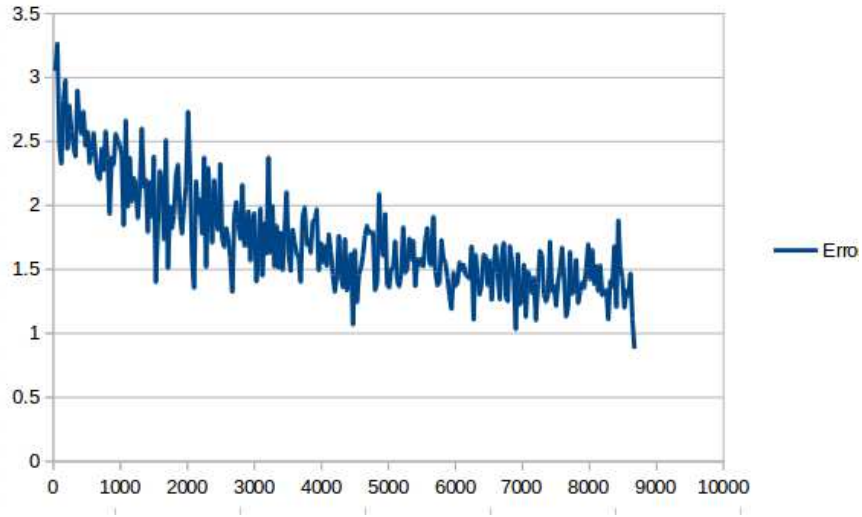
O gráfico que mostra a recompensa ganha com as políticas adotadas durante as épocas percorridas mostra o agente para de atribuir o valor mencionado de -200 para quando não acha a bola por um número arbitrário de episódios. Isso pode ser visto por conta do cálculo de políticas com frequência mais positiva a partir de 50 episódios, tornando o agente mais eficiente, apesar de ser possível que o afunilamento da variação apresentada é uma tendência no gráfico. Isso é mostrado na figura 7.5.



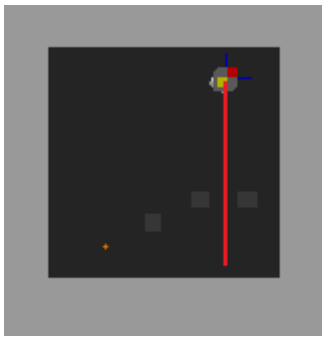
7.4 - Gráfico de valores da política adotada ao longo do tempo

7.2.2 CONVOLUTIONAL

Como dito anteriormente, esse tipo de rede tem tendência de convergência mais lenta que as *MLPs*. Por isso, com a rede convolucional os resultados se diferenciaram muito dos obtidos com a utilização da MLP. O gráfico do erro, apesar de apresentar comportamento similar, apresenta uma variação maior do que a vista em 7.2. Porém é necessário também que se veja a diferença de escala entre ambos. Entretanto, é sabido que seus hiperparâmetros diferem bastante das *MLPs*, mesmo quando se tratando do mesmo problema. As são mostradas respectivamente em 7.5 e 7.6.



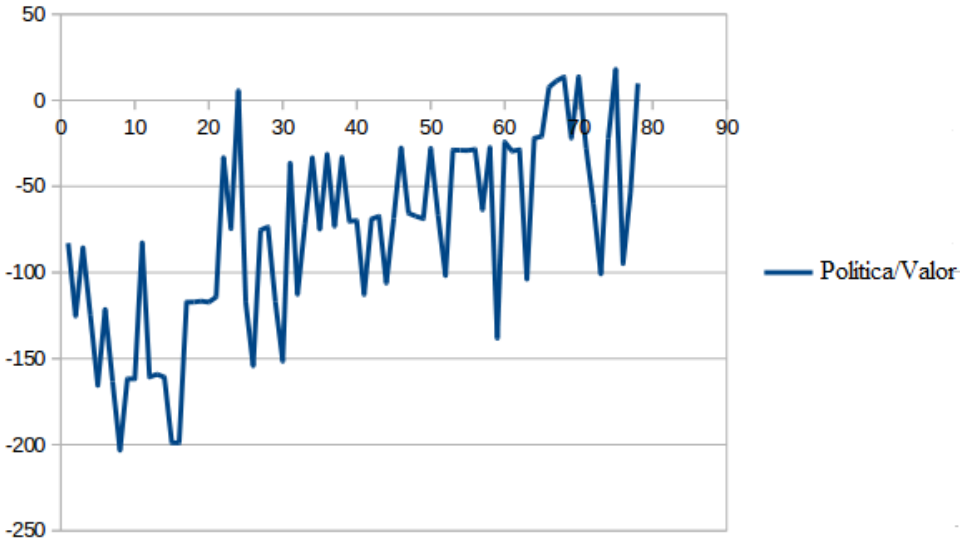
7.5 - Gráfico do erro com a rede CNN



7.6 - Trajetória do agente com a CNN

O gráfico de erro foi colhido a partir de 5 amostras distintas de aprendizagem com a CNN, e utilizou-se o mesmo método utilizado com as *MLPs*. Isto é, foram colhidos os dados até o momento em que a política ε-greedy atingiu probabilidade de 0.7. Apesar disso, é visto que o gráfico de erro ainda não obteve uma estabilidade, assim como em 7.6 a trajetória apresentada ainda não é a ideal. Porém, a média dos *outputs* obtidos entre os testes indica que a convergência se mostra correta. Isto é, as direções de sul e oeste apresentam valores mais altos do que as outras duas, o que condiz com o problema apresentado.

Com isso, o gráfico de recompensa segundo as políticas também se distanciou entre os dois modelos utilizados, como pode ser visto em 7.7. Como visto através da figura 7.6, o agente ainda apresenta trajetória não coerente com a solução, o que faz com que muitas vezes ele agregue ao valor da política o valor de -200 mencionado anteriormente.



7.7 - Gráfico dos valores das políticas com a rede CNN

7.3 CONSIDERAÇÕES

Com todos os inúmeros testes feitos e também das adaptações feitas, como explicado no início desta seção, os resultados obtidos, apesar de promissores para o problema apresentado, ainda não foram satisfatórios. Isso por que o VSSS é um sistema dinâmico e de múltiplos estados, o que impede uma implementação tabular e estática como a apresentada ao longo do tópico. Também há o fato de que o VSSS é um problema considerado como multiagentes. O *DQN* foi inicialmente desenvolvido em jogos de Atari com jogos *single player*, o que torna difícil sua adaptação para um sistema de muitos agentes como o VSSS.

Para as dificuldades apresentadas ao longo dos inúmeros testes feitos até que se pudesse chegar a uma solução para o problema, há algumas explicações para o comportamento apresentado. Uma delas é que a bola esta em um ponto muito próximo da parede. Levando-se em conta que a parede representa a menor recompensa possível e esta presente em uma área maior do campo do que a bola, que representa a maior recompensa, o agente não vê vantagem suficiente de se chegar até a bola, visto que na maior parte das vezes ele acaba batendo contra a parede.

Apesar da implementação do *priorityzed experience replay*, essa abordagem ainda precisou de uma análise mais aprofundada para agir mais efetivamente no algoritmo. Isso por que, apesar de o agente estar realizando seus gradientes sempre com uma taxa destinada a episódios ligados a ações que levassem até a bola, sua estabilidade estava ligada ao número de estados presentes no *mini-batch*.

Outra consideração a se fazer é que o ponto de objetivo onde se ganha a maior recompensa é dificilmente acessado pelo agente. Mesmo com as adaptações feitas, também precisa se levar em consideração os estados que dão acesso a esse estado terminal, o que não foi feito durante a implementação do *pryoritized replay*. Eles fazem tanto parte da aprendizagem quanto este último.

Por fim, há a própria abordagem ao problema conhecido como *path finding*. Os jogos de Atari não tem por natureza um problema semelhante ao abordado. Neles, o agente sempre ocupa uma intervalo de posições, em sua maior parte, fixa na tela. Mas no *path finding* o agente pode ocupar qualquer estado. Isso pode dificultar a convergência da rede neural em direção a uma solução para o problema.

8. TRABALHOS FUTUROS

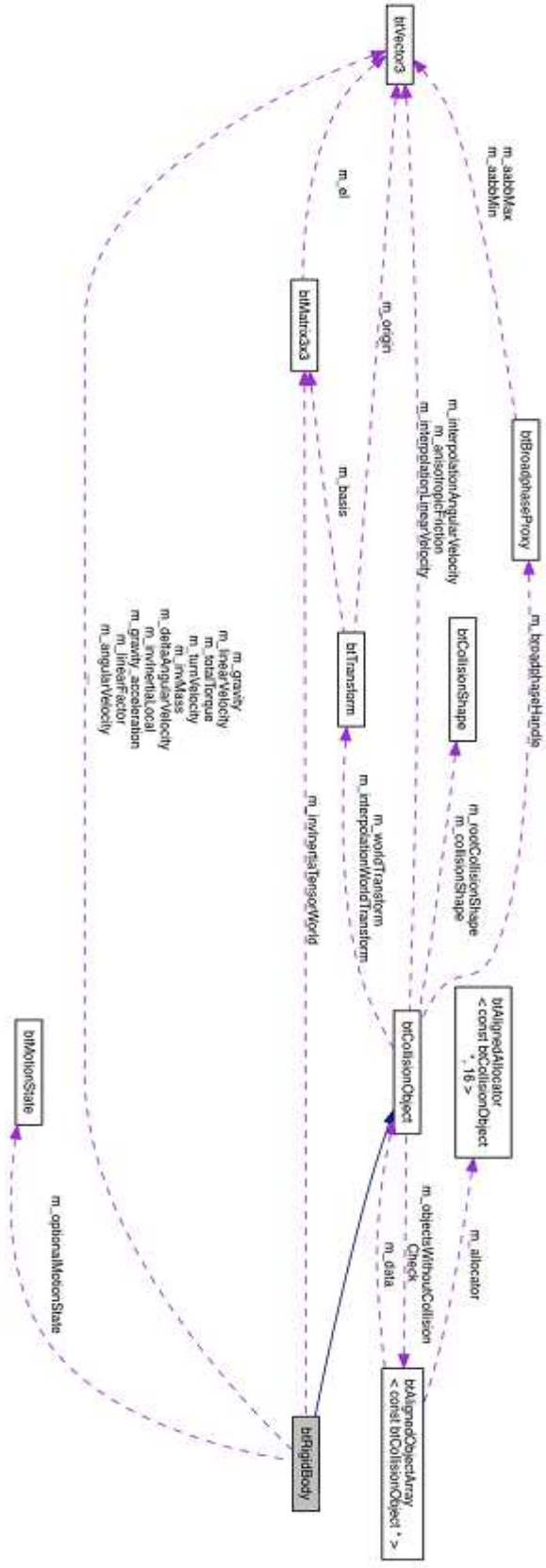
O algoritmo *DQN* hoje é visto como o mais promissor e atual dentre os algoritmos disponíveis para aprendizado de máquina. Em conjunto com a técnica de *Monte Carlo Tree Search*, pode ser muito útil na tarefa de explorar espaços de estado muito grandes, como visto em [10]. A utilização de redes neurais para tomada de decisão é vista como uma tendência pela sua eficiente característica de aproximação da função ótima.

A desvantagem, entretanto, de aplica-lo em um sistema multi agentes como é o VSSS é que, justamente, ele não possui boa performance para cada um dos agentes em campo. O algoritmo não é eficiente, ainda, como método de tomada de decisão conjunta, o que no caso do VSSS e de muitos outros problemas atuais é muito necessário.

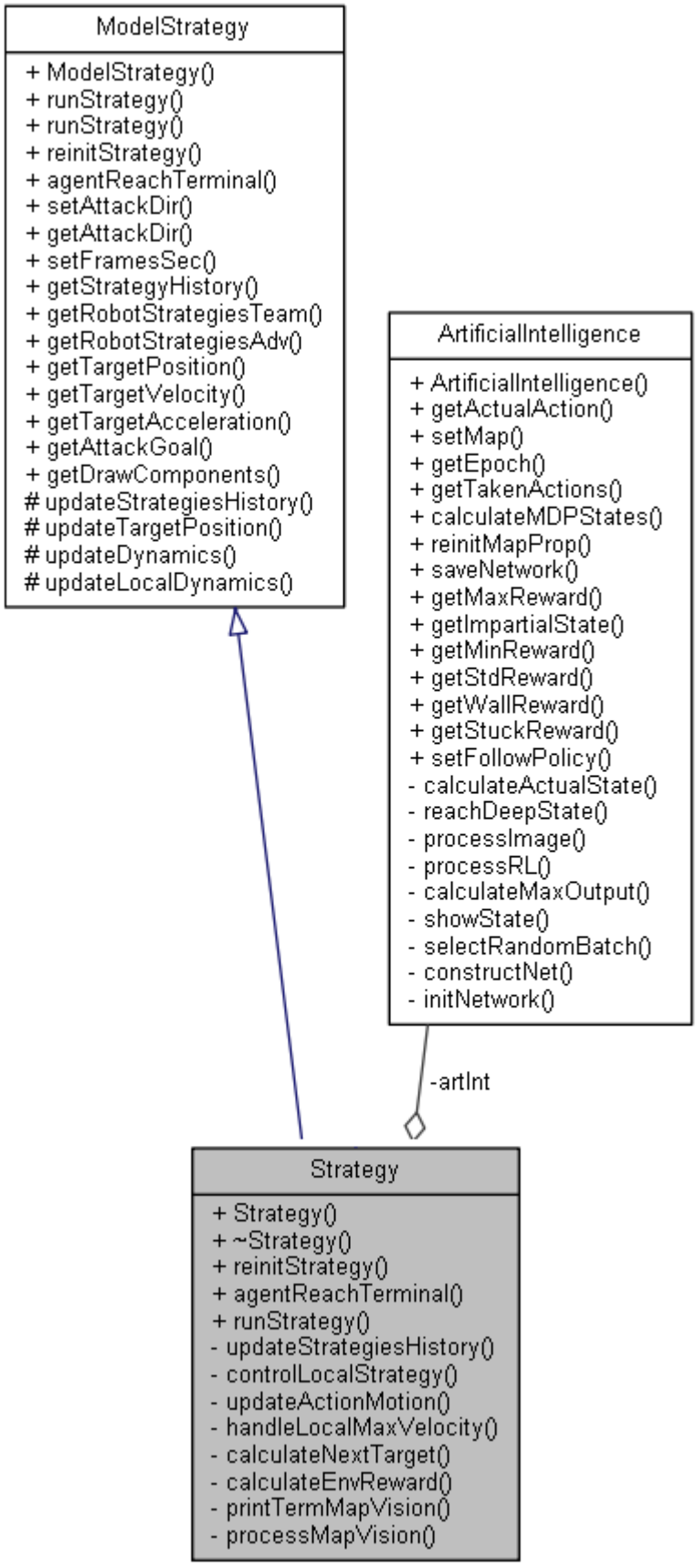
Algumas abordagens nesse sentido foram tentadas, mas são muito recentes, como descrito em [34, 35]. Essas são abordagens ainda iniciais e que precisam de um tempo para que possam evoluir.

O objetivo, então, é realizar pesquisas nesse sentido que se utilizem justamente da característica das redes neurais para sistemas multi agentes que possam cooperar entre si para que atinjam uma recompensa maior, o que é visto hoje como uma tendência da área de aprendizado de máquina. Isso envolve também a pesquisa relacionada a sistemas mais complexos e de maior espaço de estados, como as tentativas recentes de aplicação das técnicas descritas no jogo *Star Craft*, visto como o próximo desafio a ser superado pela área [35]

ANEXO A


btVector3
m_aabbMax
m_aabbMin
m_el
m_origin
btMatrix3x3
m_interpolationAngularVelocity
m_anisotropicFriction
m_interpolationLinearVelocity
m_basis
btBroadphaseProxy
m_broadphaseHandle
btCollisionShape
m_rootCollisionShape
m_collisionShape
btTransform
m_worldTransform
m_interpolationWorldTransform
m_invInertiaTensorWorld
m_gravity
m_linearVelocity
m_totalTorque
m_turnVelocity
m_invMass
m_deltaAngularVelocity
m_invInertiaLocal
m_gravity_acceleration
m_linearFactor
m_angularVelocity
btAlignedAllocator
< const btCollisionObject
*, 16 >
m_allocator
btCollisionObject
m_objectsWithoutCollision
Check
m_data
btAlignedObjectArray
< const btCollisionObject * >
btMotionState
m_optionalMotionState
btRigidBody

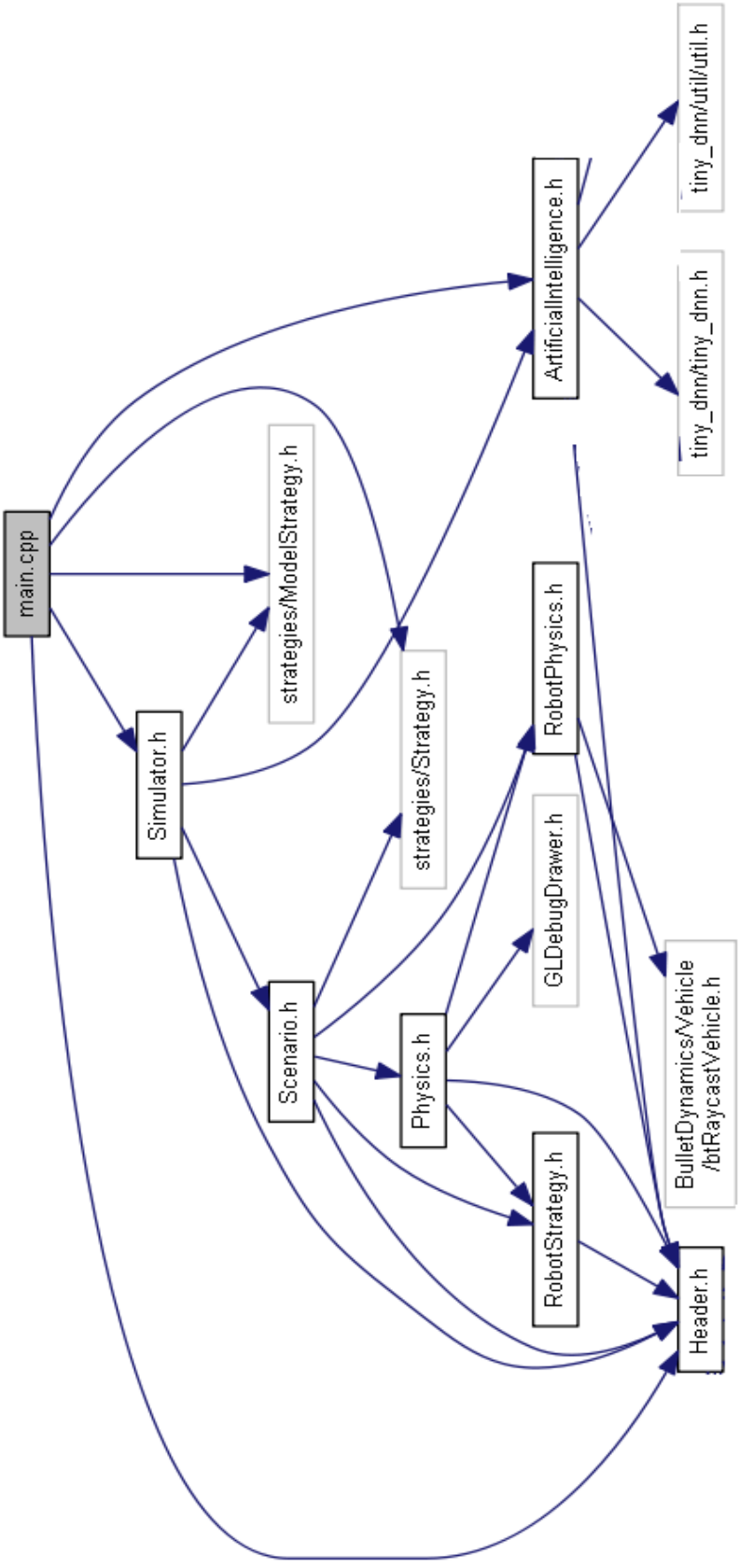
ANEXO B


ModelStrategy
+ ModelStrategy()
+ runStrategy()
+ runStrategy()
+ reinitStrategy()
+ agentReachTerminal()
+ setAttackDir()
+ getAttackDir()
+ setFramesSec()
+ getStrategyHistory()
+ getRobotStrategiesTeam()
+ getRobotStrategiesAdv()
+ getTargetPosition()
+ getTargetVelocity()
+ getTargetAcceleration()
+ getAttackGoal()
+ getDrawComponents()
# updateStrategiesHistory()
# updateTargetPosition()
# updateDynamics()
# updateLocalDynamics()
ArtificialIntelligence
+ ArtificialIntelligence()
+ getActualAction()
+ setMap()
+ getEpoch()
+ getTakenActions()
+ calculateMDPStates()
+ reinitMapProp()
+ saveNetwork()
+ getMaxReward()
+ getImpartialState()
+ getMinReward()
+ getStdReward()
+ getWallReward()
+ getStuckReward()
+ setFollowPolicy()
- calculateActualState()
- reachDeepState()
- processImage()
- processRL()
- calculateMaxOutput()
- showState()
- selectRandomBatch()
- constructNet()
- initNetwork()
-artInt
Strategy
+ Strategy()
+ ~Strategy()
+ reinitStrategy()
+ agentReachTerminal()
+ runStrategy()
- updateStrategiesHistory()
- controlLocalStrategy()
- updateActionMotion()
- handleLocalMaxVelocity()
- calculateNextTarget()
- calculateEnvReward()
- printTermMapVision()
- processMapVision()

ANEXO C


RobotStrategy
- id
- maxCommand
- command
- stopped
- robotPos
- robotStepTarget
- linVel
- linAce
- localFront
- localRight
- angVel
- angAce
- localFunction
- maxVelocity
- maxTargetAngle
- actDistToTarget
- targetPos
- motion
- invertFront
+ RobotStrategy()
+ setStandardMotion()
+ getStandardMotion()
+ setStopped()
+ getStopped()
+ updateCommand()
+ getCommand()
+ setPosition()
+ getPosition()
+ getLocalFront()
+ getLocalRight()
+ getId()
+ getLocalFunction()
+ getFieldAngle()
+ getPointAngle()
+ getTargetAngle()
+ getTargetDistance()
+ getMaxAngToTarget()
+ getMaxCommand()
+ getActDistToTarget()
+ getLinearVelocity()
+ getLinearAcceleration()
+ getAngularVelocity()
+ getAngularAcceleration()
+ setLinearVelocity()
+ setLinearAcceleration()
+ setAngularVelocity()
+ setAngularAcceleration()
+ setLocalFront()
+ setLocalRight()
+ getTargetPosition()
+ setTargetPosition()
+ getStepTargetPosition()
+ setStepTargetPosition()
+ setLocalFunction()
+ invertLocalFront()
+ getInvertLocalFront()

ANEXO D


main.cpp
Simulator.h
strategies/ModelStrategy.h
strategies/Strategy.h
ArtificialIntelligence.h
tiny_dnn/util/util.h
tiny_dnn/tiny_dnn.h
Scenario.h
Physics.h
RobotPhysics.h
GLDebugDrawer.h
RobotStrategy.h
BulletDynamics/Vehicle
/btRaycastVehicle.h
Header.h

BIBLIOGRAFIA

[1] CBR: IEEE Very Small Size. Disponível em: http://www.cbrobotica.org/?page_id=81&lang=pt. Acesso em:12 out. 2016.

[2] ZICKLER, Stefan. "Physics Based Robot Motion Planing in Dynamic Multi-Body Environment", Computer Science Department, Carnegie Mellon University, 2010.

[3] GOULD, Harvey. "An Introduction to Computer Simulation Methods", 2005.

[4] FOX, Robert. "Introdução a Mecânica dos Fluídos", LTC Editora, 2014.

[5] GUERRA, Patrícia. "Linear Modeling and Identification of a Mobile Robot With Differencial Drive", Departament of Computer Engineering and Automation, UFRN, 2004.

[6] MERIAM. "Mecânica para Engenharia Dinâmica", LTC Editora, 2016.

[7] DICKINSON, Chris. "Learning Game Physics with Bullet Physics and OpenGL" ,Packt Publishing Ltd, 2013

[8] P. F. Muir, C. P. Neuman. "Kinematic Modeling of Wheeled Mobile Robots" ,Neuman - Journal of Field Robotics, 1987.

[9] W. D. J. Dierssen. "Motion Planning in a Robot", Department of Computer Science, University of Twente, 2003.

[10] D. Silver. "Mastering the game of Go with deep neural networks and tree search", Nature, 2016.

[11] “MuJoCo physics engine.” [Online]. Disponível em: www.mujoco.org

[12] An Overview of Gradient Descent Optimization Algorithms. Disponível em: http://sebastianruder.com/optimizing-gradient-descent/. Acesso em: 30 de abril de 2017.

[13] “PhysX physics engine.” [Online]. Available: www.geforce.com/ hardware/technology/physx

[14] “Bullet physics engine.” [Online]. Disponível em www.bulletphysics.org

[15] “Havok physics engine.” [Online]. Disponível em www.havok.com

[16] “Open dynamics engine.” [Online]. Disponível em http://ode.org

[17] "OpenGL documentation". [Online]. Disponível em https://www.opengl.org/

[18] "Neural Networks and Deep Learning". Disponível em: http://neuralnetworksanddeeplearning.com. Acesso em: 30 de abril de 2017.

[19] "Convolutional Neural Networks for Visual Recognition". Disponível em: http://cs231n.github.io/convolutional-networks. Acesso em: 30 de abril de 2017.

[20] "Convolutional Neural Networks". Disponível em: http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork. Acesso em: 30 de abril de 2017.

[21] V. Mnih, K. Kavukcuoglu, D. Silver e A. Graves. "Playing Atari with Deep Reinforcement Learning", DeepMind Technologies, 2013.

[22] M. A. Goodrich. "Potential Fields Tutorial", Disponível em: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.3259&rep=rep1&type=pdf. Acesso em: 30 de abril de 2017.

[23] R. S. Sutton e A. G. Barto. "Reinforcement Learning: An introduction", Division of Neuroscience, Baylor College of Medicine, Houston, 1998.

[24] J. Chen e L. Yang. "Combining Fully Convolutional and Recurrent Neural Networks for 3D Biomedical Image Segmentation", Universidade de Notre Dame, 2016.

[25] "AI: An Introduction to Neural Networks". Disponível em: https://www.codeproject.com/articles/16419/ai-neural-network-for-beginners-part-of. Acesso em: 30 de abril de 2017.

[26] V. Dumoulin e F. Visin. "A guide to convolution arithmetic for deep learning". arXiv:1603.07285, 2016.

[27] T. Schaul, J. Quan. "Prioritized Experience Replay". arXiv:1511.05952, 2016.

[28] S. Ioffe e C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", arXiv:1502.03167, 2015.

[29] RM French."Catastrophic Forgetting in Convencionist Networks", Trends in cognitive sciences, 1999.

[30] R. Hecht e M. Nielsen "The Theory of the Backpropagation Neural Network", Neural Networks, 1988.

[31] L. Chen, G. Papandreou, I. Kokkinos, K. Murphy e A. Yuille. "Semantic image segmentation with deep convolutional nets and fully connected crfs". In ICLR, 2015.

[32] I. Goodfellow, Y. Bengio e A. Couville. "Deep Learning". MIT Press, 2016.

[33] D. L. Poole e A. K. Mackworth. "Artificial Intelligence - Foundations of Computational Agents", University of British Columbia, 2010.

[34] A. Tampuu. "Multiagent Cooperation and Competition with Deep Reinforcement Learning", arXiv:1511.08779, 2015.

[35] " DeepMind and Blizzard to release StarCraft II as an AI research environment". Disponível em: https://deepmind.com/blog/deepmind-and-blizzard-release-starcraft-ii-ai-research-environment. Acessado em: 30 de abril de 2017.

[36] J Kober, JA Bagnell, J Peters. "Reinforcement learning in robotics: A survey", The Internacional Journal of Machine Learning, 2013.

[37] A. Coates e A. Ng. "Deep learning with COTS HPC systems", Stanford University Computer Science Dept, 2013.

[38] A. Bai, F. Wu e X. Chen. "Online planning for large MDPs with MAXQ decomposition", Proceedings of the 11th International Conference, 2012.

[39] "Robot Soccer Simulation 2D". Disponível em: http://wiki.robocup.org/Soccer_Simulation_League. Acesso em: 30 de abril de 2017.

[40] J. Thompson. "Accelerating Eulerian Fluid Simulation With Convolutional Networks", arXiv:1607.03597, 2017.

[41] J. Stewart. "Calculus II - Vol I", Editora Cengage Learning, 2013.

[42] R. G. Simmons. "Structured Control for autonomous robots", IEEE transactions on robotics and automation, 1994.

[43] T. Erez, Y. Tassa e E. Todorov. "Simulation Tools for Models-Based Robotics", University of Washington, Departments of Applied Mathematics and Computer Science & Engineering, 2015.

[44] "Scientific and Technical Academy Award for the development of Bullet Physics". Disponível em: http://bulletphysics.org/wordpress/?p=427. Acesso em: 30 de abril de 2017.

[45] "Mirosot: Micro Robot World Cup Soccer Tournament". Disponível em: http://www.fira.net/contents/sub03/sub03_3.asp. Acesso em: 28 out. 2014.

[46] Tieleman, T. and Hinton e G. (2012), Lecture 6.5 - rmsprop, COURSERA: Neural Networks for Machine Learning.